HAL open science

Interconnexion et routage dans les systèmes pair à pair

Salma Ktari

▶ **To cite this version:**

Salma Ktari. Interconnexion et routage dans les systèmes pair à pair. domain_other. Télécom ParisTech, 2009. English. NNT : . pastel-00005737

HAL Id: pastel-00005737

https://pastel.hal.science/pastel-00005737

Submitted on 19 May 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École Doctorale
d'Informatique,
Télécommunications
et Électronique de Paris

Thèse

présentée pour obtenir le grade de Docteur
de l'École Nationale Supérieure des Télécommunications

Spécialité : **Informatique et Réseaux**

SALMA KTARI

Interconnexion et routage dans les systèmes pair à pair

Soutenue le 14 Décembre 2009 devant le jury composé de

Isabelle Chrisment	Rapporteurs
Pascal Lorenz	
Maurice Gagnaire	Examinateurs
Sami Tabbane	
Yutaka Takahashi	Invité
Houda Labiod	Directeur de thèse
Artur Hecker	Co-directeur de thèse

À mon cher époux,

À ma famille.

Merci,

Au terme de ce travail, je tiens à remercier mes directeurs de thèse, madame Houda Labiod et monsieur Artur Hecker qui m'ont accompagné dans ma recherche ces trois années. Un grand merci à Artur pour sa grande disponibilité et son suivi sérieux et parfois exigeant, qui m’ont été d’une aide précieuse. Je remercie également Houda d’avoir encadré mon travail depuis le DEA. Je la remercie pour sa gentillesse, ses conseils et son soutien.

J’ai eu également le plaisir de collaborer avec la société WaveStorm pendant la première année de ma thèse. Je remercie toute l’équipe et particulièrement Franck Springinsfeld, Mathieu Zoubert et Alexander Casassovici.

Je souhaite aussi présenter mes remerciements aux membres du jury qui ont accepté d'évaluer mon travail. Je tiens à remercier chaleureusement professeur Isabelle Chrisment et professeur Pascal Lorenz pour leur lecture attentive de la thèse. Je remercie également professeur Maurice Gagnaire, professeur Sami Tabbane et professeur Yutaka Takahashi pour m’avoir fait l’honneur de participer au jury de ma thèse et pour et l’intérêt qu’ils ont porté à mon travail.

Un grand merci à tous mes amis du département INFRES. Je leur exprime ma profonde sympathie et leur souhaite beaucoup de réussite.

Enfin, j’ai une pensée toute particulière pour mon cher époux ainsi que toute ma famille. Je suis heureuse de leur dédier ce travail.

RESUME

Les systèmes pair-à-pair sont l'outil de choix pour réaliser un système informatique autonome tout en assurant sa haute disponibilité au coût relativement faible. Ces systèmes sont toutefois complexes à concevoir et posent divers problèmes liés à la gestion de l'espace virtuel (*overlay*) crée. Nous nous concentrons principalement sur deux aspects de ces environnements : l'organisation des nœuds dans l'*overlay* dynamique et l'organisation des données dans ce dernier.

Concernant l'organisation des nœuds dans l'*overlay*, nous proposons Power DHT, une nouvelle structure d'interconnexion et de routage. Partant de l'hétérogénéité observée dans les tables de hachage (DHTs) déployées, nous transformons dynamiquement la DHT vers une structure décentralisée, exhibant les propriétés d'un graphe sans échelle (distribution des degrés, faible diamètre). Nous exploitons les propriétés de cette nouvelle structure et implémentons à la fois le routage KBR (*Key Based Routing*), offrant un diamètre plus court à un coût de signalisation moindre, et le support de la diffusion efficace, réalisant ainsi des recherches floues.

Quant à l'organisation des données dans l'espace virtuel, nous employons la réplication pour améliorer la disponibilité et l'accessibilité des objets de l'*overlay* potentiellement instable. Nous avons implémenté et évalué différentes méthodes de réplication. Nous choisissons d'intégrer à notre structure la réplication symétrique. A partir de ces résultats, nous avons conçu un mécanisme d'inondation efficace pour les systèmes P2P structurés. Ce mécanisme, évalué sur notre plate-forme, exploite la structure de la DHT et les propriétés de la réplication symétrique pour permettre les recherches floues dans les DHTs, tout en limitant le coût de signalisation.

TABLES DES MATIERES

INTRODUCTION GENERALE **1**

CHAPITRE 1. VUE D'ENSEMBLE DES SYSTEMES PAIR A PAIR **7**

1.1 Terminologie **8**

1.2 Définition **9**

1.3 Différents niveaux de centralisation **9**
1.3.1 Systèmes P2P centralisés 10
1.3.2 Systèmes P2P distribués 10
1.3.3 Systèmes P2P décentralisés 13

1.4 Conclusion **13**

CHAPITRE 2. LES TABLES DE HACHAGE DISTRIBUEES **17**

2.1 Les tables de hachage distribuées **17**
2.1.1 Principe général 17
2.1.2 Propriétés 18

2.2 DHTs existantes **19**
2.2.1 Topologie en anneau 19
2.2.2 Topologie en arbre basée sur l'algorithme de Plaxton 21
2.2.3 Topologie en hypercube 22
2.2.4 Autres propositions 25
2.2.5 Tableau comparatif 27

2.3 Optimisations possibles **29**
2.3.1 Réseaux hétérogènes et réseaux hiérarchiques 29
2.3.2 Extension du voisinage 30
2.3.3 Duplication des données 31
2.3.4 Méthodes de recherches floues 33
2.3.5 Rapprocher le réseau physique du réseau logique 34
2.3.6 Equilibrage de charge 38

2.4 Conclusion **39**

CHAPITRE 3. REVERSE_DHT ET POWER_DHT : VERS UNE STRUCTURE DECENTRALISEE **43**

3.1 Observations & Motivations **44**

3.2 1ère étape : Reverse_DHT, approche de routage bidirectionnel **48**
3.2.1 Architecture Reverse_DHT 48
3.2.2 Application à Chord, Pastry et Kademlia 49
3.2.3 Evaluation de l'approche Reverse_DHT 53
3.2.4 Conclusion 56

3.3 2ème étape : Power_DHT, approche de routage 'Power Law' **58**
3.3.1 Modèle Barabási Albert pour la génération des graphes en loi de puissance 58
3.3.2 Architecture Power_DHT 60
3.3.3 Application à Chord, Pastry et Kademlia 62
3.3.4 Evaluation de l'approche Power_DHT 65

3.4 Conclusion **73**

CHAPITRE 4. OPTIMISATION DE L'INONDATION DANS LES DHTS PAR REPLICATION SYMETRIQUE **79**

4.1 1ère étape : Evaluation des méthodes de réplication dans les DHTs **81**
4.1.1 Méthodes de réplication dans les DHTs 82
4.1.2 Evaluation des méthodes de réplication 87
4.1.3 Conclusion 92

4.2 2ème étape : Optimisation de la recherche par inondation **94**
4.2.1 Détail de l'algorithme 94
4.2.2 *Sym_Flood* : Utilisation de la réplication symétrique pour limiter l'inondation 95
4.2.3 Synthèse 96
4.2.4 Evaluation de l'inondation Sym_Flood 97

4.3 Conclusion **100**

CONCLUSION GENERALE **103**

PUBLICATIONS **107**

BIBLIOGRAPHIE **108**

LISTE DES FIGURES

Figure 1. 1 Classification des systèmes P2P 14
Figure 2. 1 Extrait de (44) (a) Acheminement d'une requête dans un anneau simple (b) Calcul des fingers (c) Acheminement d'une requête par les fingers 20
Figure 2. 2 Extrait de (34) Table de routage du pair 10233102 avec b = 2 21
Figure 2. 3 Extrait de (51) Voisinage du nœud 0011. Les cercles en bleu représentent voisins sibling du nœud 0011. 24
Figure 2. 4 Extrait de (35) Exemple de route Kademlia : le nœud 0011 localise le nœud 1110 25
Figure 3. 1 Distribution des degrés entrants dans Chord et Pastry 44
Figure 3. 2 (a) Distribution quasi uniforme des zones de responsabilité, (b) Distribution non uniforme des zones de responsabilité dans Chord, (c) liens vers le nœud A, responsable de la plus grande zone 45
Figure 3. 3 Distribution des degrés entrant dans Kademlia 47
Figure 3. 4 (a) ajout des liens inverses depuis chaque finger, (b) nœuds reverse du point A (ronds blancs) 50
Figure 3. 5 Résultats de Reverse Chord 55
Figure 3. 7 Distribution des degrés dans Power_Kad 63
Figure 3. 6 Distribution des degrés dans Power_Pastry et Power_Chord 63
Figure 3. 8 Interconnexion entre les nœuds dans Power_Chord (N=200) 64
Figure 3. 9 Distribution des degrés entrants dans e-Chord 66
Figure 3.10 Résultats Power Chord 67
Figure 3. 11 Résultats de Power_Pastry 69
Figure 3. 12 Fréquence d'utilisation des tables dans Power Pastry 70
Figure 3. 13 Résultats de Power_Kad 71
Figure 3. 14 Fréquence d'utilisation des tables dans Power_Kad 72
Figure 3. 15 Comparaison des Power_DHTs 73
Figure 4. 1 Effet de la variation du degré de réplication 90
Figure 4. 2 Effet du churn 92
Figure 4. 3 Algorithme Sym_Flood 96
Figure 4. 3 Résultats des émulations des différentes méthodes d'inondation 98

LISTE DES TABLEAUX

Tableau 2. 1 Dénomination des différents nœuds 25
Tableau 2. 2 Performances des différentes DHTs 28
Tableau 2. 3 Effet de la mobilité MANET sur un réseau P2P 36
Tableau 2. 4 Comparaison des approches P2P MANETs 38
Tableau 4. 1 Comparaison des méthodes de réplication dans les systèmes structurés 86

INTRODUCTION GENERALE

Contexte

Depuis les débuts d'Internet, le modèle client-serveur était le modèle de référence pour la mise à disposition des ressources. Dans ce modèle, le système repose sur un serveur dédié qui centralise et maintient l'ensemble des ressources et des services. Dès lors, l'augmentation du nombre d'utilisateurs exige un plus grand investissement des fournisseurs de services. Il est en effet nécessaire de garantir la disponibilité des ressources et des services, malgré le grand nombre de demandes simultanées. Ceci nécessite d'importantes ressources et impose des contraintes logicielles et matérielles, qui rendent de tels systèmes très coûteux. Pourtant, l'augmentation du nombre de participants implique aussi que cet ensemble possède une forte ressource cumulée et une multitude de services ; d'où le paradigme Pair à Pair.

Le principe du pair-à-pair consiste en la mise en relation d'utilisateurs afin de mutualiser les ressources et distribuer les tâches (1). Un système pair-à-pair est ainsi un système distribué de pairs connectés, consommateurs et fournisseurs de service. Les systèmes reposant sur ce paradigme sont mis en œuvre sous forme de réseaux logiques, connectant les participants au dessus des réseaux physiques.

Les systèmes pair-à-pair firent leur apparition à la fin des années 90, et ont été depuis en développement continuel. Pourtant, le concept du pair-à-pair est loin d'être récent. Bien au contraire, il est à l'origine même d'Internet. Dès 1969, ARPANET, l'ancêtre d'Internet, fonctionnait déjà suivant le modèle pair à pair. Ce réseau, composé d'universités et d'entreprises, était principalement utilisé pour partager les données entre ces différents sites. Les premières applications et protocoles de communication conçus pour l'échange des fichiers, étaient déjà des applications pair à pair, dans le sens où chaque pair pouvait être consommateur et fournisseur à la fois. Le succès d'ARPANET a contribué à l'expansion du projet, et donna naissance au protocole TCP/IP, qui sera la base d'Internet. Les nouvelles applications basées sur ce nouveau modèle de communication ont transformé successivement le réseau Internet en un système essentiellement client-serveur. Le réseau est ainsi passé vers un système orienté vers la consommation.

C'est le logiciel Napster qui a considérablement popularisé le concept du pair à pair. Le succès et l'agitation suscitée par l'affaire Napster furent alors à l'origine du boom du *P2P file sharing* (49% à 83% du trafic Internet (2)). Plusieurs logiciels de partage de fichiers se sont succédés, nous citons : Gnutella, eMule, KaZaA, BiTtorrent…

Si les premières applications de ces systèmes étaient exclusivement liées à l'échange et le partage de fichiers, les systèmes pair-à-pair sont utilisés depuis quelques années pour des applications variées telles que le stockage, la distribution de contenu, la communication, ou encore le calcul distribué. Le modèle P2P ouvre de nouveaux horizons aux applications déployées. Il permet en effet de décentraliser la réalisation des services et de mettre à disposition des ressources partagées dans un réseau. Cette décentralisation présente de nombreux avantages qui repoussent les limites du modèle client-serveur, tels que la tolérance aux pannes, le passage à l'échelle et la minimisation des coûts. En effet, l'absence d'élément central permet de mieux pallier aux pannes. La répartition équitable des données et des tâches permet d'équilibrer le trafic sous jacent. Enfin, la mutualisation de toutes les ressources permet de réduire les coûts liés à l'achat et à la maintenance des équipements.

Toutefois, ces systèmes pair-à-pair décentralisés ne peuvent pas faire appel à une entité centrale pour coordonner l'interconnexion des pairs et s'organiser selon une topologie dynamique permettant d'assurer un routage efficace. C'est pourquoi sont apparus des systèmes qui imposent une structure entre les pairs afin de garantir un diamètre optimal. Parmi ces systèmes, les structures basées sur le principe des tables de hachages distribuées (DHTs). Le principe est d'organiser les pairs selon un réseau logique structuré par exemple un anneau ou un hypercube, afin de pouvoir employer des techniques de routage efficaces. Ces structures disposent d'une administration transparente, mais sont toutefois complexes à concevoir, et posent divers problèmes liés à la gestion du réseau logique ainsi crée.

Par ailleurs, l'organisation des pairs de la structure doit tenir compte de l'hétérogénéité des participants, afin de permettre un routage plus efficace. Certaines solutions proposent des structures hiérarchiques qui reposent sur un réseau logique basé sur les super nœuds. Cela permet certes une meilleure utilisation des ressources disponibles, seulement la sélection et la maintenance des super nœuds s'avèrent délicates.

Contributions

Dans cette thèse, nous nous intéressons à l'amélioration des structures de routage dans les systèmes P2P de partage de ressources, basés sur les DHTs. Nous nous concentrons principalement sur deux aspects de cet environnement : l'organisation des nœuds dans le réseau logique et l'organisation des données dans ce dernier.

Concernant l'organisation des nœuds dans le réseau logique, nous proposons Power_DHT, une nouvelle structure d'interconnexion et de routage. La motivation pour nos travaux part de l'observation d'une dichotomie dans le graphe d'interconnexion de la DHT. Alors que les algorithmes DHTs existants sont purement égalitaires, nous constatons une forte hétérogénéité dans la connectivité des pairs dans les réseaux déployés. Notre idée est d'exploiter cette hétérogénéité. Ainsi, nous transformons dynamiquement la DHT vers une structure décentralisée, non égalitaire, exhibant les propriétés d'un graphe sans échelle (distribution des degrés, faible diamètre). Nous exploitons les propriétés de cette nouvelle structure et implémentons à la fois, le routage KBR offrant un plus petit diamètre à un coût de signalisation moindre, et le support de la diffusion efficace réalisant ainsi des recherches floues. Nous choisissons pour notre évaluation d'appliquer Power_DHT à trois géométries différentes : l'anneau de Chord, l'arbre de Kademlia et la structure hybride de Pastry. Les résultats de Power_DHT sont publiés dans (3) (4) (5) (6) (7).

Quant à l'organisation des données dans l'espace virtuel, nous employons la réplication pour améliorer la disponibilité et l'accessibilité des objets. Nous avons implémenté et évalué différentes méthodes de réplication employées dans les DHTs. Les résultats sont publiés dans (8). A partir de ces résultats, nous avons conçu un mécanisme d'inondation efficace pour les systèmes P2P structurés. Ce mécanisme, évalué sur notre plate-forme, exploite la méthode de réplication symétrique pour permettre les recherches floues dans les DHTs, tout en limitant le coût de signalisation de la recherche par inondation (9) (10).

Organisation du document

Ce document se ~~présente~~ présente en trois parties. La première présente un état de l'art des systèmes P2P et la seconde aborde nos deux principaux axes de recherches à savoir l'organisation des nœuds dans le réseau logique dynamique et l'organisation des données dans ce dernier.

La première partie, composée des deux premiers chapitres, présente le principe du paradigme pair à pair. Nous commençons par définir le model pair à pair, ses domaines d'application et ses propriétés, puis nous présentons une classification des grandes catégories des systèmes existants. Nous détaillons en particulier dans le deuxième chapitre la catégorie des systèmes P2P structurés, basés sur les tables de hachage distribuées (DHT). Plus spécifiquement, nous présentons le principe général des DHTs, puis nous identifions les différentes géométries de routage existantes. Nous nous attarderons en particulier sur Chord, Pastry et Kademlia, les trois structures DHTs sur lesquelles nous avons travaillé. Enfin, une dernière partie expose les différents travaux d'optimisation autour des DHTs dont la hiérarchisation, la réplication et les méthodes de recherche aléatoires.

La seconde partie du document détaille les contributions de cette thèse. Concernant l'organisation des nœuds dans le réseau logique, nous détaillons dans le troisième chapitre le travail qui a mené à notre contribution principale Power_DHT. Nous proposons dans un premier temps Reverse_DHT, une nouvelle structure DHT bidirectionnelle intermédiaire. A partir des résultats de Reverse_DHT, nous proposons notre structure finale Power_DHT. Nous présentons dans cette partie l'architecture et les fonctionnalités de cette nouvelle structure. L'évaluation de Power_DHT montre que notre structure Power_DHT permet d'assurer un routage plus efficace, tout en réduisant le diamètre des recherches et le coût de signalisation.

Dans la suite de notre travail, nous avons exploré quelques pistes auxiliaires, pour l'optimisation des performances de notre structure. Ainsi, nous proposons dans le quatrième chapitre une nouvelle méthode d'inondation dans les DHTs nommée Sym_Flood. Notre approche Sym_Flood exploite les propriétés avantageuses de la réplication symétrique pour limiter de coût de l'inondation et le diamètre des recherches. Pour cela, nous détaillons d'abord les différentes techniques de réplication employées dans les DHTs. Nous présentons une analyse comparative de leurs propriétés et de leurs comportements afin d'en relever les différences. Nous implémentons et évaluons par émulation ces différentes méthodes de réplication. A partir des résultats, nous choisissons la réplication symétrique. Par la suite, nous détaillons notre approche Sym_Flood et nous évaluons notre proposition. Nos résultats montrent que notre approche Sym_Flood permet de réduire le coût des recherches par inondation tout en assurant un routage robuste et un faible diamètre.

Pour terminer, nous concluons par une synthèse des principales contributions de ce travail et listons une série de perspectives.

CHAPITRE 1.

VUE D'ENSEMBLE DES SYSTEMES PAIR A PAIR

Le paradigme pair-à-pair (de l'anglais *peer to peer* : P2P) a été centré dès le départ sur la gestion de larges quantités de données réparties à très grande échelle. Même s'il a été utilisé jusqu'à présent essentiellement à des fins de partage de fichiers, de nombreux projets de recherche s'y intéressent aujourd'hui pour d'autres types d'applications dont la collaboration, le calcul distribué, le partage et la distribution du contenu. Les applications P2P collaboratives proposent à des communautés d'usagers des services de communication et d'échange de données. Différents moyens de communications sont souvent offerts, avec notamment la messagerie instantanée, la voix et la vidéo (ICQ (11), Skype (12), PPLive (13)). Le calcul distribué permet la distribution, par un serveur central, d'un calcul sur un ensemble de participants. Un des projets les plus célèbres est SETI@home (14). Toutefois, le partage et la distribution de contenu reste aujourd'hui l'utilisation principale des systèmes pair-à-pair (15). La distribution de données P2P couvre un large spectre, partant des systèmes de partage de fichier simplistes (Bittorent (16)) jusqu'à des systèmes élaborés construisant une infrastructure de stockage distribuée permettant la publication, l'organisation, et la recherche des données (Oceanstore (17), PAST (18)).

Dans cette thèse, nous nous intéressons aux applications de partage et de distribution du contenu dans un réseau pair à pair. Dans ce premier chapitre, nous introduisons brièvement les systèmes pair-à-pair. Nous détaillons d'abord la terminologie employée dans ce rapport puis nous présentons un ensemble de définitions et caractéristiques spécifiques au domaine pair à pair. Nous présentons ensuite une classification des systèmes P2P selon les architectures proposées.

1.1 Terminologie

Les notations utilisées dans ce rapport sont détaillées dans la terminologie suivante.

Dans un système P2P, chaque pair crée des connexions vers un ensemble de pairs, en utilisant les services de télécommunications disponibles localement. Lorsqu'on cherche à insister, dans un système P2P, sur cet aspect d'interconnexion des pairs, on parle d'un *réseau* pair-à-pair. Le réseau P2P est ainsi le maillage établi par les pairs, en utilisant les protocoles du système P2P.

En plus d'employer les services existants dans le réseau des télécommunications, le réseau P2P possède ses propres mécanismes pour le nommage, l'adressage, le routage, etc. Quand ces mécanismes sont distincts du réseau des télécommunications utilisé, on parle d'un réseau *logique en superposition* (en anglais : *overlay*), pour insister sur le fait que le réseau P2P devient alors un réseau distinct. Ainsi, un lien dans un réseau P2P désigne une connexion entre deux pairs et peut passer par plusieurs nœuds du réseau des télécommunications *en dessous*, appelé par analogie réseau *physique* (en anglais : *underlay*).

Un réseau logique est donc l'interconnexion qui relie virtuellement les participants d'un système pair à pair, au dessus d'un réseau physique.

Un lien entre deux pairs du réseau logique est une connexion virtuelle permettant la communication entre les deux nœuds. Ce lien est souvent appelé *virtuel* ou *logique*. Analogiquement, les liens dans le réseau de télécommunications sont appelés *réels* ou *physiques*.

Le lien sortant est le lien direct qui mène vers un pair voisin du réseau logique. L'ensemble des liens sortants définit le nombre de voisins d'un nœud, et donc son *degré sortant*. Le lien entrant est le lien direct depuis un nœud du réseau logique. L'ensemble des liens entrants d'un nœud A définit le nombre de nœuds dont A est voisin, et donc son *degré entrant*. Enfin, le degré d'un nœud est la somme de son degré entrant et de son degré sortant (19).

L'objet est la donnée partagée dans le système pair à pair. Si le système intègre un mécanisme de réplication, plusieurs copies d'un même objet peuvent exister. Le terme *réplica* désigne donc la copie d'un objet. On nommera nœud *racine* le nœud qui détient la copie *zéro* de l'objet, et nœud *répliquant* le nœud qui détient la copie i. Le degré de réplication $\boldsymbol{r}$ désigne dans ce cas le nombre de réplicas.

Lorsqu'un nœud A envoie un message de recherche pour un objet donné, ce message sera nommé *requête.* Le nœud source désigne le nœud émetteur du message et le nœud destination la destination finale du message.

Le taux *churn* (contraction de l'anglais *change and turn*) désigne en télécommunications le rapport entre le nombre d'arrivées et le nombre de départs des nœuds sur une période donnée. Par abus de langage, nous utilisons le terme *churn* pour dénoter le dynamisme des nœuds.

1.2 Définition

Un système pair-à-pair est un système distribué dans lequel des nœuds égaux (en termes de rôle et usage), échangent directement des informations et des services (20).

Derrière cette définition académique se trouve la réalité du cadre d'utilisation de ces systèmes. En effet, c'est le rôle des pairs de les mettre en place. Il n'y a alors aucune entité centrale qui contrôle le système. Cela entraîne un certain nombre d'avantages, mais aussi des contraintes.

Pour les avantages, le fait d'augmenter le nombre des participants augmente aussi les ressources à disposition tout en répartissant la charge entre les particiants. Il est donc possible de passer à l'échelle à un coût relativement faible. De plus, l'absence d'entité de contrôle centrale confère au système une meilleure robustesse, dans la mesure où la disponibilité des ressources n'est pas directement liée à un pair en particulier (21).

Pour les contraintes, l'absence d'élément central nécessite la mise en place d'un mécanisme d'auto-organisation. Les pairs doivent se connecter ou se déconnecter au réseau sans que cela n'affecte la disponibilité des ressources. De plus, les systèmes pair à pair doivent gérer un réseau de noeuds fortement hétérogènes (architecture matérielle : processeur, mémoire et type de connexion : modem, haut débit…).

Par ailleurs, la localisation des objets dans un système P2P devient difficile lorsqu'on ne peut pas faire appel à un serveur central. De ce fait, la technique de recherche à employer est directement liée à l'organisation des pairs dans le réseau logique. Ces pairs peuvent s'organiser selon trois grandes classes que nous définissons dans la partie qui suit.

1.3 Différents niveaux de centralisation

Les systèmes P2P peuvent être classés en trois grandes familles. Nous présentons dans cette partie ces trois familles selon le niveau de centralisation qui se traduit par une distribution plus ou moins importante des tâches à accomplir. Nous décrivons ainsi le modèle P2P centralisé,

distribué et décentralisé (hybride). Pour chacun d'eux, nous présentons un exemple type des systèmes existants.

1.3.1 Systèmes P2P centralisés

Le modèle centralisé est à la limite du modèle P2P, car il repose sur un serveur dédié qui centralise et maintient l'ensemble des connaissances des pairs, les ressources étant toujours hébergées sur les pairs. Dans ce modèle, le serveur est une entité de localisation et ne contient aucune ressource. Seule la récupération d'objets est décentralisée.

L'intérêt de ce modèle est de permettre une recherche exhaustive, une localisation rapide des ressources et une gestion simple grâce à l'entité centrale. L'inconvénient majeur est que le fonctionnement du système repose uniquement sur cette entité. Cette dernière doit être capable de supporter un grand nombre de requêtes et d'effectuer beaucoup de recherches.

Le système le plus connu qui rentre dans la catégorie des systèmes pair-à-pair centralisés est le système de partage de fichiers MP3 Napster (22). Napster repose sur 150 serveurs index et un méta serveur. Pour rendre ses fichiers accessibles aux autres utilisateurs, un pair doit se connecter à un des serveurs. Une fois connecté, le pair envoie la liste des fichiers qu'il partage. Pour récupérer un fichier, un pair demande au serveur auquel il est connecté la liste des pairs possédant le fichier, ensuite il choisit un des pairs dans la liste. L'échange des fichiers se fait ensuite directement entre les pairs. Même si une grande partie du protocole repose sur une architecture client/serveur, il s'agit d'un système pair-à-pair puisque le transfert de fichier se fait indépendamment de l'index central.

1.3.2 Systèmes P2P distribués

Les systèmes P2P distribués essayent de répartir la totalité des fonctions du système entre les pairs à savoir : la recherche, le routage et la récupération des objets dans le réseau logique. Les pairs doivent s'organiser pour former une architecture dynamique efficace les connectant entre eux. Il existe deux approches pour cela : construire un graphe aléatoire et reposer sur des explorations/recherches aléatoires du réseau, ou bien construire un graphe structuré dans lequel un routage efficace de proche en proche est supporté (23).

1.3.2.1 Graphes aléatoires

Dans cette approche, le réseau logique se construit selon un processus d'exploration aléatoire : un nouvel arrivant parcourt le réseau et choisit aléatoirement ses pairs voisins. La recherche aléatoire dans ces graphes ne tient pas compte des caractéristiques du réseau et s'effectue le

plus souvent par inondation ou marche aléatoire (de l'anglais *random walk*) (24). Ces systèmes P2P sont appelés systèmes P2P non structurés. Dans la suite du document, l'expression recherche/exploration aléatoire désigne l'inondation ou la marche aléatoire.

Ces méthodes d'explorations aléatoires permettent les recherches complexes tel que les recherches par suffixe (bitto*) pour les mots commençant par (bitto) ou les recherches par groupe de mots, par exemple (bittorent, DHT).

Pour assurer le routage dans un graphe dont on ignore la topologie, la recherche par inondation propose de retransmettre récursivement la requête de recherche à tous les voisins d'un pair (sauf celui dont il a reçu la requête) jusqu'à la localisation de l'objet ou l'expiration du nombre de sauts TTL. Cette garantie d'acheminement est assurée même si la topologie change à la suite, par exemple, d'une défaillance de certains pairs. Le prix à payer pour ces qualités de simplicité, de robustesse et de rapidité d'acheminement est une mauvaise utilisation des ressources du réseau et un coût de signalisation important (24). Un grand nombre de travaux ont tenté d'optimiser le coût de l'inondation en proposant des méthodes de recherches basées sur la marche aléatoire, l'inondation par TTL croissant (25) ou les filtre de bloom (26) .

Gnutella (27) (28) dans sa première version v0.4, propose de connecter les nœuds de manière aléatoire et emploie l'inondation pour ses recherches dans le réseau logique ainsi formé. Pour une meilleure utilisation des ressources, la version suivante, Gnutella v0.6 (29), propose d'organiser le réseau logique selon une architecture décentralisée sur deux niveaux. Les nœuds de faible capacité se rattachent à un super nœud et les super nœuds entre eux sont reliés dans un réseau Gnutella V0.4. Un nœud peut être nommé super nœud suivant plusieurs critères : cela peut dépendre de son adresse (publique ou privée), de son système d'exploitation, de sa bande passante, de l'instant depuis lequel il est connecté au réseau, ou de ses ressources matérielles (puissance de calcul, capacité mémoire) (30).

1.3.2.2 Graphes structurés

Les systèmes structurés proposent d'utiliser la théorie des graphes pour imposer une organisation spécifique des nœuds et des interconnexions.

Ces systèmes attribuent à chaque nœud et chaque objet un identifiant/clé unique dans un même espace de nommage (ID_{objet}, $ID_{nœud}$ respectivement). Ensuite ils distribuent l'index (ID_{objet}, objet) en donnant la responsabilité de chaque objet à un identifiant de nœud $ID_{nœud}$ du réseau. La technique consiste à stocker l'association (ID_{objet}, objet) sur le nœud

d'identifiant le plus proche selon une métrique prédéfinie. Cette distribution nécessite alors que chaque nœud du réseau puisse trouver un objet à partir de son identifiant, et donc le nœud qui en est responsable.

Skip list et Skip Graph

Une solution pour trouver un objet à partir d'une clé consiste à employer les *Skip Graphs* et les *Skip Lists*. Une *skip list* ou liste à enjambements est une structure de données probabiliste, à base de listes chaînées parallèles (31) (32). Une *skip list* se présente comme une amélioration d'une liste chaînée triée. Elle contient des pointeurs supplémentaires vers l'avant, ajoutés de façon aléatoire, de sorte que la recherche dans la liste puisse sauter de nombreux éléments. La *skip list* est organisée en couches. La couche la plus basse de niveau 0 est simplement une liste chaînée standard. Chaque couche supérieure est une voie rapide pour parcourir les couches inférieures. En moyenne, il y a $\mathrm{O}(\log \mathrm{N})$ niveaux. La recherche commence toujours par le plus petit élément sur la couche la plus haute, qui fait de lui un point de saturation. Pour chaque couche visitée, on parcourt les chaînons jusqu'à atteindre le dernier élément inférieur ou égal à l'élément recherché.

Un *skip graph* est composé de plusieurs listes à chaque niveau pour ajouter de la redondance et éviter la saturation du point du plus haut niveau de la liste. Le niveau 0 contient toutes les clés. Chaque nœud du *skip graph* participe dans chaque niveau mais dans une liste différente. Une liste de niveau i comporte les nœuds de la liste de niveau 0 partageant un même préfixe de longueur i. Chaque nœud dans un *skip graph* maintient $\mathrm{O}(\log \mathrm{N})$ voisins en moyenne et route en $\mathrm{O}(\log \mathrm{N})$ sauts.

Tables de hachage distribuées

Un domaine très actif ces dernières années concerne les tables de hachage distribuées (DHTs). Ces systèmes se basent sur des graphes orientés. Ils utilisent une table de hachage commune à tous les nœuds pour associer à chaque objet et chaque nœud un identifiant, ou clé dans un même espace de nommage. Lorsqu'un nœud recherche un objet, il envoie une requête de recherche vers $\mathrm{ID}_{\mathrm{objet}}$ stocké dans le nœud $\mathrm{ID}_{\mathrm{nœud}}$. L'envoi d'un message vers une clé permet donc d'atteindre le nœud responsable de cette clé et donc de l'objet. Lorsqu'un nœud envoie un message vers une clé, ce message est transmis de nœud en nœud selon l'algorithme de routage de la DHT. Parmi les différentes DHTs, nous citons Chord (33), Pastry (34),

Kademlia (35), CAN (36),… Nous détaillons les différentes propositions DHTs dans le deuxième chapitre.

1.3.3 Systèmes P2P décentralisés

Le modèle décentralisé, appelé aussi hybride, ajoute un degré de hiérarchie au modèle centralisé. Ce modèle repose sur des interconnexions de super nœuds sur le niveau haut de la hiérarchie, suivant le modèle distribué. Chaque nœud feuille se rattache à un ou plusieurs super nœuds. Un super nœud gère un ensemble de nœuds feuilles. Le protocole FastTrack, par exemple, propose 1 super nœud pour 100 feuilles. Dans Gnutella2 (37), chaque super nœud est connecté au plus à 10 autres super nœuds, et gère entre 10 et 100 nœuds feuilles.

Les objets partagés par une feuille sont enregistrés sur le super nœud responsable de cette feuille. Lorsqu'une feuille recherche un objet, elle envoie sa requête à son super nœud. Celui-ci effectue alors la recherche parmi les objets des nœuds qui lui sont rattachés, voire les super nœuds voisins si besoin.

Le rôle des super nœuds varie d'une application à une autre. Cependant, ils sont utilisés en général pour assurer les fonctions relatives à la localisation, au routage ou à l'organisation des pairs. Skype (27) (38), par exemple, les utilise pour la découverte et la localisation des pairs.

Cette architecture décentralisée permet de diminuer le nombre de messages reçus par une grande partie des nœuds (les feuilles) en augmentant la charge des super nœuds. Cela permet aussi d'augmenter l'efficacité de la recherche sur les super nœuds puisqu'ils sont responsables de plus d'objets. Toutefois, le choix optimal des super nœuds n'est pas trivial, plusieurs critères sont à définir selon les besoins de l'application.

1.4 Conclusion

Dans ce premier chapitre, nous avons présenté une vue d'ensemble sur les systèmes pair-à-pair de distribution et localisation de contenu. Différents niveaux de décentralisation permettent de classifier ces systèmes en trois catégories qui sont : les systèmes P2P centralisé, distribué et décentralisé. Certaines classifications P2P considèrent les systèmes P2P décentralisé et centralisé comme identiques, le système centralisé étant un cas particulier du système décentralisé contenant un seul super nœud. Dans ce document, nous préférons conserver la distinction entre les deux systèmes, car le système décentralisé peut être vu comme la combinaison des systèmes d'indexation distribué et centralisé, dans le sens où, chaque super

nœud agit comme une entité centrale pour ses pairs feuilles alors que les super nœuds sont organisés suivant un système distribué.

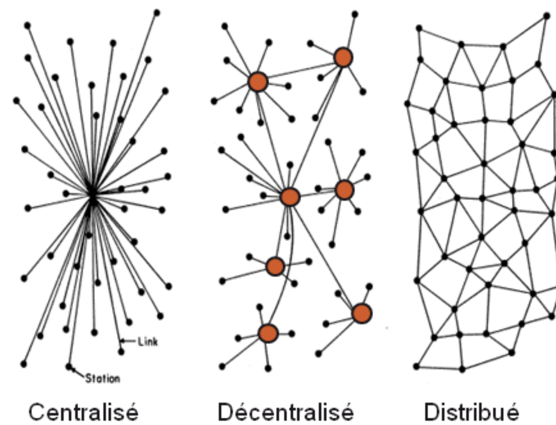


Figure 1. 1 Classification des systèmes P2P

Parmi les systèmes distribués, nous avons évoqué les tables de hachage distribuées DHTs. Comme nous allons voir, ces DHTs proposent des topologies différentes comme l'anneau de Chord, l'hypercube de CAN ou l'arbre de Kademlia… Dans la suite de ce document, nous proposons d'étudier plus en détail ces systèmes. Nous allons d'abord présenter les propositions DHTs existantes puis nous résumerons les différents travaux d'optimisation réalisés.

CHAPITRE 2.

LES TABLES DE HACHAGE DISTRIBUEES

Dans ce chapitre, nous présentons les tables de hachage distribuées qui sont les principales structures de routage distribuées dans les systèmes pair à pair. Nous commençons par exposer le principe général de fonctionnement des DHTs ainsi que leurs principales caractéristiques. Au niveau des propositions, il existe plusieurs grandes classes de DHTs qui se différencient principalement par le modèle topologique sur lequel elles reposent. Au sein de chaque classe, il existe de nombreuses propositions. Dans ce chapitre, nous allons présenter une DHT majeure pour chaque classe. Nous nous intéresserons principalement à Chord, Pastry et Kademlia, les trois DHTs sur lesquelles nous avons travaillé. Enfin, dans une dernière partie, nous passons en revue les différents travaux qui proposent des optimisations et des extensions à ces structures DHTs.

2.1 Les tables de hachage distribuées

2.1.1 Principe général

Une table de hachage est une structure de données qui associe par hachage une clé à un objet afin de construire un index. Une clé est le résultat d'une fonction de hachage appliquée à un élément de l'objet. Par exemple, la clé d'un pair est le résultat de l'application d'une fonction de hachage sur l'adresse IP de son nœud, alors que la clé d'un objet peut être le hachage de son nom.

La particularité des tables de hachage distribuées est la nouvelle technique de distribution et de localisation qu'elle propose. Dans une DHT, les pairs et les objets sont identifiés dans un même espace de nommage. Chaque nœud est responsable d'une partie des clés dans le

système et chaque objet est stocké dans le nœud dont l'identifiant est le plus proche de sa clé selon la métrique de distance utilisée.

Lorsqu'un nœud met à disposition un objet, il doit d'abord l'annoncer/publier au système. La publication consiste à calculer la clé associée à l'objet, puis à envoyer un message au nœud responsable de la clé à travers le réseau logique. Lorsqu'un nœud cherche un objet associé à une clé, il cherche directement la clé correspondante. Quand le nœud responsable est trouvé, ce dernier répond au nœud source et envoie les éventuelles informations relatives à l'objet. La recherche basé sur la clé dans les DHTs est appelée recherche exacte ou recherche basée sur la clé (en anglais : *Key Based Routing* ou *KBR)*

Pour assurer le routage, chaque nœud dans la DHT maintient régulièrement les adresses de quelques nœuds voisins, choisis selon la structure. Lors de l'envoi d'un message dans le réseau, cette structure permet de se rapprocher du destinataire à chaque saut. Le routage dans une DHT permet à chaque nœud recevant un message de décider localement à quel voisin faire suivre le message.

2.1.2 Propriétés

Les structures DHTs s'inspirent des graphes (par exemple les hypercubes, les arbres, etc.). Ces graphes proposent des propriétés intéressantes dont :

Faible diamètre de routage

Dans la plupart des DHTs, le nombre de sauts nécessaires pour le routage d'une requête s'exprime en $O(\log N)$, N étant la taille du réseau. Dans un réseau à 2^{10} nœuds par exemple, la DHT produit un diamètre de 10 sauts (si l'on considère une base binaire).

Passage à l'échelle

Le faible diamètre de la DHT et la taille constante ou logarithmique des tables de routage permettent aux DHTs une bonne propriété de passage à l'échelle.

La tolérance aux pannes

L'absence de centralisation confère aux structures DHTs une meilleure robustesse face aux pannes. Les requêtes de recherches peuvent être acheminées par des routes secondaires, même en cas de *churn*. Nous allons voir dans la suite que certaines DHTs sont plus vulnérables que d'autres face au *churn*. Des mécanismes de redondance sont souvent mis en place pour éviter la perte définitive ou l'inaccessibilité des ressources.

L’équilibre de charge

L’utilisation de fonctions de hachage pour créer les identifiants des pairs et les clés des objets suppose que les données soient équitablement partagées entre les nœuds. Cet équilibre induit l’équilibre du trafic, dans le cas où chaque ressource est sollicitée de manière équivalente. Nous allons voir dans la suite que cette propriété n’est pas souvent vérifiée.

2.2 DHTs existantes

Nous utilisons la terminologie suivante dans notre description des DHTs :

Les voisins d’un nœud A sont les nœuds maintenus dans sa table de routage et calculés suivant la métrique de distance utilisée. Les voisins séquentiels sont les nœuds maintenus par A et dont les identifiants IDs sont les plus proches du nœud A (suivant la métrique de la DHT).

Les identifiants des nœuds sont codés par 2^b chiffres, b est un paramètre de configuration fixé à 4, par exemple, pour un alphabet hexadécimal et à 1 pour un alphabet binaire.

2.2.1 Topologie en anneau

Plusieurs structures DHTs utilisent la structure en anneau d’une manière plus ou moins directe. Le système DKS (39) par exemple repose sur une topologie en anneau. Pastry, bien que basé principalement sur une topologie en arbre, emploie aussi une structure en anneau pour finaliser le routage. La topologie en hypercube torique de CAN peut être assimilée à une topologie annulaire multi-dimensionnelle. Toutefois, la DHT basé essentiellement sur une topologie en anneau est celle de Chord.

Chord

Chord est déployée dans plusieurs applications. La DHT est utilisée dans le système de stockage de fichiers distribués CFS (40), DDNS (41) la version pair-à-pair du DNS, P2P/SIP (42) et dans le système de localisation de ressource RELOAD (43)

La topologie de Chord (33) est représentée sous forme d'un anneau. Les pairs sont placés dans l’anneau dans l’ordre croissant de leurs identifiants ID. Le nœud dont l'identifiant est immédiatement supérieur (respectivement inférieur) à ID sera nommé successeur succ (ID)(respectivement prédécesseur pred (ID)*)*. Dans ce système, chaque clé est associée au nœud d'identifiant immédiatement supérieur. Ainsi, chaque pair ID est responsable de l’intervalle des clés]pred(ID), ID].

Connexion entre les nœuds : Chaque pair dans Chord connait son prédécesseur et la liste de ses successeurs qui constituent l'ensemble de ses voisins séquentiels. Pour un pair donné, la simple connaissance du prédécesseur et des successeurs n'est pas suffisante pour garantir une bonne performance dans l'anneau, notamment en termes de nombre de sauts nécessaire pour l'acheminement des requêtes. Pour cela, en plus de maintenir les $\mathrm{succ(ID)}$ et le $\mathrm{pred(ID)}$, chaque pair ID se connecte vers d'autres nœuds voisins, appelés *fingers*, qui constituent sa table de routage. Dans l'espace $[0, 2^l[$, le $n^{\text{ième}}$ finger est $\mathrm{succ(ID}+2^{\mathrm{i}})$, i dans $[0, l]$ (Figure 2. 1). Ainsi, le nombre de *fingers* par nœud est $\mathrm{O(log\ N)}$.

Routage : Lorsqu'un nœud souhaite trouver une clé, il utilise l'algorithme suivant : il cherche parmi ses *fingers* le nœud dont l'identifiant est le plus grand tout en étant inférieur à la clé, et lui transmet le message. Le nœud qui reçoit le message exécute alors à son tour cet algorithme. La recherche divise ainsi la distance logique séparant le nœud courant du nœud destination par un facteur au moins 2 à chaque étape, assurant de trouver la destination en un nombre de sauts de $\mathrm{O(log\,N)}$.

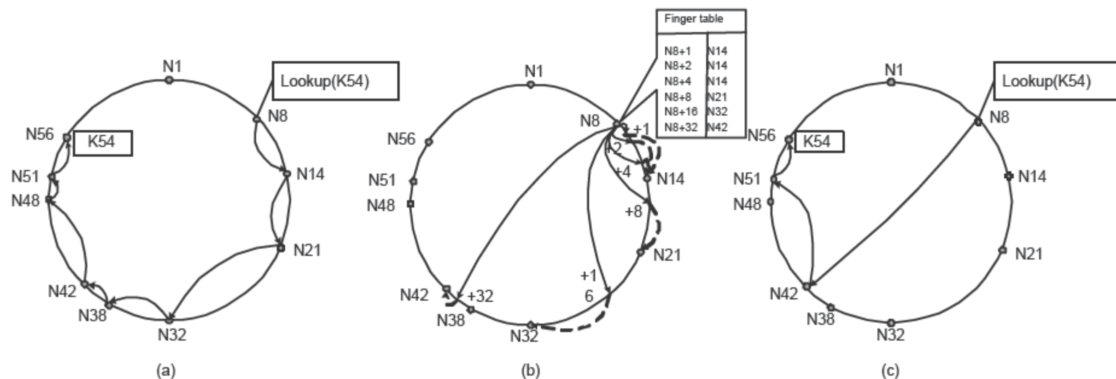


Figure 2. 1 Extrait de (44) (a) Acheminement d'une requête dans un anneau simple (b) Calcul des fingers (c) Acheminement d'une requête par les fingers

Maintenance : le processus de maintenance s'exécute périodiquement pour maintenir la structure c'est à dire les liens vers les voisins séquentiels et les voisins *fingers* présents dans la table de routage. La vérification des *fingers* est similaire à la construction de la table et consiste à rechercher pour chaque entrée le nœud d'identifiant immédiatement supérieur à $(\mathrm{ID}+2^{\mathrm{i}-1})$, i dans $[0, l]$ moyennant les messages *fix_finger*. La table est mise à jour si une nouvelle entrée c'est à dire un nouveau *finger* est trouvé. De plus, afin de limiter les échecs de requêtes,

chaque nœud maintient la liste de ses successeurs, qui peuvent être utilisés alternativement pour le routage.

2.2.2 Topologie en arbre basée sur l'algorithme de Plaxton

Une seconde famille de DHTs repose sur l'algorithme de Plaxton (45). Dans cette famille, on classe Pastry (34), Tapestry (46), Bamboo (47)… Nous détaillons dans ce qui suit la DHT de Pastry que nous avons considérée dans nos travaux.

Routage par préfixe dans PASTRY

La DHT de Pastry utilise une approche de routage hybride : un routage en préfixe selon l'algorithme de Plaxton, puis lors des dernières étapes, un routage selon la topologie en anneau similaire à Chord. Le protocole Pastry est la base du système de stockage de donnée anonyme PAST (18), et de l'application de diffusion d'évènement Scribe (48).

Connexion entre les nœuds : Dans Pastry, lorsqu'un message est routé vers un destinataire, le préfixe commun entre les nœuds intermédiaires et la destination augmente à chaque saut. Dans Pastry, chaque nœud A maintient une table de routage comme suit : chaque ligne i contient 2^{b-1} entrées dont les i premiers chiffres sont communs à ceux de A. Ainsi, une table de routage Pastry contient $O\left(\left(2^{b-1}\right) * \log_{2^b} N\right)$ voisins. En plus de la table de routage, Pastry maintient un ensemble de voisins virtuels (séquentiels) nommé *leaf* (feuille), qui contient $L/2$ successeurs et $L/2$ prédécesseurs $(L = 8)$ (Figure 2. 2). Enfin, la dernière table de Pastry, maintient un ensemble des voisins réels (physique). La métrique choisie pour évaluer la distance réelle entre les pairs est le nombre de sauts dans le réseau physique sous jacent.

Pair **10233102**			
Voisins virtuels			
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232
Table de routage			
-0-2212102	**1**	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	**2**	10-3-23302
102-0-0230	102-1-1302	102-2-2302	**3**
1023-0-322	1023-1-000	1023-2-121	**3**
10233-0-01	**1**	10233-2-32	
0		102331-2-0	
		2	
Voisins réels			
13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Figure 2. 2 Extrait de (34) Table de routage du pair 10233102 avec b = 2

Routage : si un nœud cherche à router une clé, il vérifie d'abord si le responsable de cette clé est un de ses nœuds feuilles, c'est à dire dans l'un des intervalles $]\mathrm{leaf}_i, \mathrm{ID}]$ ou $]\mathrm{ID}, \mathrm{leaf}_i]$. Si c'est le cas, le message est envoyé à ce nœud feuille. Sinon, le nœud détermine le nombre k de chiffres communs entre son identifiant et la clé, puis cherche dans sa table de routage le nœud qui partage un plus long préfixe. Enfin, s'il ne trouve pas un tel nœud dans sa table, le nœud cherche parmi ses voisins réels celui dont le préfixe commun avec la clé est au moins égale à k et dont l'identifiant est numériquement plus proche de la clé. Ainsi, dans un réseau de N pairs, Pastry route les messages en $O(\log_{2^b} N)$ sauts.

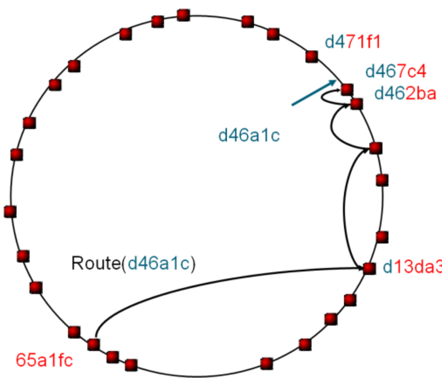


Figure 2. 3 Routage vers la clé (d46a1c)

Maintenance : la procédure de maintenance dans Pastry est similaire à celle de Chord. Périodiquement, chaque nœud vérifie un voisin aléatoire de chaque ligne de sa table de routage, moyennant les messages *row_request*. Il vérifie aussi ses voisins feuilles ainsi que ses voisins réels.

2.2.3 Topologie en hypercube

La DHT la plus connue dans cette catégorie est celle de CAN (36). La DHT de CAN propose un exemple basé sur une topologie de tore à d dimensions. Le routage dans CAN s'effectue de proche en proche : à chaque saut, on ne peut changer de coordonnées que sur une dimension. Toutefois, plusieurs autres structures reposent sur une topologie en hypercube. Chord, par exemple, repose sur un espace cartésien circulaire à une seule dimension modélisé par un anneau. Kademlia (35), bien que modélisé souvent sous la forme d'un arbre binaire, repose aussi sur un hypercube de dimension égale en moyenne à $\log N$. Nous choisissons dans ce travail de détailler la DHT de Kademlia.

Routage en XOR : Kademlia

Kademlia est le premier protocole déployé réellement à une grande échelle, avec le logiciel eMule, via l'implémentation Kad! (49) et plus récemment dans la version Exeem (50) avec Bittorent (16).

Kademlia utilise la métrique *ou exclusif* (XOR) (35) pour calculer la distance séparant, dans le réseau logique deux identifiants de nœuds. L'un des principaux avantages de Kademlia est le nombre réduit de messages de signalisation qu'il envoie. En effet, la mise à jour du voisinage se fait implicitement à travers les messages qu'il intercepte, en stockant si nécessaire les informations relatives au nœud expéditeur selon sa distance XOR. Comme il s'agit d'une métrique symétrique, cela permet aux nœuds Kademlia de ne recevoir des messages que depuis les nœuds présents dans sa table de routage. L'autre avantage de Kademlia concerne les recherches parallèles. Kademlia envoie k requêtes en parallèle pour une même clé pour accélérer les recherches. La recherche converge vers les nœuds dont les IDs sont les plus proches de la clé.

Connexion entre les nœuds : Chaque nœud A dans Kademlia maintient une table de routage composée d'ensembles nommés *buckets*. Les *buckets* sont classés par ordre d'éloignement selon la métrique XOR. Un *bucket* regroupe k nœuds dont les distances sont comprises entre 2^i et 2^{i+1}. Le degré moyen d'un nœud est donc $O\left(k*\log_{2^b}N\right)$. Tous les nœuds du même *bucket* sont à la même distance du nœud A. Chaque *bucket* correspond donc à un sous arbre et contient k voisins classés selon leur ancienneté dans le système. A l'origine, Kademlia ne maintient pas des voisins séquentiels. Cependant, cette information peut être tirée directement de sa table de routage comme proposé dans S/Kademlia (51). En effet, les premiers *buckets* contiennent les identifiants des nœuds les plus proches selon la métrique XOR. Ces nœuds nommés *sibling* constituent l'ensemble des voisins séquentiels dans S/Kademlia (Figure 2. 4).

Figure 2. 4 Extrait de (51) Voisinage du nœud 0011. Les cercles en bleu représentent voisins *sibling* du nœud 0011.

Routage : Lorsqu'un nœud S cherche une clé K, il recherche itérativement les k nœuds les plus proches de K. S commence par identifier et envoyer sa requête à ses k voisins les plus proches parmi ses buckets. Ces voisins vont répondre à leur tour en renvoyant leurs k nœuds les plus proches, présents dans leurs tables de routage. Le nœud S met à jour sa table de routage puis renvoie sa requête aux k nouveaux nœuds les plus proches. La même procédure se répète jusqu'à ce que le nœud S ait contacté tous les autres nœuds renvoyés dans les réponses, et ainsi ait récupéré les k nœuds dont l'ID est le plus proche de la clé K. Cet algorithme a pour but d'augmenter le préfixe commun entre l'identifiant de la clé et celui des nœuds contactés à chaque saut. Le nœud qui effectue une recherche contacte donc un à un tous les nœuds intermédiaires de la route jusqu'à atteindre la destination. Ceux ci lui fournissent en retour une liste de contacts possibles pour le saut suivant (Figure 2. 5). Le nombre de sauts moyen nécessaires pour une recherche de clé sur un alphabet de 2^b chiffres est $O\left(\log_{2^b} N\right)$.

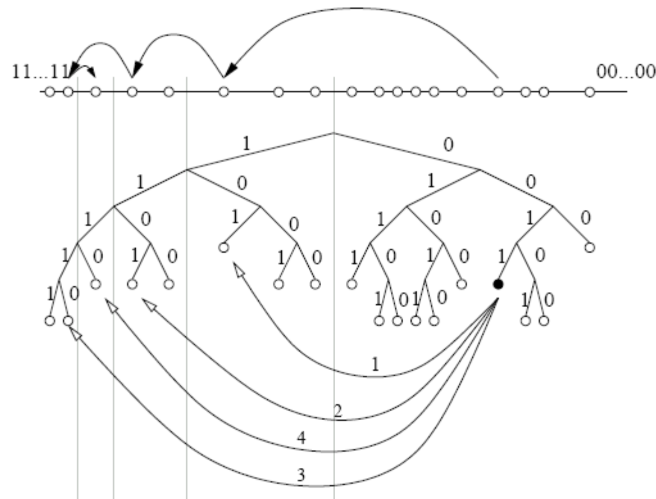


Figure 2. 5 Extrait de (35) Exemple de route Kademlia : le nœud 0011 localise le nœud 1110

Maintenance : la maintenance dans Kademlia est implicite. Lorsqu'un nœud reçoit un message *find_node*, il met à jour le *bucket* approprié. Si le nœud expéditeur se trouve déjà dans ce *bucket*, il est déplacé en bas de la liste. Sinon, si le bucket n'est pas plein, la nouvelle entrée est insérée. Si le *bucket* est plein, le nœud A émet un PING vers le voisin le plus ancien dans ce *bucket.* Si ce dernier répond au PING, il est alors déplacé en fin de liste et la nouvelle entrée est ignorée. Kademlia conserve donc dans sa table les nœuds les plus anciens dans le réseau.

Nous résumons dans le Tableau 2. 1 la terminologie que nous utiliserons dans la suite pour l'appellation des différents nœuds et messages dans Chord, Pastry et Kademlia.

Tableau 2. 1 Dénomination des différents nœuds

	Chord	Pastry	Kademlia
Voisin séquentiel	Successeurs + prédécesseur	feuille	sibling
Nœud de la table de routage	finger	voisin	voisin
Message de maintenance	fix_finger	row_request	find_node

2.2.4 Autres propositions

Topologie en papillon : VICEROY

Viceroy (52) repose sur un graphe en papillon (53). C'est est une proposition de DHT qui s'inspire de Chord tout en y ajoutant de nombreuses améliorations. La principale consiste à construire une topologie à multiples anneaux (multi niveaux) où chaque pair présente une connectivité constante c'est à dire un nombre de voisins fixe (typiquement égale à 7).

Pour N nœuds dans le réseau, Viceroy comporte $\log N$ niveaux de $N/\log N$ nœuds chacun. Un niveau consiste en un anneau bidirectionnel reliant les nœuds d'un même niveau selon l'ordre croissant de leurs identifiants. Le graphe papillon relie les nœuds vers des nœuds voisins des niveaux supérieur et inférieur et vers l'anneau de niveau 1. Les liens raccourcis entre les niveaux, appelés liens papillons permettent de parcourir des grandes distances.

Le routage s'effectue en $O(\log N)$ sauts suivant trois étapes. D'abord la requête parcourt le graphe en remontant jusqu'au niveau supérieur. Ensuite elle effectue la descente en s'approchant de la clé puis s'arrête lorsqu'aucun pair de niveau inférieur ne se rapproche de la clé. La requête continue la recherche dans l'anneau même jusqu'à localiser la destination.

Ulysses propose une amélioration du graphe papillon (54). La DHT d'Ulysses rajoute de nouveaux liens au graphe papillon pour éviter les liens congestionnés et de nouveaux mécanismes de routage et maintenance plus résistants au *churn*. Dans Ulysses chaque nœuds maintient $O(\log N)$ voisins permettant un routage en $O\left(\frac{\log N}{\log\log N}\right)$.

Routage par décalage : Graphe de Bruijn

Le graphe de Bruijn $B(d, k)$ (55) (56) est un graphe orienté dont les nœuds sont des chaînes de longueur k. Les arcs lient le nœud $x_1x_2 \dots x_k$ aux k nœuds $x_2 \dots x_k y$ avec y dans $\{0..d-1\}$. Le principe du routage vers une clé $k_1k_2 \dots k_k$ est de contacter des nœuds de préfixes respectifs $k_k \dots k_1$. Le plus court chemin vers la clé revient donc à déterminer la plus longue suite de chiffres commune qui est à la fois suffixe de $x_1x_2 \dots x_k$ et préfixe de $k_1k_2 \dots k_k$.

Le graphe de Bruijn permet ainsi la construction de réseaux de degré constant permettant un routage en $O(\log N)$. Ces propriétés intéressantes ont permis la création de réseaux à contenu adressable comme Broose (57) et D2B (58).

Topologie basée sur les petits mondes

DDHT (59) et Symphony (60) sont des DHT qui reposent sur le modèle petit monde (61). La topologie utilisée est un anneau, où les identifiants sont rangés par ordre croissant. Dans Symphony, chaque pair maintient deux types de voisins: des voisins de courte distance dont 2 prédécesseurs et 2 successeurs, et k voisins de longue distance choisis aléatoirement. Un pair DDHT maintient également 2 prédécesseurs et 2 successeurs ainsi que k voisins de longue distance, choisis selon une probabilité proportionnelle à $1/d$, d étant la distance qui sépare le

nœud de son voisin. Le routage s'effectue récursivement en diminuant la distance entre un pair et la clé requise. Symphony assure un routage en un nombre de sauts logarithmique qui s'exprime en $O\left(\frac{\log^2(N)}{k}\right)$ avec un degré de connectivité constant. DDHT, de son côté, génère des routes en $O(\log N)$ avec un degré de connectivité variable.

2.2.5 Tableau comparatif

Des études comparant les performances théoriques des différentes DHTs ont été effectuées dans de nombreux travaux dont (62) (63). D'autres, dans (64) (65) évaluent le comportement des DHTs face au *churn*.

Le Tableau 2. 2 résume les principales propositions de DHTs qui livrent une analyse complète de leurs propriétés théoriques. Pour chacune des topologies, nous présentons le diamètre, le degré, et la flexibilité du voisinage et du routage. Le degré représente la taille de la table de routage. Le diamètre est la plus grande distance, en nombre de sauts, entre deux pairs (19). Enfin, la flexibilité d'une DHT représente le degré de liberté des nœuds lors du choix des nœuds des tables de routage et du prochain saut sur la route.

Si on considère l'exemple de Chord, lors de la sélection des voisins, le *finger* i du nœud A est choisi dans l'intervalle $[A+2^i, A+2^{i+1}[$, ce qui offre 2^i possibilités pour le voisin i. Ceci résulte donc en $N^{logN}/2$ tables de routage possibles. Lors du routage, le premier saut a le choix parmi $\log N$ voisins possibles, le deuxième $(\log N)$-1, … Il existe donc en tout $(\log N)\,!{=}c\log N$ routes possibles (c constante). Ce dernier paramètre influe sur la résistance du protocole face aux pannes des nœuds (66).

Tableau 2. 2 Performances des différentes DHTs

DHT	Diamètre	Degré	Flex voisinage	Flex routage
Tapestry	$O(\log_{2^b} N)$	$O\left((2^{b-1}) * \log_{2^b} N\right)$	$N^{\frac{\log N}{2}}$	1
Pastry	$O(\log_{2^b} N)$	$O\left((2^{b-1}) * \log_{2^b} N\right)$	$N^{\frac{\log N}{2}}$	2^b
Chord	$O(\log N)$	$O(\log N)$	$N^{\frac{\log N}{2}}$	$c \log N$
Kademlia	$O(\log_{2^b} N)$	$O(k * \log_{2^b} N)$	$N^{\frac{\log N}{2}}$	2^b
CAN	$O\left(d\, n^{1/d}\right)$	$O(d)$	d	$c \log N$
Viceroy	$O(\log N)$	$O(1)$	1	1

Avec :

- b : nombre de chiffres dans l'alphabet utilisé
- N : nombre de nœuds
- k : (pour Kademlia) le nombre de voisins dans chaque sous-arbre d'un nœud
- d : (pour CAN) le nombre de dimensions $(d = 2^b)$

La première remarque concernant les caractéristiques théoriques des DHTs réside dans la similitude des propriétés. En particulier, le nombre de sauts moyen pour router une requête s'exprime souvent en $O(\log N)$. La différence majeure entre les différentes propositions de DHTs porte sur le degré de connectivité et le degré de flexibilité. Pour le premier critère, on distingue deux types de connectivités : les DHTs qui présentent un degré de connectivité logarithmique, et celles qui présentent un degré de connectivité constant. L'aspect logarithmique assure le passage à l'échelle et est donc une propriété intéressante pour l'usage dans un environnement P2P à une grande échelle. L'aspect constant du degré est très avantageux pour deux raisons : comme pour le premier cas, il permet d'assurer le passage à l'échelle en assurant que la taille de la table de routage sera constante quelque soit la taille du réseau. Il permet en plus de maîtriser les coûts de la maintenance des nœuds du réseau, car la mise à jour des entrées de la table de routage dépend du nombre de pairs voisins. Cette caractéristique de performances est importante dans le cas de réseaux de très grande taille. Concernant le critère de flexibilité, nous remarquons que les DHTs à degré constants sont moins (voire non) flexibles. Notons que la DHT de Chord est la plus flexible. La DHT de

CAN assure une flexibilité dans routage mais pas dans le choix du voisinage, contrairement à Tapestry, Pastry et Kademlia. Enfin, Viceroy est la DHT la moins flexible et donc la moins résistante face aux pannes (66)

La majeure partie de ces propositions de DHT a été effectuée entre 2001 et 2003. Actuellement, le domaine des DHTs s'élargit et les travaux concernent principalement la comparaison des différentes topologies, et la manière dont elles peuvent évoluer. Dans la partie qui suit, nous présentons les travaux d'optimisations qui ont permis l'amélioration des systèmes P2P basés sur les DHTs.

2.3 Optimisations possibles

Plusieurs travaux d'optimisation ont été proposés dans le but d'améliorer les performances des DHTs notamment en termes de :

- longueur moyenne des recherches/des routes : le nombre de sauts moyen nécessaire pour le routage d'une requête avec succès,
- délai des recherches : le délai moyen pour le routage d'une requête avec succès,
- taux de succès des requêtes : la fraction de requêtes de recherches abouties
- surcharge du réseau (*overhead*) que nous appelons aussi coût des recherches ou encore coût de signalisation : nombre et taille des messages de signalisation nécessaires pour la maintenance du réseau et le routage des requêtes de recherches.

Dans ce qui suit, nous résumons les différentes approches d'optimisation effectuées dans le cadre des systèmes P2P basés sur les DHTs.

2.3.1 Réseaux hétérogènes et réseaux hiérarchiques

Le principe des réseaux P2P est la distribution égalitaire des responsabilités entre les pairs. Cependant, dans la pratique, les nœuds du réseau sont fortement hétérogènes et possèdent différentes capacités de mémoire, calcul et bande passante. Il serait donc intéressant d'exploiter les ressources présentes dans certains nœuds, en leur attribuant plus de fonctionnalités. Les systèmes décentralisés hiérarchiques comme Gnutella 0.6 et giFT FastTrack en sont un exemple (67). Dans ces systèmes, les super nœuds, organisés selon un graphe aléatoire, assurent les fonctions de routage et de recherche et gèrent les pairs qui y sont rattachés.

Certains travaux comme TOPLUS (68), S-Chord (69) et Super Pastry (70) proposent de calquer ce même schéma hybride dans un réseau logique structuré, c'est à dire avec des super nœuds composant la DHT. Dans Super Pastry et S-Chord, le niveau haut est composé de super nœuds organisés selon l'arbre de Pastry et l'anneau de Chord respectivement. Dans Super Pastry, chaque super nœud gère 30 pairs et chaque pair est rattaché 3 super nœuds. S-Chord regroupe les nœuds proches physiquement en sous groupes et sélectionne un super nœud par groupe. Les sous groupes et les groupes sont organisés suivant l'anneau de Chord. TOPLUS propose d'organiser la topologie logique en fonction de la topologie physique. Le réseau TOPLUS se construit par le regroupement des pairs ayant le même préfixe IP. Ces groupes de nœuds sont par la suite organisés récursivement suivant la topologie hiérarchique de l'Internet : les pairs qui sont topologiquement proches dans l'Internet sont organisés en groupes. En plus, les groupes qui sont topologiquement proches sont organisés à leurs tours dans des super groupes, et les super groupes proches en hyper groupes, et ainsi de suite.

Une deuxième alternative propose des systèmes hybrides non hiérarchiques comme Hybrid Chord (71) ou Hétéro Pastry. Hétéro Pastry (70), propose une architecture similaire à Gia (72) mais dans une DHT. La sélection des voisins est ainsi modifiée de façon à ce que chaque pair sélectionne principalement des super nœuds dans sa table de routage. La comparaison de Super Pastry, Hétéro Pastry, Gia et Gnutella 0.6 montre l'avantage de l'approche Hétéro Pastry en termes de taux de succès, de délai de routage et de trafic de signalisation.

Cependant, dans les deux cas, la gestion des super nœuds s'avère délicate (73). En effet, la procédure de sélection et de maintenance des super nœuds est complexe. Plusieurs problèmes se posent (74) (75): Quel sont les critères de choix à considérer ? À partir de quand un nœud peut-il devenir super nœud ? Combien de super nœuds doivent-ils exister dans le réseau ? Combien de nœuds gère un super nœud ? Que faire si un super nœud tombe en panne ?

2.3.2 Extension du voisinage

Certains travaux proposent d'élargir le voisinage des nœuds pour accélérer le routage. (76) propose de multiplier la taille de la table de routage de Chord par un facteur d en choisissant les voisins *finger* selon $\mathrm{id}\,(\mathrm{succ}\,(\mathrm{id}\,(\mathrm{x}) + (1 + 1/\mathrm{d})^{\mathrm{i}-1}))$ ce qui résulte en des routes de longueur moyenne égale à $\log \mathrm{N}/((1 + \mathrm{d}) \log (1 + \mathrm{d})\, \mathrm{d} \log (\mathrm{d}))$. Par conséquent, lorsque le facteur d augmente l'amélioration est moins perceptible, sans oublier l'augmentation du trafic de signalisation nécessaire à la maintenance des voisins *finger* additionnels. Pour pouvoir

router dans les deux directions, Symmetric Chord (77) propose de choisir des fingers selon $id\ (succ\ (id\ (x)+4^{i-1}))$ et $id\ (pred\ (id\ (x)-4^{i-1})),\ 1< i<m$. D'autres comme Bi-Chord (78) and Janus (79) proposent de rendre les liens bidirectionnels, en rajoutant pour chaque *finger* le lien inverse. Bi-Chord emploie le routage KBR et Janus la marche aléatoire.

Une deuxième alternative pour élargir le voisinage consiste à augmenter la taille de l'alphabet 2^b, utilisé pour le codage des identifiants. Pour CAN par exemple, passer d'un alphabet binaire à un alphabet *d-aire* est équivalent à l'augmentation du nombre de dimensions de 2 à d. Augmenter la taille de l'alphabet a pour effet d'augmenter le degré de 2 à d et ainsi de diminuer le nombre de sauts par un facteur $\log_2 d$.

Enfin, une autre optimisation consiste à utiliser plusieurs réseaux logiques à la fois, en attribuant à chaque nœud plusieurs identifiants, moyennant plusieurs tables de hachage. Chaque nœud multiplie alors son degré par un facteur r, correspondant au nombre total de tables de hachages utilisées. Cela permet d'une part, de multiplier le nombre de choix possible pour le prochain saut et donc d'améliorer la flexibilité du routage, et d'augmenter la redondance des objets et donc la tolérance aux pannes, d'autre part. En revanche, la maintenance des réseaux logiques devient plus complexe et nécessite notamment une signalisation importante.

2.3.3 Duplication des données

Les tables de hachage distribuées sont définies pour être déployées dans des environnements instables, où les pairs se connectent et se déconnectent aléatoirement, ou encore tombent en panne. Certaines DHTs flexibles assurent le routage même en cas de pannes, par le biais de chemins secondaires. Cependant, la défaillance d'un pair implique la perte définitive des données qu'il stockait. Autrement dit, si le routage dans une table de hachage distribuée est tolérant aux pannes, le stockage de données ne l'est pas. Les systèmes P2P doivent, par conséquent, prendre en compte le comportement volatile des pairs du réseau, et mettre en place des méthodes de redondance afin d'assurer la pérennité des données. Une solution consiste à dupliquer les données de la table sur plusieurs nœuds.

La duplication des données consiste à copier un objet sur un certain nombre de nœuds afin d'augmenter la disponibilité et l'accessibilité des objets dans les systèmes n'utilisant pas de méthodes de recherche exhaustive. Pour cela, plusieurs solution existent à savoir : la mise en cache, la réplication ou encore '*l'erasure coding*'.

La mise en cache

La mise en cache appelée aussi réplication passive est la mise à disposition dans le réseau des objets partagés une fois récupérés. Les nœuds disposant d'une ressource suite à son obtention, signalent leur mise à disposition de la ressource, par insertion de sa clé dans la DHT. La popularité d'une ressource entraine ainsi la création de nombreux réplicas.

La réplication

La réplication de clés permet à un nœud de définir un certain nombre de responsables pour une clé en copiant l'objet correspondant dans plusieurs nœuds. La réplication permet de répartir ainsi la charge due aux objets populaires. L'avantage de la réplication par rapport à la mise en cache est que le nœud responsable d'une clé peut maîtriser le nombre et l'emplacement des copies de ses clés. Le coût supplémentaire pour chaque nœud est alors une quantité de mémoire complémentaire nécessaire pour le stockage des réplicas additionnels. Le coût pour le réseau est une quantité de signalisation supplémentaire pour l'insertion et la maintenance régulière des réplicas. Ces coûts sont directement liés aux nombre de réplicas à insérer et à l'emplacement de ces derniers.

Le nombre de réplicas et le choix des nœuds répliquants varient selon la géométrie de la DHT et les exigences de l'application (délais, dynamisme…). Nous discuterons cette problématique plus tard dans le chapitre 4.

L'erasure coding : duplication par fragmentation et codage

Dans la technique de *erasure coding*, chaque objet à répliquer est divisé en m fragments puis codé sur n fragments tel que $n > m$. Ceci suppose donc un taux de réplication r égale à n/m. La principale caractéristique de la méthode *erasure coding* est que l'objet peut être récupéré à partir de n'importe quels m fragments choisis. Cette méthode est employée dans le système de stockage et partage de fichiers distribué Wuala (80).

La duplication par fragmentation et codage permet d'avoir une meilleure disponibilité des données par rapport à la réplication (81). Les résultats de (82) montrent que pour un niveau de *churn* faible, la réplication requiert un facteur de réplication plus important pour garantir un même taux de disponibilité des données, et demande par conséquent plus de bande passante et d'espace mémoire pour le stockage des données.

En revanche, la méthode *erasure coding* rend le système plus complexe, à cause de la fragmentation et du codage/décodage, d'une part, et la maintenance des multiples blocs

d'autre part. De plus, la récupération des données avec cette technique induit un temps de latence plus élevé. En effet, avec une réplication classique on récupère l'objet le plus proche, alors que la fragmentation requiert la récupération et le décodage des m blocs. De plus pour récupérer une partie de l'objet, on doit d'abord récupérer tous les blocs pour pouvoir faire le décodage, et extraire le bloc recherché. Dans la réplication classique il suffit juste de récupérer le bloc désiré.

Par ailleurs, dans la pratique, un objet n'est disponible que si au moins m fragments de l'objet sont disponibles. Cela veut dire que pour récupérer un objet, il faut que les m fragments soient disponibles et que les routes menant vers les nœuds partageant ces fragments soient également disponibles. Au contraire, dans la réplication, pour récupérer un objet, il faut qu'au moins un des r réplicas et la route correspondante soient valides. Le *churn* affecte beaucoup plus la méthode *erasure coding* que la réplication. Les résultats de (82) montre que la réplication par fragmentation et codage réalise une meilleure performance par rapport à la réplication quand la validité d'un nœud est comprise entre 80% et 90%. Au delà de cette valeur, l'amélioration n'est pas visible vu que le degré de disponibilité est très élevé. Au dessous de cette valeur, la maintenance requise devient contraignante et inhibe les performances du système. Cette méthode semble donc intéressante pour les applications exigeant une disponibilité maximale des données dans un réseau faiblement dynamique.

2.3.4 Méthodes de recherches floues

Le point critique des systèmes P2P structurés est leur incapacité de faire des recherches complexes. Or vu l'engouement important pour les systèmes de partage de fichiers, la recherche par mot clé se révèle plus importante que la recherché exacte. En effet, '*most request are for hay, not for needle*' (83)

Les méthodes de recherche aléatoire classiques utilisées dans les DHTs se révèlent inefficaces, notamment si la donnée que l'on recherche n'est pas assez répliquée (84). Une alternative présentée dans (85) propose d'utiliser l'inondation bornée dans la DHT. La nature structurée du réseau permet en effet, de limiter l'inondation à un ensemble de nœuds afin d'éviter le nombre de requêtes de recherche émises. L'étude dans (70) montre que l'inondation bornée dans la DHT génère certes un trafic de signalisation supérieur à celui de la recherche exacte. Toutefois, cette quantité de signalisation reste toujours inférieure à celle générée dans un réseau non structuré. Une autre solution décrite dans (86) propose d'utiliser la recherche exacte et l'inondation. Le système lance d'abord la recherche exacte dans la DHT. Si le

nombre de réponses générées est inférieur à un certains seuil, après un intervalle de temps prédéfini, la recherche par inondation est déclenchée. D'autres systèmes hybrides comme GAB (87) et PIERSearch (88) emploient à la fois, l'inondation pour la recherche des objets populaires, et la recherche exacte pour les objets rares. A la différence de PIERSearch, GAB collecte dynamiquement des statistiques sur la popularité des documents partagés pour décider de la méthode de recherche à employer. Comparé à PIERSearch, GAB produit des temps de réponses plus courts de 25 à 50%, et réduit l'utilisation de la bande passante de 45% en moyenne.

Afin de permettre des recherches sur des mots morphologiquement similaires dans les DHTs, (89) propose d'ôter quelques caractères à la fin du mot et d'attribuer une clé à chaque 'sous mots' résultants. Ces variations permettent par exemple d'ôter les pluriels ou de chercher un préfixe de mot. Toutefois, le problème avec cette technique de recherche est que le nombre de clés résultantes peut parfois être très important ; si tous les objets relatifs à un mot (sous mot) clé seront stockés dans un seul nœud, ce nœud sera surchargé. Pour pallier ce problème, KSS (90) propose d'attribuer une clé à des combinaisons de mots (*k* mots) au lieu d'un seul. Suivant le paramètre *k*, la DHT attribue une clé à toutes les combinaisons de *k* mots possibles. Ceci permet d'une part, de faire des recherches plus fines et d'autre part, d'alléger la charge des nœuds. La comparaison de KSS avec les méthodes de recherches par listes inverses (91) montre que la quantité de signalisation générée pour l'insertion des clés par KSS est plus importante. Vu que KSS insère des combinaisons des mots et non le mot seul, le nombre de clés insérées sera plus important. Pour des requêtes avec 5 mots par exemple, KSS insère 92.5 millions clés contre 12,1 millions pour les listes inverses. En revanche, la quantité de signalisation générée par les recherches dans KSS est nettement inférieure. 90% des requêtes KSS génèrent une signalisation inférieure à 100 Ko, contre 50% seulement pour les listes inverses. En effet, le système KSS répond aux requêtes de recherche par l'intersection des différents mots clés alors que la liste inverse rassemble toutes les réponses, correspondant chacune à un mot clé, puis fait le tri en croisant les réponses entre elles.

2.3.5 Rapprocher le réseau physique du réseau logique

Un système pair-à-pair dont le réseau logique est proche du réseau physique suppose que les nœuds voisins dans le réseau logique sont proches dans le réseau physique, assurant ainsi un délai plus court entre chaque saut logique.

Plusieurs travaux se sont attaqués au problème de la proximité physique dans le réseau logique. Tapestry et Pastry proposent de sélectionner pour chaque nœud les voisins les plus proches lorsque plusieurs nœuds candidats se présentent. Dans LPRS (92) on propose un nouveau mécanisme de sélection des voisins basé sur le calcul des latences. Les techniques PNS et PRS proposées dans (66) modifient le choix des nœuds voisins et des routes en choisissant, respectivement, les voisins et les routes les plus proches physiquement.

Dans ce travail, nous nous intéressons à la problématique de la proximité physique et logique en particulier dans les réseaux MANETs P2P basés sur les DHTs. Le croisement des deux domaines semble en effet intéressant vu la convergence des deux systèmes. Certaines propriétés communes telles que, la décentralisation, l'auto organisation et la spontanéité motivent le croisement des deux domaines. D'autres en revanche, telles que la non fiabilité des liens, la volatilité des nœuds et la surcharge importante en signalisation met en doute l'efficacité de tels systèmes.

Dans le cadre du croisement P2P MANET, deux approches de routage s'opposent : le routage non intégré et le routage intégré.

2.3.5.1 Routage non intégré

Il s'agit de superposer deux couches : la DHT au niveau applicatif et le protocole de routage MANET au niveau réseau. Les deux fonctions sont complètement isolées l'une de l'autre et le calcul des routes physiques et logiques se fait séparément. Dans ce cas, il y a deux tables de routage et de voisinage et deux types de messages de contrôle. Lorsqu'un un nœud A cherche un objet qui se trouve dans un nœud B, il consulte d'abord sa table de routage logique et sélectionne le prochain saut dans le réseau logique. Or, un voisin logique n'est généralement pas un voisin physique. Le calcul de la route physique vers le prochain voisin physique, dans ce cas, se fait moyennant le protocole de routage MANET. Cette opération se répète à chaque saut logique jusqu'à la destination.

Nous avons évalué à l'aide d'émulations les performances des réseaux P2P MANET résultants. En particulier, nous avons analysé l'impact de la volatilité des nœuds, gérées par un algorithme de routage MANET, sur un système P2P de localisation des ressources structuré.

Pour le calcul de la charge du réseau , nous avons choisi de tester la faisabilité de l'algorithme Chord dans un MANET avec les protocoles de routage réactif AODV et proactif OLSR. L'objectif étant d'évaluer la faisabilité de déploiement d'un réseau P2P MANET, dans un

environnement quasi réel moyennant des émulations. Les détails des protocoles et de la plate forme des émulations sont publiés dans (93).

Le Tableau 2. 3 présente les résultats de performances des systèmes OLSR_Chord et AODV_Chord dans un réseau de 100 nœuds, faisant varier le paramètre mobilité (nombre de connexion/déconnexion par seconde).

Tableau 2. 3 Effet de la mobilité MANET sur un réseau P2P

	OLSR_Chord					**AODV_Chord**				
Fiabilité	**TSR>95% Elevé**			**85%<TSR<95% Moyenne**	**TSR<85% Faible**	**TSR>95% Elevé**	**85%<TSR<95% Moyenne**		**TSR<85% Faible**	
Mobilité	**0**	**1\20**	**1\15**	**1\10**	**1\5**	**0**	**1\20**	**1\15**	**1\10**	**1\5**
(a) Taux de succès (TSR) %	100	98	95	85	84	100	94	90,5	77,2	65,6
(b) Délai (sec)	0,27	0,29	0,3	0,4	0,49	0,3	0,35	0,55	0,68	0,88
(c) Nbre de msg MANET (10^3)	655	668	644	674	680	1343	1499	1582	1651	1878
(d) Taille des msg MANET (Mo)	196	208	210	214	299	31	32	33	32	35
(e) Nbre de msg DHT (10^3)	358	373	388	386	404	393	374	371	396	414
(f) Taille de msg DHT (Mo)	43	46	45	47	49	45	55	46	49	50

Nous observons que dans un réseau statique, on récupère 100% des clés pour les deux systèmes. Plus la mobilité augmente, moins les requêtes aboutissent (Tableau 2. 3(a)). Le protocole AODV_Chord est plus sensible à la mobilité. En effet, pour toutes les valeurs de mobilité, le taux de succès est inférieur à 95% et le délai est nettement supérieur à celui d'OLSR_Chord. La maintenance proactive d'OLSR réduit l'effet de la mobilité puisque les routes sont vérifiées périodiquement et donc réparées immédiatement en cas de pannes.

Nous avons choisi de calculer la quantité de signalisation produite par la DHT et le protocole de routage séparément. Nous remarquons que la signalisation générée par le protocole de routage est nettement supérieure à celle de la DHT. Aussi, le nombre et la taille des messages générés pour la maintenance de la DHT varient très peu avec la mobilité (Tableau 2. 3 (e, f)).

Toutefois, le nombre et la taille des messages générés par le protocole de routage diffèrent. AODV_Chord, bien qu'il soit proactif, génère plus de messages de signalisation (Tableau 2. 3 (c)), mais emploie des messages de plus petite taille (Tableau 2. 3 (d)). La diffusion par MPR (94) dans OLSR réduit le nombre de messages véhiculés, ce qui explique le premier résultat. De plus, les nœuds OLSR s'échangent périodiquement leurs tables de routage et de voisinage.

Ces messages décrivent toute l'architecture du réseau, et donc leur taille dépend de la taille du réseau ; d'où la taille importante du trafic de signalisation. Dans un réseau de 80 nœuds par exemple, la taille moyenne d'un message OLSR est de 331 octets contre 20 octets pour AODV. Nous avons aussi calculé ces valeurs pour un réseau de 40 nœuds avec les mêmes paramètres. La taille moyenne d'un message AODV est égale à 23 octets et celle d'un message OLSR est de 170 octets. L'utilisation d'OLSR_Chord est donc limitée par la taille du réseau bien que le protocole réalise des performances meilleures par rapport à AODV_Chord.

En résumé, nos résultats montrent que le protocole OLSR_Chord présente des performances meilleures que celles d'AODV_Chord, mais produit aussi une forte signalisation et consomme plus de ressources. En effet, la double maintenance requise limite considérablement les performances du système. OLSR_Chord conviendrait alors dans le cas des réseaux de tailles moyennes exigeant un délai minimal et une haute fiabilité. Les résultats montrent aussi que le protocole AODV_Chord est peu approprié aux réseaux fortement mobiles.

L'indépendance totale entre les deux niveaux résulte en deux réseaux complètement isolés et requiert par conséquent une double maintenance. L'intégration des deux niveaux de routage est donc indispensable afin de réduire la signalisation résultante. Il s'agit de faire communiquer les deux niveaux afin d'assurer un routage plus optimal. La structure de la DHT, dans ce cas, sera utilisée pour localiser les données et acheminer les messages selon des routes logiques, calculées tenant compte de l'architecture physique réelle : d'où l'appellation 'routage intégré'.

2.3.5.2 Routage intégré

Dans ce schéma, les deux niveaux communiquent entre eux. On utilise les données topologiques du niveau physique pour le routage logique et inversement. Une seule table de routage est utilisée pour les fonctions de routage logique et physique. Ce croisement permet de diminuer la maintenance requise et d'accélérer les requêtes de recherche puisque l'on considère la proximité physique et logique en même temps.

Cette problématique a été déjà traitée et plusieurs solutions ont été proposées ; nous citons : SSR (95) (96), MADPastry (97) (98), VRR (99), DHT_OLSR (100), ou encore DPSR (101) (102). Nous détaillons à titre d'exemple de protocole MADPastry.

MADPastry combine la DHT de Pastry et la technique de routage réactif d'AODV. Pour trouver la route, Pastry calcule d'abord le prochain saut logique puis AODV calcule la route correspondante. Afin de diminuer le trafic de contrôle, MADPastry utilise le mécanisme d'*overhearing :* chaque fois qu'un nœud intermédiaire reçoit un paquet, il intercepte toutes les

informations de contrôle (expéditeur, destination…) et les enregistre dans son cache local. Ce mécanisme de cache est utilisé pour réduire l'information de contrôle d'une part, et pour accélérer les requêtes de recherches d'autre part. Pour rapprocher la topologie physique du niveau logique, MADPastry utilise le concept des clés *landmark*. Ce sont des clés bien définies connues par tous les nœuds du réseau. Un nœud successeur de cette clé et donc responsable de cette clé, devient un nœud *landmark* et diffuse périodiquement des messages *beacon* pour annoncer à son voisinage physique son identité. Chaque nœud choisit son nœud *landmark* suivant les informations qu'il reçoit en continue dans le réseau. Quand un nœud détecte un nouveau nœud *landmark*, il change le préfixe de son ID de manière à avoir le même préfixe que le nœud *landmark* auquel il s'est rattaché. Ainsi les nœuds proches physiquement vont avoir le même nœud *landmark* et le même préfixe et par la suite vont être proche logiquement.

Le Tableau 2. 4 présente une analyse comparative des principales propositions P2P MANETs existantes.

Tableau 2. 4 Comparaison des approches P2P MANETs

Approches	Protocoles	Proximité physique	Maintenance	Résistance à la mobilité	Utilisation de cache	Performances
SSR	Chord & DSR	Choix du voisin physique proche logiquement	Overhearing	Stocker les voisins physiques du voisin logique	Oui	Bien adapté aux réseaux denses mais pas à la mobilité
MADPastry	Pastry&AODV	Clé landmark	Overhearing	Non	Oui	Bien adapté aux réseaux denses et mobiles mais pas au faible trafic
DHT OLSR	Pastry & OLSR	Clé landmark	Overhearing ; maintenance locale OLSR	Non	Oui	Bien adapté aux réseaux denses mais génère une forte signalisation
DPSR	Pastry & DSR	Non	Overhearing	Stocker plusieurs routes physiques vers un voisin logique	Oui	Bien adapté aux réseaux denses mais pas à la mobilité
VRR	Routage KBR en anneau et routage physique proactif	Connaissance du voisinage physique direct	Maintenance périodique et overhearing	Utiliser les routes secondaires pour contourner les pannes	Oui	Bien adapté aux réseaux denses mais génère une forte signalisation

2.3.6 Equilibrage de charge

Lorsque le nombre de nœuds dans à un système pair-à-pair augmente, le nombre de messages transitant dans le réseau augmente d'autant. Il devient donc nécessaire de répartir la charge des messages parmi tous les nœuds du réseau, afin d'éviter les points de concentration.

Les auteurs de (103) proposent e_Chord, une variante de Chord qui tend à équilibrer les degrés des nœuds afin de répartir uniformément les messages entres les pairs. Pour ce faire,

e_Chord propose de modifier la procédure de choix des *fingers.* Les liens dans e_Chord sont unidirectionnels, et le routage se fait exactement comme dans Chord moyennant les *fingers.*

DASIS (104) propose d'utiliser une nouvelle procédure pour l'insertion des nouveaux nœuds dans la DHT, en dirigeant les requêtes *join* vers le nœud le moins chargé (le nœud de profondeur minimale dans un arbre ou le nœud responsable du plus grand intervalle dans un anneau). Ceci est rendu possible par l'échange périodique d'information topologique entre voisins dans l'overlay. Le nœud se voit donc attribuer une identité suivant l'emplacement choisi. L'évaluation de l'algorithme DASIS montre que dans un réseau de 2108 pairs, 42% des nœuds présentent une profondeur optimale contre seulement 5% pour une DHT classique.

Les auteurs de (105) (106) proposent un mécanisme d'équilibrage de charge qui repose sur les serveurs virtuels. Chaque pair correspond à un ou plusieurs serveurs virtuels dans l'overlay. Lorsque la charge (nombre de requêtes entrantes) au niveau d'un serveur virtuel dépasse un certain seuil, une partie des données stockées dans ce nœud sera transférée vers un autre serveur virtuel moins chargé. Cette procédure de transfert est équivalente à un départ et une arrivée de nœud. L'utilisation des serveurs virtuels produit un taux d'utilisation de nœuds égale à 99.9 %, mais augmente le trafic de signalisation de 10 à 20%.

La solution proposée dans (107) consiste à refuser les requêtes de recherche au delà d'un certain seuil. La recherche se dirige ainsi vers le nœud qui partage une copie de l'objet. Ceci suppose donc que les objets sont auparavant répliqués. Cette solution permet une répartition plus optimale des requêtes de recherche entre les différents nœuds, et une récupération plus rapide de l'objet.

2.4 Conclusion

Nous avons présenté dans ce chapitre un état de l'art des systèmes pair-à-pair structurés basés sur les tables de hachage distribuées. Dans un premier temps, nous avons introduit le concept des tables de hachage distribuées et définit leurs principales caractéristiques. Ensuite, nous avons passé en revue les principales architectures DHTs en les classant suivant leur modèle topologique. Nous avons particulièrement détaillé la DHT de Chord, Pastry et Kademlia, les trois DHTs sur lesquelles nous nous sommes concentrés dans le cadre de cette thèse.

Dans la dernière partie du chapitre, nous avons exposé quelques travaux d'optimisations proposées dans le cadre des structures DHTs, notamment l'exploitation de l'hétérogénéité des pairs pour optimiser le routage. Comme nous l'avons vu, ces propositions reposent le plus

souvent sur des systèmes hiérarchiques basés sur les super nœuds. Ces systèmes hiérarchiques essaient de combiner les systèmes P2P centralisés et distribués en formant un réseau logique distribué, où chaque super nœud agit comme une entité centrale pour ses nœuds feuilles. Deux problèmes se posent dans ce cas : la sélection et la maintenance des super nœuds, et la méthode de recherche à employer. De plus, pour répondre au deuxième problème, il faut toujours trancher entre deux options : organiser les super nœuds selon un graphe aléatoire et employer la recherche aléatoire au prix d'un coût de signalisation important, ou déployer une structure DHT et utiliser la recherche KBR mais se limiter aux recherches exactes. Dès lors, une question s'impose : faut-il toujours opposer les systèmes centralisés aux systèmes distribués et la recherche KBR à la recherche aléatoire ?

A partir de ces observations, nous proposons dans le chapitre 4 une nouvelle structure nommé Power_DHT, qui au lieu d'opposer les systèmes structurés aux non structurés, réunit les deux principes. Power_DHT propose une structure hétérogène, mais non hiérarchique, et intègre à la fois la recherche KBR et la recherche aléatoire. Nous détaillons dans la suite le travail qui a mené à la proposition Power_DHT et nous présentons l'architecture et les fonctionnalités de cette nouvelle structure.

CHAPITRE 3.

REVERSE_DHT ET POWER_DHT : VERS UNE STRUCTURE DECENTRALISEE

Les réseaux à contenu adressable permettent un routage efficace pour la recherche des objets, au prix d'un maintien d'une structure d'interconnexion entre nœuds (section 2.2). Ces systèmes sont particulièrement intéressants en termes de performances lorsqu'on peut facilement associer une clé à un objet, c'est à dire lorsque les objets peuvent être nommés d'une manière unique, avec un format spécifique. Nous avons vu dans la section 2.2.5 que la majorité des DHTs présente des performances théoriques assez similaires, et leurs différences portent surtout sur la taille constante ou logarithmique de leur table de routage. Les topologies résultantes peuvent former un anneau, un arbre ou encore un hypercube. Nous proposons dans ce chapitre d'étudier de plus près la topologie des DHTs qui résulte de l'interconnexion entre les nœuds voisins dans l'espace logique.

La motivation de nos travaux part de l'observation d'une dichotomie dans le graphe d'interconnexion de la DHT. Alors que les algorithmes DHTs existants sont purement égalitaires, nous constatons une forte hétérogénéité dans la connectivité des nœuds dans les réseaux déployés (103) (108) . Notre idée est d'exploiter cette hétérogénéité entre les nœuds de la structure. Pour cela, nous proposons Power_DHT, une nouvelle structure d'interconnexion et de routage qui transforme dynamiquement la DHT vers une structure décentralisée, exhibant les propriétés d'un graphe sans échelle (faible diamètre, distribution des degrés….).

Dans ce qui suit, nous présentons la structure Power_DHT et détaillons les nouveaux modèles d'interconnexion et de routage qu'elle propose. Nous choisissons dans ce travail d'évaluer Power_DHT appliquées à trois géométries différentes : l'anneau de Chord, l'arbre de

Kademlia et la structure hybride (anneau/arbre) de Pastry. Pour cela, nous identifions les paramètres applicables à toutes les DHTs, ainsi que des idées spécifiques pour chacune. Nous évaluons ensuite notre proposition par simulations et vérifions comment Power_DHT réduit le diamètre des recherches tout en réduisant le coût de signalisation.

3.1 Observations & Motivations

Dans les structures DHTs, chaque nœud calcule et met à jour ses voisins logiques selon l'algorithme et les métriques employées. Le graphe résultant constitue un graphe orienté où chaque nœud possède un nombre prédéfini de liens unidirectionnels vers les voisins calculés. Le degré sortant est un paramètre connu, fixé par la DHT. La distribution des degrés entrants est cependant inconnue. De ce fait, nous proposons d'étudier le graphe interconnectant les voisins logiques dans l'*overlay* structuré. Nous nous intéressons en particulier à trois DHTs différentes à savoir Chord, Pastry et Kademlia. Pour chacune, nous calculons la distribution des degrés entrants.

Pour ce faire, nous considérons un réseau de 2^{14} nœuds dans l'espace des identités $[0,\ 2^{160}[$ (nous travaillons dans la base 2^b avec b=1). Pour chaque nœud nous mesurons le degré entrant et calculons pour chacune de ces valeurs le nombre de nœuds correspondants. La Figure 3. 1 présente la distribution des degrés entrants pour la DHT de Chord et Pastry.

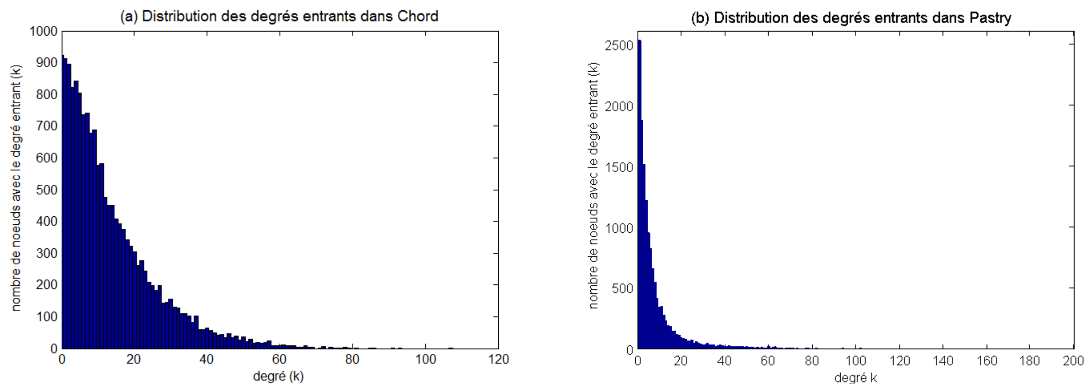


Figure 3. 1 Distribution des degrés entrants dans Chord et Pastry

Nous observons une forte hétérogénéité dans la connectivité des nœuds et ce pour les deux topologies. En effet, la majorité des nœuds présente un degré entrant faible, tandis que très peu de nœuds sont fortement connectés. Pour Chord, par exemple, le degré entrant moyen est 13.31 et la valeur médiane est 10. Cependant, cette moyenne ne reflète pas la distribution réelle. En effet, 80% des nœuds possèdent un degré inférieur à 22 tandis que 5% seulement des nœuds ont un degré supérieur à 55. Pour Pastry, la moyenne et la valeur médiane sont

égales à 11.45 et 4 respectivement. 80% des nœuds possèdent un degré inférieur à 12 et seulement 5% des nœuds possèdent un degré supérieur à 43.

Dans un premier temps, nous supposons que cette distribution résulte de la grande taille de l'espace des identités comparée au nombre de nœuds N, et qui produirait un décalage dans la répartition des zones de responsabilités entre les nœuds. Nous procédons donc à d'autres simulations en réduisant la taille de l'espace des identités. Toutefois, la distribution des degrés reste inchangée. D'où provient donc cette disparité ?

Dans la DHT de Chord, (103) a démontré analytiquement que cette disparité provient de la différence entre la taille des zones assignées à chaque nœud. En effet, l'étendue de la zone d'un nœud Chord est proportionnelle au nombre de clés, et donc *fingers* dont il est responsable (Figure 3. 2). Le degré entrant d'un nœud dépend donc de la taille de sa zone dans l'espace des identités et suit une distribution de la forme (109) :

$$P(f) = \binom{F}{f} N \frac{f!\,(F+N-f-1)!}{(F+N)!}$$

f désigne le degré entrant, F est la somme des degrés entrants des N nœuds.

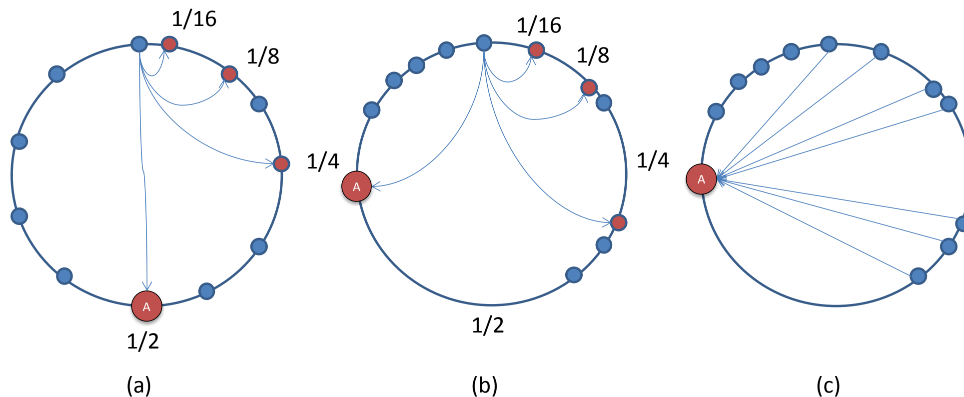


Figure 3. 2 (a) Distribution quasi uniforme des zones de responsabilité, (b) Distribution non uniforme des zones de responsabilité dans Chord, (c) liens vers le nœud A, responsable de la plus grande zone

Pour Pastry, nous présumons que cette distribution hétérogène provient également de la différence des zones de responsabilités entre les nœuds. Par ailleurs, nous remarquons que pour Pastry, cette hétérogénéité est plus prononcée. Nous estimons que cette disparité résulte de la procédure de *join* et de la sélection de voisins dans Pastry. En effet, lorsqu'un nouveau nœud Pastry rejoint le réseau, il initie sa table de routage par l'union les tables des nœuds traversés lors du *join.* Les nœuds ayant rejoint le réseau en premier seront donc plus présents dans les tables de routage des nouveaux arrivants. De plus, Pastry se base sur la proximité physique pour choisir ses nœuds voisins. Lorsqu'un nœud met à jour ses voisins, il ne

remplace une ancienne entrée de sa table que si le voisin correspondant ne répond plus ou qu'un second voisin plus proche physiquement est trouvé. Le fait de choisir un critère de sélection, semble donc accentuer la disparité entre les nœuds.

Pour Kademlia, nous mesurons la distribution des degrés pour le réseau Kademlia (k=8) (Figure 3. 3(a)). Le paramètre k définit la taille du *bucket* et donc la taille de la table de routage.

Le résultat de Kademlia diffère de celui de Pastry et Chord. En effet, nous observons une distribution relativement homogène où la majorité des nœuds possèdent un degré autour de 20. La moyenne et la valeur médiane sont 20.56 et 20 respectivement. Les nœuds dans Kademlia mettent à jour leurs tables de routage à chaque message reçu, en stockant si nécessaire les informations relatives au nœud expéditeur. La métrique XOR étant symétrique, les voisins d'un nœud A ont donc réciproquement A comme voisins. Cela permet au nœud de recevoir des messages depuis le même ensemble de nœuds que celui contenu dans sa table (qui correspond à l'ensemble des nœuds contactés).

Nous remarquons aussi l'existence de quelques nœuds à degré élevé. Ceci est une conséquence de la procédure de choix des nœuds des tables de routage. En effet, une fois sa table remplie, le nœud Kademlia n'insère une nouvelle entrée que si son voisin le moins récent s'est déconnecté. Comme pour Pastry, les premiers nœuds à rejoindre le réseau seront donc les nœuds les plus connectés. Pour confirmer notre supposition nous mesurons la distribution des degrés dans un deuxième réseau Kademlia (k=2). Nous remarquons d'après la Figure 3. 3(b) que la distribution devient très hétérogène. En effet, la taille du *bucket* étant limitée, les premiers nœuds insérés demeurent dans la table tant qu'ils seront connectés. Les premiers nœuds arrivés seront par conséquent les plus connectés.

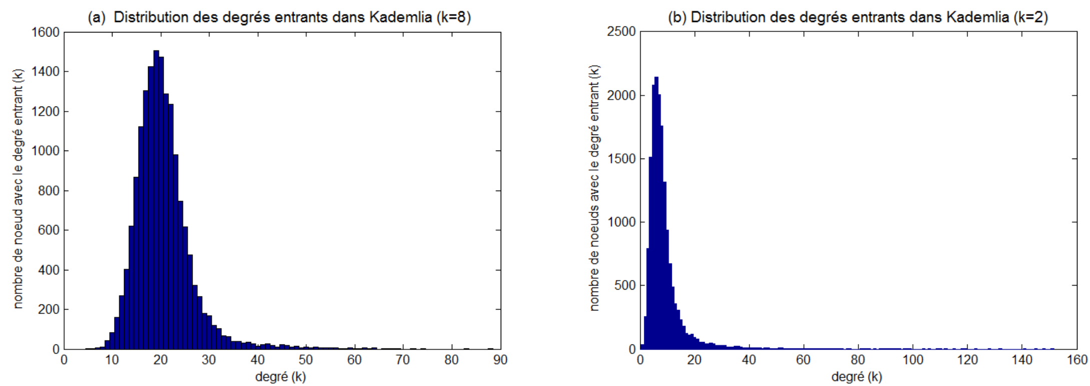


Figure 3. 3 Distribution des degrés entrant dans Kademlia

L'interconnexion entre les nœuds de la DHT n'est donc pas égalitaire. Contrairement au degré sortant, fixant la taille des tables de routage, le degré entrant varie fortement. Certains nœuds du réseau sont fortement connectés tandis que la majorité des nœuds possède un degré entrant très faible. Par construction, les nœuds de la DHT emploient pour le routage les liens sortants uniquement. En effet, les tables de routage maintiennent seulement les liens vers les voisins, et ignorent les liens entrants. Il serait donc intéressant de pouvoir exploiter les liens entrants pour le routage d'autant plus que certains nœuds présents dans la DHT sont fortement connectés.

A partir de ces observations, nous proposons dans un premier temps une nouvelle structure, qui maintient en plus les liens inverses, que nous appelons Reverse_DHT. Reverse_DHT est une extension de l'algorithme de la DHT habituelle et conserve donc toutes les fonctionnalités et les structures de la DHT classique. Le procédure de routage est légèrement modifiée pour tenir compte de la nouvelle structure.

Dans ce qui suit, nous détaillons notre approche Reverse_DHT et montrons comment la nouvelle structure exploite l'hétérogénéité des degrés d'interconnexion lors du routage pour raccourcir les recherches.

3.2 1ère étape : Reverse_DHT, approche de routage bidirectionnel

3.2.1 Architecture Reverse_DHT

Afin de maintenir les liens entrants, nous introduisons une nouvelle table nommée $reverse_{table}$. Pour chaque nœud, cette table maintient la liste des nœuds qui l'ont sélectionné comme voisin : nœud *reverse*. La taille de cette table varie selon la distribution des degrés entrants. La $reverse_{table}$ contient trois champs : $\left(reverse,\ seq_{neigh},\ time\right)$

- Le paramètre *reverse* contient l'identifiant du nœud correspondant au lien inverse codé sur 20 octets.
- Le paramètre seq_{neigh} contient la liste des identifiants des voisins séquentiels du nœud *reverse.* L'identifiant d'un seq_{neigh} est codé sur 20 octets. Nous détaillerons ce paramètre dans la partie qui suit.
- Le paramètre *time* est le temps d'insertion du nœud *reverse* codé sur 8 octets. Une entrée *reverse* reste valide pendant une durée t. Cette information permet au nœud de supprimer les entrées *reverse* obsolètes une fois le temps $(time + t)$ atteint.

Reverse_DHT est une extension de la DHT originale. Elle conserve donc toutes les structures, tables, protocoles et fonctionnalités de la DHT de base. En plus, elle garde en mémoire la liste des liens entrants. Nous détaillons dans la suite la procédure de maintenance de la nouvelle table et la procédure de routage, modifiée en conséquence.

3.2.1.1 Maintenance

La procédure de maintenance de la nouvelle table $reverse_{table}$ est relativement simple et ne requiert pas de signalisation supplémentaire. Pour maintenir les liens entrants, nous proposons dans Reverse_DHT d'employer le même message de signalisation utilisé pour la maintenance des voisins dans la DHT classique. Habituellement, chaque nœud de la DHT doit envoyer périodiquement un message *refresh* pour vérifier si le voisin en question existe toujours. Ce message contient notamment l'identifiant du nœud expéditeur $source_{node}$. Nous proposons d'étendre ce message avec un champ supplémentaire seq_{neigh} qui contient les voisins séquentiels de la source.

Ainsi, chaque fois qu'un nœud reçoit un message *refresh*, il rajoute à sa $reverse_{table}$ l'entrée correspondante (l'expéditeur du *refresh*). Le champ *reverse* prend la valeur $source_{node}$ et le champ seq_{neigh} prend la valeur seq_{neigh} du message reçu. Enfin, le paramètre *time* enregistre le temps d'insertion de l'entrée. Si l'entrée correspondante existe déjà, le nœud met simplement à jour le champ *time.*

Par défaut, le nœud voisin reçoit périodiquement le message *refresh* tant que le nœud *reverse* est en vie. Après une durée t, si l'entrée n'est pas mise à jour, c'est-à-dire que le nœud voisin ne reçoit plus le message *refresh*, l'entrée correspondante est effacée de la table.

3.2.1.2 Routage KBR

Dans les DHTs habituelles, les nœuds utilisent pour l'acheminement les voisins de la table de routage et les voisins séquentiels. La route vers la clé recherchée est calculée de proche en proche suivant l'algorithme de la DHT. Nous proposons dans Reverse_DHT d'employer en plus les voisins *reverse* de la nouvelle table. Comme certains nœuds de la DHT présentent un grand nombre de liens entrants, la taille de leurs $reverse_{table}$ serait par conséquent importante. Ces nœuds particuliers auront une vision plus large du réseau et donc permettront un routage plus rapide.

Afin de mieux comprendre la technique de routage dans Reverse_DHT nous présentons dans ce qui suit notre approche appliquée à Chord, Pastry et Kademlia.

3.2.2 Application à Chord, Pastry et Kademlia

3.2.2.1 Application à Chord : Reverse_Chord

Dans Chord (33) chaque nœud maintient une liste de voisins appelés *fingers* et une liste de voisins séquentiels dont le prédécesseur et les successeurs. Dans Reverse_Chord, chaque nœud maintient en plus une table $reverse_{table}$ qui contient la liste des nœuds qui l'ont choisi comme *finger.* La maintenance de la nouvelle table se fait par le message *fix_finger* qui maintient dans Chord les liens vers les voisins *fingers.* A chaque nœud *reverse,* est associé un voisin séquentiel qui est son prédécesseur. La $reverse_{table}$ de Reverse_Chord contient donc les paramètres suivant : (*reverse, pred, time*)

Dans Reverse_Chord, la taille d'une ligne dans la table des nœuds *reverse* est de 48 octets. Pour un nœud avec 900 liens entrants (Figure 3. 1(a)) cela demanderait donc 42.18 ko d'espace mémoire en plus.

Sachant l'identifiant du nœud *reverse* et celui de son prédécesseur, un nœud connait l'espace des identifiants géré par ce dernier. Pour chaque entrée de sa $reverse_{table}$, un nœud connait donc la zone dont elle est responsable. Par exemple, si le nœud A possède dans sa table $reverse_{table}$ l'entrée $(reverse\ 10,\ pred\ 6,\ time\ 509)$, cela veut dire que le nœud *reverse* 40, dont le prédécesseur est le nœud 5, est responsable de la zone des identifiants $[5, 40[$. Le paramètre 509 indique le temps de la dernière mise à jour de l'entrée. Ces informations seront utilisées pour le routage.

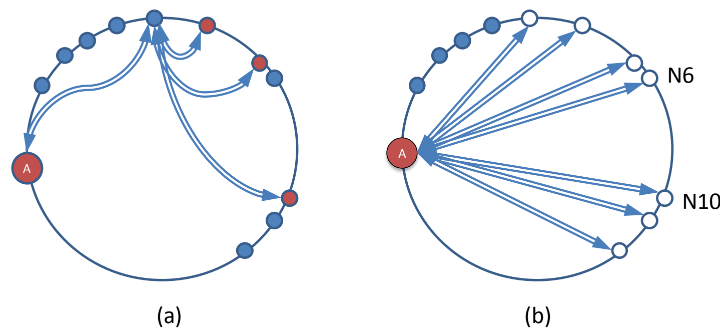


Figure 3. 4 (a) ajout des liens inverses depuis chaque finger, (b) nœuds reverse du point A (ronds blancs)

Nous étendons la procédure de routage Reverse_Chord comme suit : lorsqu'un nœud intermédiaire reçoit une requête de recherche, il examine d'abord sa table $reverse_{table}$ pour voir s'il trouve la destination finale. Pour cela, il compare la clé recherchée à l'identifiant du nœud *reverse* et de son prédécesseur.

- Si la clé est dans l'intervalle $[reverse, pred[$, la requête de recherche est transférée directement à la destination *reverse,* responsable de la clé recherchée (destination finale).
- Sinon, le nœud intermédiaire cherche dans sa table de *fingers* et dans sa nouvelle $reverse_{table}$ le voisin le plus proche de la clé recherchée. La requête est transférée dans ce cas au prochain nœud intermédiaire.

3.2.2.2 Application à Pastry : Reverse_Pastry

Dans la DHT de Pastry (34) classique, chaque nœud maintient une table de routage qui classe ses nœuds voisins selon leurs identifiants. Il maintient en plus une liste de voisins séquentiels

qui sont les nœuds feuilles. Nous proposons dans Reverse_Pastry d'utiliser la liste des nœuds feuilles pour remplir la $reverse_{table}$. Ainsi, pour chaque entrée de la nouvelle table, on enregistre des identifiants de son plus petit et plus grand nœud feuille (*smallest, biggest*). Cette information sera véhiculée dans le message *row_request,* utilisé pour la maintenance des liens vers les voisins de la table de routage de Pastry. Une entrée de la $reverse_{table}$ contient donc : (*reverse,* (*smallest, biggest*), *time*).

Dans Reverse_Pastry*,* la taille d'une ligne dans la table des nœuds *reverse* est donc de 68 octets.

Nous proposons l'extension suivante pour le routage Reverse_Pastry : lorsqu'un nœud reçoit une requête de recherche, il examine sa nouvelle table pour vérifier si la destination est dans l'intervalle]*smallest, biggest*] d'une des entrées de la $reverse_{table}$.

- Si la clé est dans cet intervalle, la requête est envoyée directement au nœud reverse correspondant puisque la destination se trouve parmi ses feuilles.
- Sinon, comme pour Reverse_Chord, le nœud cherche dans sa table de routage et de *reverse*, le voisin le plus proche de la clé selon l'algorithme de Pastry.

3.2.2.3 Application à Kademlia : Reverse_Kad

Nous avons vu que la distribution des degrés dans le réseau Kademlia (k=8) varie peu et que le graphe Kademlia (k=8) est naturellement bidirectionnel. Ainsi, l'extension Reverse_Kad que nous proposons ne rajouterait à Kademlia que l'information sur les voisins séquentiels et pas le routage bidirectionnel.

Nous proposons d'employer la même approche S/Kademlia en sélectionnant comme voisins séquentiels, les nœuds *siblings,* les s nœuds les plus proches parmi les *buckets.*

Nous proposons également de transporter l'information sur les voisins séquentiels dans les messages de la DHT classique. Contrairement à Chord et Pastry, dans Kademlia, il n'y a pas une procédure de maintenance proprement dit. Ce dernier utilise les messages *find_node* qu'il intercepte, pour la mise à jour de sa table de routage. Nous proposons d'utiliser ce même message dans Reverse_Kad pour remplir la nouvelle $reverse_{table}$. Ainsi, dans Reverse_Kad, nous étendons le message *find_node* avec la liste des identifiants des voisins séquentiels. Chaque fois qu'un nœud reçoit un message *find_node*, il rajoute à sa table le nouveau *reverse*

(l'expéditeur du message), ainsi que ses nœuds *siblings.* La $reverse_{table}$ sera de la forme : (*reverse,* (*sibling1,.., sibling s*), *time*)

Nous fixons le nombre de *siblings* à 4 conformément à S/Kademlia (51). Dans ce cas, la taille d'une ligne dans la nouvelle table est de 108 octets.

Comme Kademlia maintient déjà les liens inverses, la $reverse_{table}$ peut être directement intégrée à la table de routage de Kademlia. Pour ce faire, en plus de rajouter l'expéditeur à sa table de routage, un nœud Reverse_Kad insère aussi ses voisins *siblings.* Nous choisissons cependant de maintenir deux tables séparées pour des raisons que nous expliquerons par la suite.

La procédure de routage Reverse_Kad se fait comme suit : lors du routage, une requête de recherche dans la DHT de Kademlia est envoyée parallèlement aux k nœuds les plus proches de la clé dans la table de routage (35). Nous proposons dans Reverse_Kad d'employer en plus la nouvelle $reverse_{table}$ dans la procédure de routage. Ainsi, quand un nœud reçoit un message *find_node*,

- il vérifie d'abord si la destination existe parmi les nœuds *reverse* et leurs *siblings,* dans ce cas, la requête est envoyée au nœud *reverse* et ses *siblings* (dans la limite de k nœuds).
- sinon, il cherche dans sa table de routage ainsi que dans sa $reverse_{table}$, les k nœuds les plus proches suivant la métrique XOR comme dans la DHT de Kademlia classique.

3.2.2.4 Conclusion

L'application de l'approche Reverse_DHT diffère selon la méthode de routage et de maintenance de l'algorithme. Cependant, le principe reste toujours le même. L'idée est d'utiliser les liens entrants pour élargir le voisinage notamment pour les nœuds fortement connectés. Ces liens sont simples à maintenir et ne requièrent aucune nouvelle signalisation. La procédure de routage est légèrement modifiée pour intégrer la nouvelle structure. Cette dernière permet de raccourcir les chemins de recherche. En effet, l'information sur les voisins séquentiels rajoutée à la $reverse_{table}$ permet de localiser directement la clé dans les zones gérées par les nœuds *reverse.* Cette information est véhiculée dans les messages de maintenances existants, puis sauvegardée dynamiquement dans la nouvelle $reverse_{table}$. Le coût de signalisation en terme de nombre de messages reste donc le même. Par contre, le coût en termes de tailles en octets augmente, vu l'extension des messages *refresh.* Le coût en mémoire

augmente aussi, vu l'addition de la nouvelle structure $reverse_{table}$. La taille de cette table varie selon la connectivité des nœuds. Vu que la majorité des nœuds possèdent un degré entrant faible, le coût en mémoire n'augmentera que pour les quelques nœuds fortement connectés.

3.2.3 Evaluation de l'approche Reverse_DHT

Nous avons implémenté notre approche de routage Reverse_DHT avec le simulateur OverSim/Omnet++ (110). Cette implémentation a été faite sur une machine linux $IntelQX6700$ avec $4G$ de RAM.

Oversim (111) (112) est un logiciel libre pour la simulation des réseaux pair-à-pair basé sur Omnet++ et développé à l'Université de Karlsruhe en Allemagne. Nous avons choisi ce simulateur vue qu'il intègre plusieurs modèles pair-à-pair structurés et non structurés et qu'il offre une documentation abondante et pertinente.

Pour l'implémentation de l'approche Reverse_DHT, nous avons intégré à Chord, Pastry et Kademlia un module supplémentaire pour la construction et maintenance de la nouvelle table. Nous avons également changé la procédure de routage pour intégrer la table $reverse_{table}$.

Dans toute la suite du document et pour tous les résultats présentés, nous considérons un intervalle de confiance à 95%. Pour chacune des valeurs résultats, nous calculons donc l'intervalle (113) :

$$\left[E(x) - 1.96\frac{\sigma(x)}{\sqrt{S}};\ E(x) + 1.96\frac{\sigma(x)}{\sqrt{S}}\right]$$

$E(x)$ est la moyenne de la variable x à estimer, $\sigma(x)$ est l'écart type de x et S le nombre d'échantillon de la série des mesures.

D'après nos calcul, les intervalles obtenus sont très petits (de l'ordre de 10^{-3}) vu le nombre très important d'échantillons considérés (temps de simulations très longs). Nous choisissons donc de présenter dans les figures et les résultats les valeurs moyennes seulement.

3.2.3.1 Topologies, scenarii de trafic et critères de performances

Nous évaluons, dans un premier temps, l'approche Reverse_DHT appliquée à la DHT de Chord. Les résultats de Pastry et Kademlia seront discutés par la suite.

Nous comparons notre modèle Reverse_Chord à la DHT de Chord classique, à BiChord (114) (section 2.3.2). A la différence de Reverse_Chord, BiChord utilise seulement le routage KBR et emploie ainsi les nœuds *reverse* simplement comme des *fingers.* Reverse_Chord, en

revanche, garde en plus dans sa nouvelle table la liste des voisins séquentiels du nœud *reverse,* lui permettant ainsi de connaitre leurs zones de responsabilités et de localiser directement la destination finale de l'objet à partir d'un nœud intermédiaire.

Nous évaluons notre approche faisant varier deux paramètres : la taille du réseau et le *churn.* Dans un premier temps, nous simulons un réseau de taille variable de 512 à 16348 nœuds statiques. Le nombre de successeurs est fixé à 8 et le nombre de voisins *fingers* est fixé à 8. Une fois le réseau stable, les nœuds envoient toutes les 60 secondes un message vers une clé choisie aléatoirement suivant une distribution uniforme.

Nous faisons varier ensuite le paramètre *churn.* Pour cela, nous simulons un réseau de 8192 nœuds avec une durée de vie suivant une loi de Poisson de moyenne variable de 2000 à 10000 secondes.

Nous évaluerons Reverse_Chord, BiChord et Chord tenant compte des paramètres suivants : le taux de succès des requêtes, la longueur de la route KBR et la charge du réseau (115) (62). Pour mesurer la charge du réseau en termes de taille des messages de signalisation nous considérons deux paramètres : la taille totale de tous les messages et la charge générée par les messages *fix_finger* (utilisés pour la maintenance des deux tables de routage).

3.2.3.2 Résultats

La Figure 3. 5 présente les résultats des simulations des approches Reverse_Chord, Bichord et Chord.

La Figure 3. 5(c) calcule le degré entrant moyen dans Chord qui représente aussi la taille moyenne de la $reverse_{table}$ dans Reverse_Chord. Nous remarquons que le degré moyen augmente logarithmiquement en fonction du nombre de nœuds. La taille moyenne de la $reverse_{table}$ s'exprime donc en $O(\log N)$. Ceci nécessite en moyenne $48 \log N$ octets pour le stockage.

Nous remarquons que notre approche Reverse_Chord améliore les performances de la DHT. Elle produit en effet les plus courtes routes (Figure 3. 5 (a)) et la plus petite charge de signalisation en termes de nombre de messages total (Figure 3. 5 (b)). La procédure de routage par les nœuds *reverse* permet de raccourcir les chemins de recherche puisque le routage se fait dans les deux sens, et traverse potentiellement des nœuds avec une connaissance plus large du réseau.

Concernant, la charge du réseau, nous remarquons que Reverse_Chord produit le nombre de messages de signalisation le plus petit, notamment dans les réseaux larges. Puisque les chemins de recherche sont plus courts dans Reverse_Chord, le nombre de messages de recherche transmis est moins important. Par conséquent, la charge totale en termes de nombre de messages est moins importante.

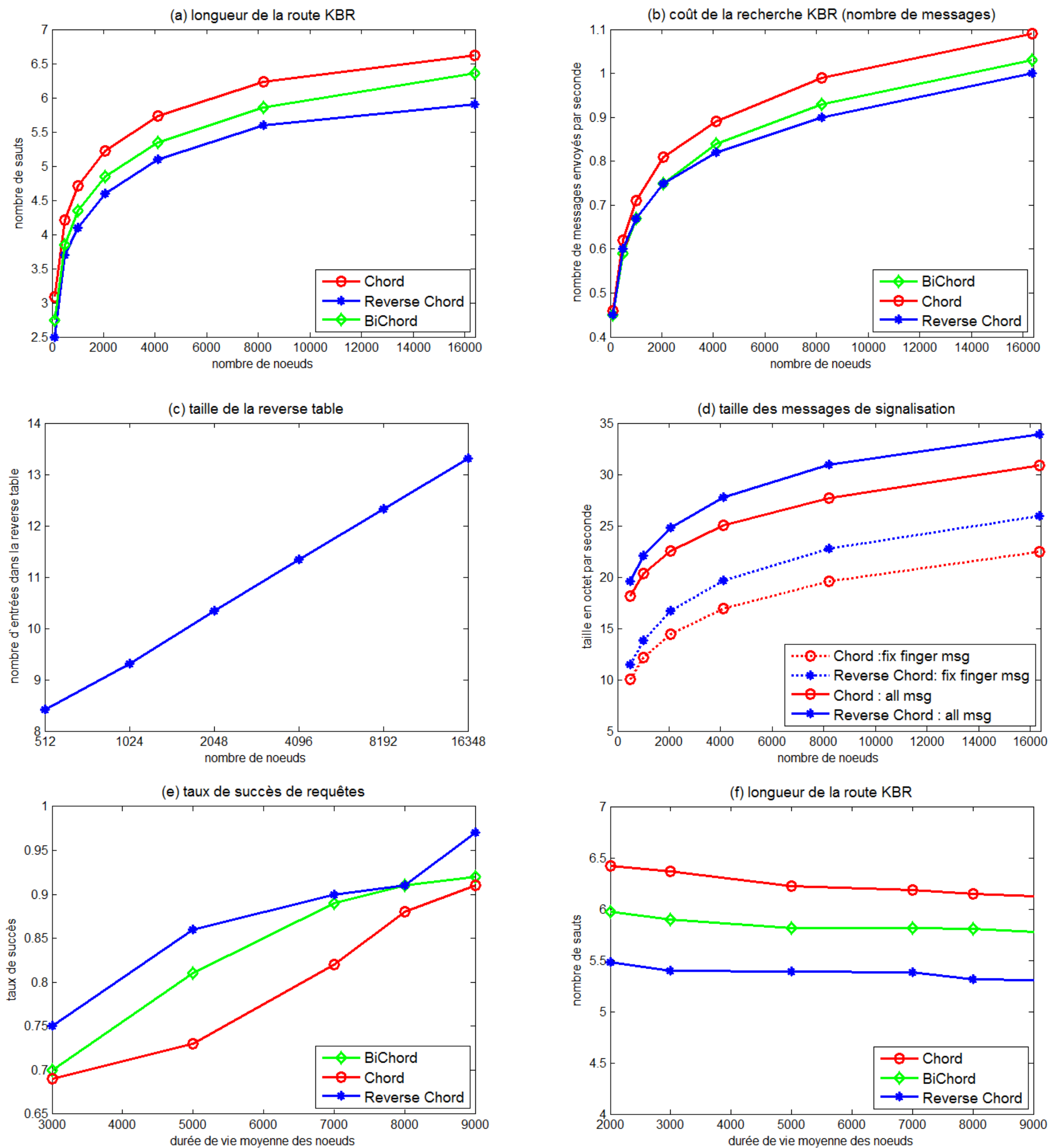


Figure 3. 5 Résultats de Reverse Chord

Pour évaluer la charge du réseau en termes de taille des messages, nous avons mesuré la taille moyenne en octets des messages de maintenance émis, faisant varier la taille du réseau. Pour cela, nous calculons le nombre d'octets moyens générés par tous les messages de maintenance de Chord *(join, stabilize, fix_finger…)* et le nombre d'octets moyens des messages *fix_finger.* La (Figure 3. 5 (d)) illustre le résultat.

Nous remarquons que la taille des messages de maintenance augmente pour Reverse_Chord (3 octets par seconde). Ceci est dû à l'extension du message *fix_finger* avec le nouveau champ seq_{neigh}. En effet, la différence de taille des messages de maintenance entre les deux approches est égale à la différence de taille des messages *fix_finger*. Nous n'avons pas inclus dans la Figure 3. 5(d) les résultats de BiChord pour ne pas encombrer l'image. Cependant, BiChord se comporte globalement comme Chord puisqu'il utilise exactement les mêmes messages.

Nous faisons varier maintenant la durée de vie des nœuds pour modéliser le *churn*. Nous évaluons le taux de succès des requêtes de recherche et la longueur des routes résultantes. Les Figure 3. 5 (e) (f) illustrent les résultats. Nous remarquons que notre approche Reverse_Chord affiche un taux de succès légèrement meilleur, dépassant les 75% pour toutes les valeurs de *churn*. Pour Bi_Chord et Chord ces valeurs chutent au dessous de 70%. Notre approche produit aussi les plus courts chemins. Le *churn* affecte moins les performances de Reverse_Chord comparé à Chord et BiChord, vu que les nœuds possèdent un voisinage plus grand et donc plus d'alternatives pour le choix du prochain saut. Aussi, vu que les routes dans Reverse_Chord sont plus courtes, le nombre de nœuds traversés lors de l'acheminement des messages est moins important, et donc moins de nœuds sont concernés par le routage.

3.2.4 Conclusion

L'observation de la topologie de différentes DHTs a montré une intéressante hétérogénéité dans la distribution des degrés entrants. Les DHTs classiques utilisent toutefois un nombre prédéfini de nœuds pour le routage. A partir de cette observation, nous avons eu l'idée de rendre le graphe de la DHT bidirectionnel afin d'exploiter cette hétérogénéité. Pour ce faire, une nouvelle structure appelée $reverse_{table}$ a été ajoutée et la procédure de routage des messages a été légèrement modifiée en conséquence.

L'évaluation de l'approche Reverse_DHT a montré l'avantage du routage bidirectionnel dans les DHTs. Le gain total n'est certes pas énorme, mais cela ne coûte rien non plus. Cette solution produit des routes plus courtes (12.7% de sauts en moins en moyenne), une meilleure résistance face au *churn* (12.6% de requêtes abouties en plus) et surtout n'induit pas de signalisation supplémentaire. L'extension des messages de maintenances par un nouveau champ, augmente peu la taille de ces messages en moyenne (4 octets). En revanche, le nombre des messages véhiculés dans le réseau est réduit (9% de messages en moins en moyenne).

Les résultats intéressants de la technique de routage Reverse_DHT nous ont amenés à pousser encore plus loin notre idée. Pour exploiter davantage cette hétérogénéité, nous proposons de ré-interconnecter le graphe de la DHT.

Notre idée est d'amplifier cette hétérogénéité en transformant le graphe de la DHT vers une structure DHT décentralisée non égalitaire. Pour ce faire, nous proposons Power_DHT, une nouvelle structure d'interconnexion et de routage, qui fait évoluer dynamiquement la DHT vers un réseau décentralisé, tout en conservant la structure originale de la DHT (l'anneau dans Chord par exemple).

3.3 2ème étape : Power_DHT, approche de routage 'Power Law'

Power_DHT est une nouvelle méthode d'interconnexion et de routage qui amplifie les propriétés observées dans les DHTs. En changeant dynamiquement les paramètres de notre algorithme, nous montrons comment obtenir des structures DHTs décentralisées ayant naturellement le diamètre minimal. La structure Power_DHT résultante permet de combiner les fonctionnalités des systèmes structurés et non structurés. D'un côté les systèmes structurés employés principalement pour les recherches exactes. De l'autre, les systèmes non structurés pour les recherches floues.

Jusque là, les seuls systèmes combinant les deux architectures proposaient une approche hybride ayant une architecture hiérarchique, généralement sur deux niveaux. Les super nœuds assurent les fonctions de recherche et de routage et gèrent les pairs qui y sont rattachés. Cependant, le problème qui se pose ici est la sélection et la maintenance des super nœuds (section 2.3.1)

A l'inverse des approches hybrides, Power_DHT organise ses nœuds suivant une structure décentralisée 'non hiérarchique'. En ré-interconnectant le graphe de la DHT, le réseau converge successivement vers une structure décentralisée. Par ailleurs, il n'existe pas de procédure de gestion des super nœuds à proprement parler. Cependant, cette 'élection' se fait 'naturellement', émanant de l'architecture résultante dans les réseaux déployés.

3.3.1 Modèle Barabási Albert pour la génération des graphes en loi de puissance

Nous avons vu dans la section 3.2.3 que la technique de routage par les nœuds *reverse* améliore les performances sans produire un coût supplémentaire. Seulement, dans Reverse_DHT, les nœuds fortement connectés ne sont sollicités pour le routage que lorsqu'une requête de recherche transite par eux (selon le routage de DHT classique). En effet, les routes dans les DHTs sont calculées selon un algorithme précis (section 2.2). Le choix de la route et des nœuds à traverser dépend, par conséquent, de l'algorithme de routage employé.

Or, notre but de départ était d'exploiter au mieux cette hétérogénéité. Chaque nœud de la DHT doit sélectionner pour le routage, les voisins avec la meilleure connaissance du réseau. Pour ce faire, notre idée est de modifier la procédure de sélection de voisins, de manière à ce que chaque nœud se connecte aux voisins ayant le plus fort degré. D'où l'idée de transformer la structure en un graphe sans échelle en s'inspirant du modèle de Barabási Albert (BA) (116).

Graphe sans échelle

Un réseau *sans échelle* (en anglais *scale free network*) est un réseau dont la distribution des degrés suit une loi de puissance de la forme $P(k) = k^{-\alpha}$, avec $2< \alpha <3$. Cette distribution fortement hétérogène est caractérisée par la présence d'un nombre limité de nœuds appelés *hubs* possédant un fort degré de connectivité, tandis que la majorité des nœuds présente un degré faible.

Le caractère sans échelle a été observé dans de nombreux réseaux complexes tels que l'Internet, WWW, les réseaux cellulaires ou encore les réseaux sociaux (117) (118) (119) (116). Dans ces réseaux, les nœuds ne se concentrent qu'autour de quelques *hubs* convergeant 'naturellement' vers une architecture hétérogène. Cette topologie offre des performances intéressantes pour le routage, à savoir une faible distance moyenne et une résistance aux pannes aléatoires (120).

Ces réseaux sont aussi dits *Ultra Small* vu qu'ils offrent une distance moyenne généralement inférieure à $\log N$, la moyenne observée dans les réseaux aléatoires ou les petits mondes (121). La distance moyenne d séparant deux nœuds d'un graphe sans échelle de taille N dépend du paramètre α (122):

- $\alpha=3,\ d\sim\left(\frac{\log N}{\log \log N}\right)$
- $2<\alpha<3,\ d\sim(\log \log N)$
- $\alpha>3,\ d\sim\log(N)$

Par ailleurs, les réseaux en loi de puissance sont particulièrement adaptés à la recherche par marche aléatoire privilégiant le voisin de plus fort degré (123). Les auteurs de (124) ont montré que la longueur moyenne des routes est de $N^{3(1-2/\alpha)}$ pour la marche privilégiant le voisin avec le plus fort degré contre $N^{2-4/\alpha}$ pour la marche aléatoire classique (N étant la taille du réseau en nombre de nœuds).

De ce fait, le modèle sans échelle est désormais utilisé pour la modélisation des réseaux réels, modélisés auparavant par des graphes aléatoires ou les petits mondes. En effet, l'utilisation de la distribution des degrés en loi de puissance dans les réseaux réels a été proposée dans (125) (126) (127). Dans ces travaux, les auteurs modélisent les distributions hétérogènes des degrés par des lois de puissance. De plus, les observations de (128) (129) ont montré que les réseaux pair-à-pair présentent une distribution des degrés en loi de puissance.

Les graphes sans échelle semblent donc une alternative intéressante pour la construction d'un réseau P2P structuré potentiellement hétérogène. Pour ce faire, nous choisissons le modèle d'attachement préférentiel proposé par Barabási Albert (116).

Modèle Barabási Albert

Le modèle le plus connu pour la génération de graphe sans échelle est celui proposé par Barabási Albert (BA). Dans ce modèle, les nœuds sont rajoutés un à un puis reliés à m nœuds existants. L'interconnexion entre les nœuds se fait comme suit : les voisins auxquels le nouveau nœud est relié sont choisis avec une probabilité qui croît avec leur degré. En d'autres termes, un nouveau nœud se connecte à m nœuds existant avec une probabilité p_i tel que :

$$p_i = k_i / \sum_j k_j$$

k_i étant le degré du nœud i.

3.3.2 Architecture Power_DHT

La technique d'attachement préférentiel proposé par le modèle Barabási Albert suppose que chaque nœud du réseau connait le degré de tous les nœuds déjà présents. Ceci suppose donc une vision globale du réseau. Or, dans le cas d'une structure distribuée, un nœud possède uniquement une vision partielle. Dans le cas des DHTs, un nœud connait un ensemble de nœuds voisins qu'il maintient régulièrement. De ce fait, l'application du modèle BA n'est pas directement adaptée à de telles structures. Pour rendre cette solution possible dans un environnement distribué, nous reprenons et modifions la technique d'attachement préférentielle en conséquence.

3.3.2.1 Construction et maintenance du graphe Power_DHT

Afin de construire un graphe sans échelle à partir de la DHT, nous modifions la procédure de sélection des voisins pour tenir compte de l'attachement préférentiel. Au lieu de choisir le voisin v selon l'identifiant calculé, le nœud choisit de se connecter à un des voisins séquentiels seq_i de v, avec une probabilité proportionnelle à son degré. En d'autres termes, on choisit le nœud dans le voisinage du voisin calculé, avec le plus fort degré localement.

A la différence de Reverse_DHT*,* les nœuds dans Power_DHT se connectent vers des voisins choisis localement selon leur degré de connectivité. Le fait de choisir localement les voisins permet d'adapter la technique d'attachement préférentielle à la nature distribuée de la DHT, et en plus de conserver la structure et donc les propriétés de la DHT originale.

Power_DHT est une extension de la structure Reverse_DHT. Elle intègre donc nos propositions dans Reverse_DHT mais change en plus la sélection des voisins et emploie la recherche aléatoire. Ainsi, les procédures de routage KBR et de maintenance des nœuds *reverse* restent invariées.

3.3.2.2 Routage KBR dans Power_DHT

Le routage par clé KBR s'effectue comme dans Reverse_DHT. A la réception d'une requête de recherche, le nœud examine sa $reverse_{table}$ pour rechercher la destination. A défaut, il cherche dans sa table de routage et sa $reverse_{table}$ le nœud le plus proche de la clé recherchée selon la métrique utilisée.

Le graphe Power_DHT résulte en une structure décentralisée non égalitaire, où un nombre limité de nœuds *hubs* sont fortement connectés. Ces *hubs* sont choisis comme principaux voisins pour remplir les tables de routage. Par conséquent, les requêtes de recherche seront essentiellement relayées ces points *hubs*. A chaque saut intermédiaire, l'espace de recherche est élargie, et le nombre de sauts est ainsi réduit.

3.3.2.3 Routage aléatoire dans Power_DHT

Actuellement, les systèmes pair-à-pair laissent le choix entre connecter les nœuds sans maintenir la structure, ce qui est simple mais nécessite l'inondation lors des recherches, ou connecter les nœuds de manière structurée ce qui permet de router les recherches exactes mais rend difficile les recherches floues.

La nature décentralisée du graphe Power_DHT motive l'emploi des méthodes de routage aléatoire dans notre nouvelle structure. De ce fait, nous proposons de tester le routage par inondation et le routage par marche aléatoire privilégiant le nœud de plus fort degré.

Le routage par inondation dans Power_DHT se fait comme suit : à la réception d'une requête de recherche, chaque nœud renvoie la requête à tous ses voisins présents dans sa table de routage et dans sa $reverse_{table}$ (sauf le nœud expéditeur). Un nœud intermédiaire renvoie à son tour la même requête à ses voisins s'il ne possède pas l'objet recherché. L'inondation se poursuit récursivement jusqu'à l'expiration du TTL.

La marche aléatoire par le voisin de plus fort degré dans Power_DHT se fait comme suit : la requête de recherche parcourt le graphe en profondeur choisissant à chaque saut le nœud avec le degré le plus élevé. La recherche s'arrête dès la localisation de l'objet ou l'expiration du

TTL. Nous expliquons ces deux stratégies de routage plus en détail appliquées à Chord et à Pastry.

3.3.2.4 Conclusion

En adaptant le modèle d'Albert Barabási à la nature distribuée des DHTs, Power_DHT fait évoluer le graphe de la DHT vers une structure non égalitaire similaire à un graphe sans échelle. Partant de la structure de la DHT classique, notre structure Power_DHT construit dynamiquement un graphe décentralisé qui accroît l'hétérogénéité observée dans le réseau initial.

Nous proposons d'employer cette nouvelle structure pour le routage KBR et le routage aléatoire. Nous verrons plus loin que cette nouvelle distribution est suffisante pour que notre stratégie de routage par inondation soit efficace.

Afin de mieux comprendre l'architecture Power_DHT et les modèles de routage, nous détaillons dans ce qui suit l'approche proposée appliquée aux algorithmes Chord, Pastry et Kademlia. Pour cela, nous identifions des paramètres applicables à toutes les DHTs ainsi que des idées spécifiques à chacune.

3.3.3 Application à Chord, Pastry et Kademlia

3.3.3.1 Power_Chord

Dans Power_Chord, la sélection des voisins (*fingers*) est modifiée comme suit : lorsqu'un nouveau nœud ID cherche à se connecter à un *finger* dont l'identifiant est supérieur à $\text{ID} + 2^{i-1}$, il commence par lui envoyer un message *fix_finger.* A la réception de ce message, le *finger* en question sélectionne parmi ses successeurs dans l'intervalle $[\text{ID}+ 2^{i-1}, \text{ID} + 2^{i}[$ le nœud successeur s suivant la probabilité p_i. Le nœud s sera sélectionné comme *finger*$_i$ du nœud ID et donc inséré dans la table des *fingers.*

Un nœud Power_Chord maintient aussi une table de nœuds *reverse,* utilisée pour le routage comme pour Reverse_Chord.

3.3.3.2 Power_Pastry

Nous appliquons maintenant cette même approche mais adaptée à la DHT de Pastry. Lorsqu'un nœud A cherche un voisin v, il lui envoie un message *row_request.* A la réception de ce message, v sélectionne parmi ses voisins séquentiels le nœud feuille i suivant la

probabilité p_i. Ce nœud feuille sera par la suite inséré dans la table de routage du nœud A. Pareillement à Reverse_Pastry, Power_Pastry maintient une table de *reverse* qu'il emploie pour le routage.

3.3.3.3 Power_Kad

L'application de Power_DHT à Kademlia est légèrement différente vu qu'il n'existe pas de procédure de maintenance par défaut. Au cours du routage, lorsqu'un nœud intercepte un message *find_node*, avec l'adresse de l'expéditeur (*reverse*) et ses voisins *siblings*, il choisit parmi ces voisins séquentiels le noeud *sibling* suivant la probabilité p_i, et l'insère dans sa table de routage dans le *bucket* correspondant. Si ce *bucket* est rempli, il efface l'entrée avec le degré minimum et rajoute le nouveau voisin sélectionné.

La maintenance de la table des *reverse* et la procédure de routage s'effectuent comme pour Reverse_Kad.

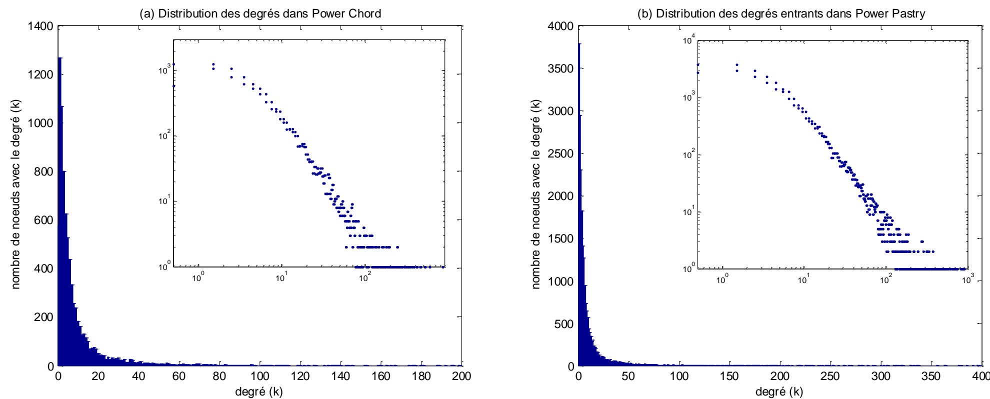


Figure 3. 6 Distribution des degrés dans Power_Pastry et Power_Chord

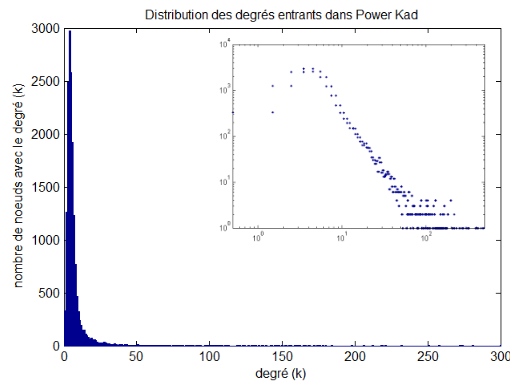


Figure 3. 7 Distribution des degrés dans Power_Kad

Les Figure 3. 6 et Figure 3. 7 présentent la distribution des degrés pour les trois structures. Nous représentons la distribution en échelle scalaire et logarithmique (log_2) dans un réseau

de 20000 nœuds. Nous observons que la distribution résultante est une distribution proche d'une loi en puissance. Dans Power_Chord par exemple, le degré maximal atteint 800 nœuds, mais seul 20% des nœuds possèdent un degré supérieur à 12. Le degré moyen, bien que peu significatif sur ce type de distribution est de 11.12.

La Figure 3. 8 représente l'interconnexion entre les nœuds de l'anneau Power_Chord dans un réseau de 200 nœuds. Nous observons la présence de quelques points *hubs* où se concentrent les liens. Dans ce schéma, nous disposons les nœuds à intervalles réguliers, indépendamment de leurs identifiants.

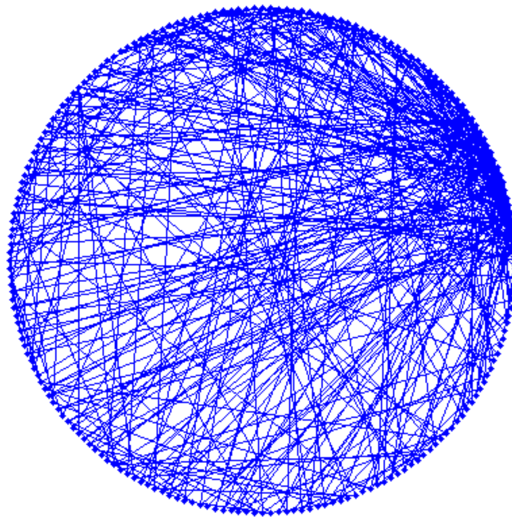


Figure 3. 8 Interconnexion entre les nœuds dans Power_Chord (N=200)

3.3.3.4 Conclusion

L'application de Power_DHT aux différentes structures suit le même principe. Se basant sur l'information sur les voisins séquentiels, le nœud sélectionne le voisin selon la probabilité p_i. En sélectionnant un des voisins séquentiels, Power_DHT préserve la structure de la DHT tout en 'décentralisant' le système. En effet, la structure originale de la DHT est conservée vu que l'on choisit de se connecter à des nœuds proches numériquement dans l'espace des identités, et qui vérifient les critères de choix le la DHT par défaut. De plus, le graphe sans échelle résultant permet de décentraliser la structure auparavant égalitaire.

Dans Power_Chord par exemple, le nouveau voisin du nœud est choisi dans l'intervalle $[\mathrm{ID} + 2^{i-1}, \mathrm{ID} + 2^{i}[$. Power_Pastry choisit un nœud feuille du voisin calculé qui partage le même préfixe. Enfin dans Kademlia, le nœud choisit comme voisin le *sibling* situé à une distance XOR dans $[2^i, 2^{i+1}]$.

Cette flexibilité dans le choix du voisinage pour les trois DHTs permet donc à Power_DHT de conserver la structure originale de la DHT, et de garantir ainsi le diamètre de routage en $O(\log N)$ dans le pire des cas. En effet, la structure originale de la DHT est conservée vu que l'on choisit de se connecter à des nœuds numériquement proches dans l'espace des identités.

Enfin, l'information sur les voisins séquentiels n'est pas toujours disponible dans toutes les DHTs. Toutefois, elle peut être intégrée facilement dans plusieurs structures comme nous avons vu pour Kademlia (Viceroy, CAN (62)). Power_DHT peut donc être facilement adaptable à d'autre structure DHTs.

3.3.4 Evaluation de l'approche Power_DHT

3.3.4.1 Topologies, scenarii de trafic et critères de performances

Nous évaluons, dans un premier temps, notre approche appliquée à la DHT de Chord. Nous comparons notre modèle Power_Chord à la DHT de Chord classique et à Balanced_Chord, (que nous décrivons dans la suite). Le but étant d'évaluer et de comparer notre proposition Power_DHT face à une approche complètement opposée.

Description de l'approche Balanced_Chord

Se basant sur l'architecture e_Chord (103) (section 2.3.6), nous proposons Balanced_Chord, une approche qui emploie le routage bidirectionnel Reverse_Chord mais dans le graphe équilibré e_Chord. Afin de mieux comprendre cette architecture, nous expliquons la procédure de sélection de *finger* proposé dans e_Chord.

Dans e_Chord, lorsqu'un nœud cherche à se connecter à un *finger,* il lui envoie un message *fix_finger.* A la réception de ce message, le nœud sélectionne parmi ses successeurs un nœud aléatoire s_i. Le nœud s_i sera sélectionné comme *finger* et inséré dans la table des *fingers.* Chaque nœud e_Chord partage ainsi ses liens avec ses successeurs, ce qui résulte en une distribution de degré équilibré, comme présentée dans la Figure 3. 9. A l'inverse de notre approche Power_DHT, le but dans e_Chord est d'uniformiser la distribution des degrés.

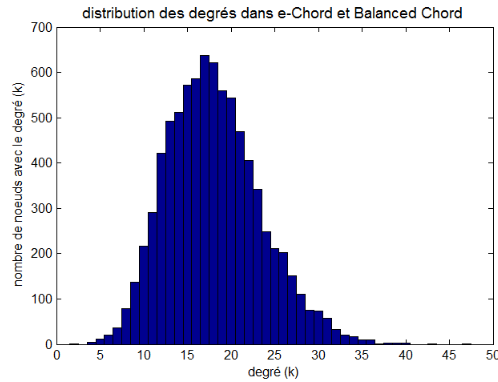


Figure 3. 9 Distribution des degrés entrants dans e-Chord

Nous proposons donc de comparer Power_DHT à Balanced_Chord, une technique qui emploie la procédure de sélection des voisins de e_Chord et la procédure de routage bidirectionnel de Reverse_Chord (emploie la $reverse_{table}$ pour le routage).

Nous comparons Power_Chord, Balanced_Chord et Chord faisant varier deux paramètres : la taille du réseau et le *churn.* Nous utilisons les mêmes paramètres de simulations et les mêmes critères de performances de l'évaluation précédente.

3.3.4.2 Résultats

Résultats de Power_Chord

La Figure 3.10 présente les résultats des simulations des approches Power_Chord, Balanced_Chord et Chord faisant varier la taille du réseau. Pour ne pas encombrer les figures, nous n'avons pas rajouté les résultats de Reverse_Chord. Toutefois, il est clair que l'approche Power_DHT produit des meilleures performances : nous comparons les techniques Power_DHT et Reverse_DHT dans la suite, avec les DHTs de Pastry et Kademlia.

Les résultats montrent que l'approche Power_Chord produit les routes les plus courtes. Dans le routage KBR (Figure 3.10 (a)), notre approche diminue de 19.7% le nombre de sauts pour le réseau le plus large. La nouvelle architecture emploie essentiellement les nœuds fortement connectés pour le routage, les requêtes de recherche traversent donc les nœuds avec une vision nettement plus large du réseau, ce qui accélère le routage.

Un deuxième résultat intéressant concerne la longueur des routes pour les recherches aléatoires. Pour la recherche par inondation, nous remarquons que la longueur de la route résultante est de l'ordre de $\log \log \mathrm{N}$ (Figure 3.10 (e)), la distance moyenne observée dans les réseaux sans échelle (section 3.3.1). La marche sélective privilégiant le nœud avec le plus fort degré génère, quant à elle, des routes d'environ $N^{2-4/\alpha}$, avec $\alpha \sim 2.2$ la valeur approximée

dans le graphe Power_Chord. Les routes de la recherche par inondation sont nettement plus courtes, permettant des recherches complexes rapides. Cependant, la surcharge générée est plus importante (Figure 3.10 (f)). Ces résultats confirment donc la nature sans échelle du graphe Power_DHT appliqué à Chord. Nous verrons dans la suite si Power_Pastry et Power_Kad confirment ces mêmes résultats.

Concernant, le coût de la recherche KBR, nous remarquons que Power_Chord génère moins de messages, notamment dans les réseaux larges (Figure 3.10 (b)). Ceci est une conséquence directe de la diminution du nombre de sauts nécessaires pour le routage KBR.

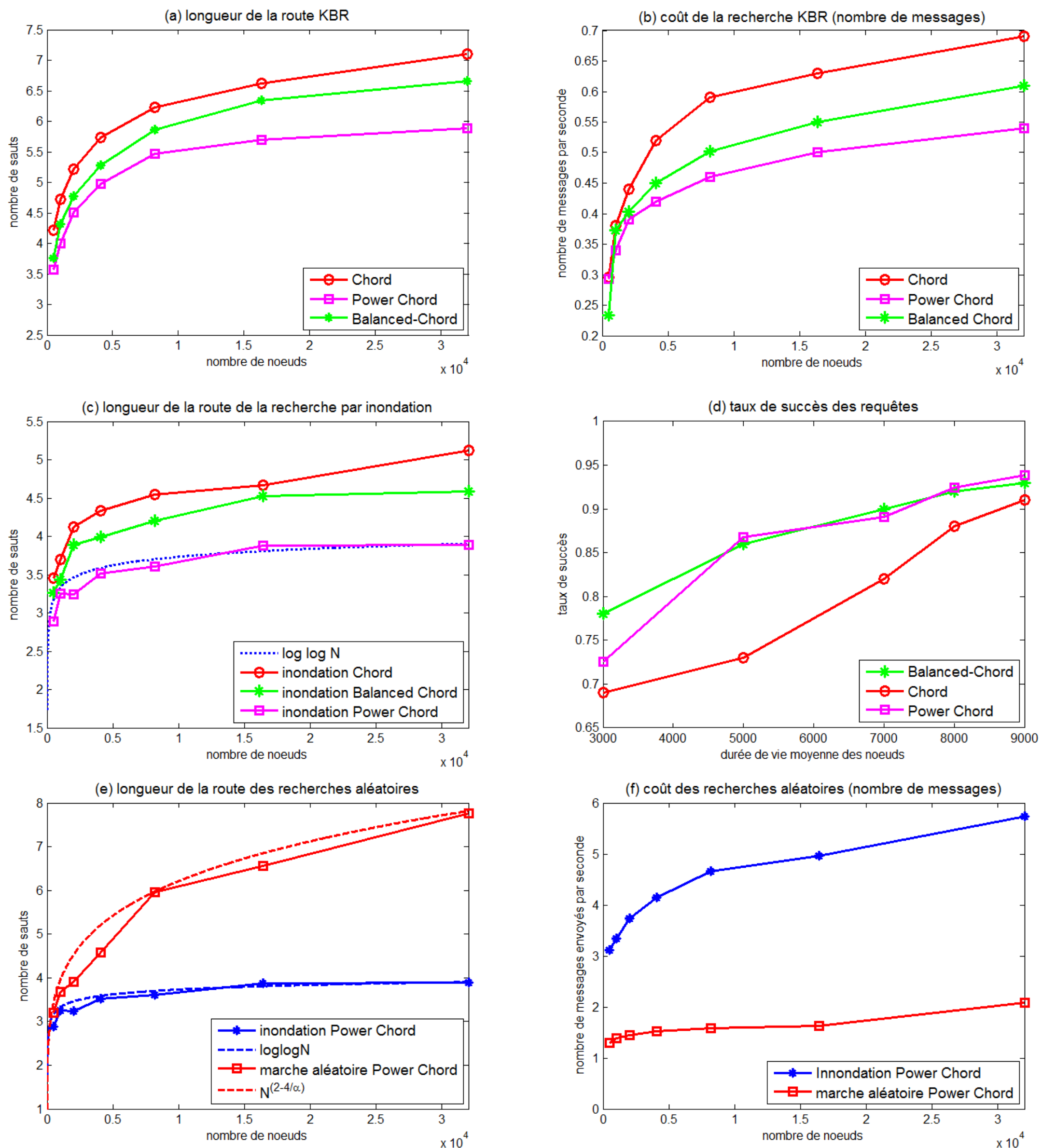


Figure 3.10 Résultats Power Chord

Nous faisons varier maintenant la durée de vie des nœuds pour modéliser le *churn* et nous évaluons le taux de succès des requêtes de recherche KBR (Figure 3.10 d). Pour un *churn*

moyen, Power_Chord et Balanced_Chord affichent un taux de succès similaire dépassant les 85%. Cependant, pour un *churn* élevé (durée de vie courte), les performances de Power_Chord baissent légèrement enregistrant un taux de succès inférieur à 75%, mais restent toujours supérieures à celles de Chord. Balanced_Chord produit un résultat légèrement meilleur avec un taux de succès dépassant les 80% pour toutes les valeurs de *churn*. Le *churn* élevé affecte plus la structure fortement hétérogène de Power_Chord que le graphe équilibré de Balanced_Chord. Les points de concentration crées dans Power_Chord sont plus fragiles face au *churn* ce qui explique le résultat. Cependant, il faut noter que les performances de notre approche restent toujours supérieures à celle de la DHT de Chord classique. Nous verrons dans le chapitre suivant comment améliorer la résistance du réseau au *churn* moyennant la réplication.

Résultats de Power_Pastry

La Figure 3. 11 présente les résultats des simulations de Power_Pastry. Afin de comparer les approches Reverse_DHT et Power_DHT, nous évaluons ici la DHT classique Pastry par rapport à Reverse_Pastry et Power_Pastry avec les mêmes paramètres décrits plus haut.

Nous évaluons d'abord le routage KBR. D'après la Figure 3. 11 (a), Power_Pastry réalise 23% de sauts en moins pour le réseau le plus large. Pareillement à Power_Chord, Power_Pastry produit une signalisation moindre qui résulte aussi de la diminution du nombre de sauts nécessaires pour le routage KBR (d'après la Figure 3. 11 (b), Power_Pastry réalise 23% de messages en moins pour le réseau de 16348 nœuds).

Pour observer la résistance du protocole face au *churn*, nous avons mesuré la longueur des routes KBR et le taux de succès des requêtes de recherche variant la durée moyenne de vie des nœuds (Figure 3. 11 (c) et (d)). En comparaison avec Reverse_Pastry et Pastry, Power_Pastry enregistre le taux de succès le plus élevé (dépassant les 85%) et la route plus courte.

Nous évaluons maintenant les techniques de recherche aléatoire. Pour cela, nous calculons la longueur des chemins de recherche par inondation et par la marche sélective. D'après la Figure 3. 11 (e), l'inondation produit des chemins d'environ $\log \log \mathrm{N}$ tandis que les chemins de la marche sélective croissent en $N^{2-4/\alpha}$. Ces résultats sont similaires à ceux de Power_Chord et confirment par conséquent la nature sans échelle du graphe Power_DHT. D'un autre côte, les recherches aléatoires produisent une signalisation importante notamment pour la recherche par inondation Figure 3. 11 (f).

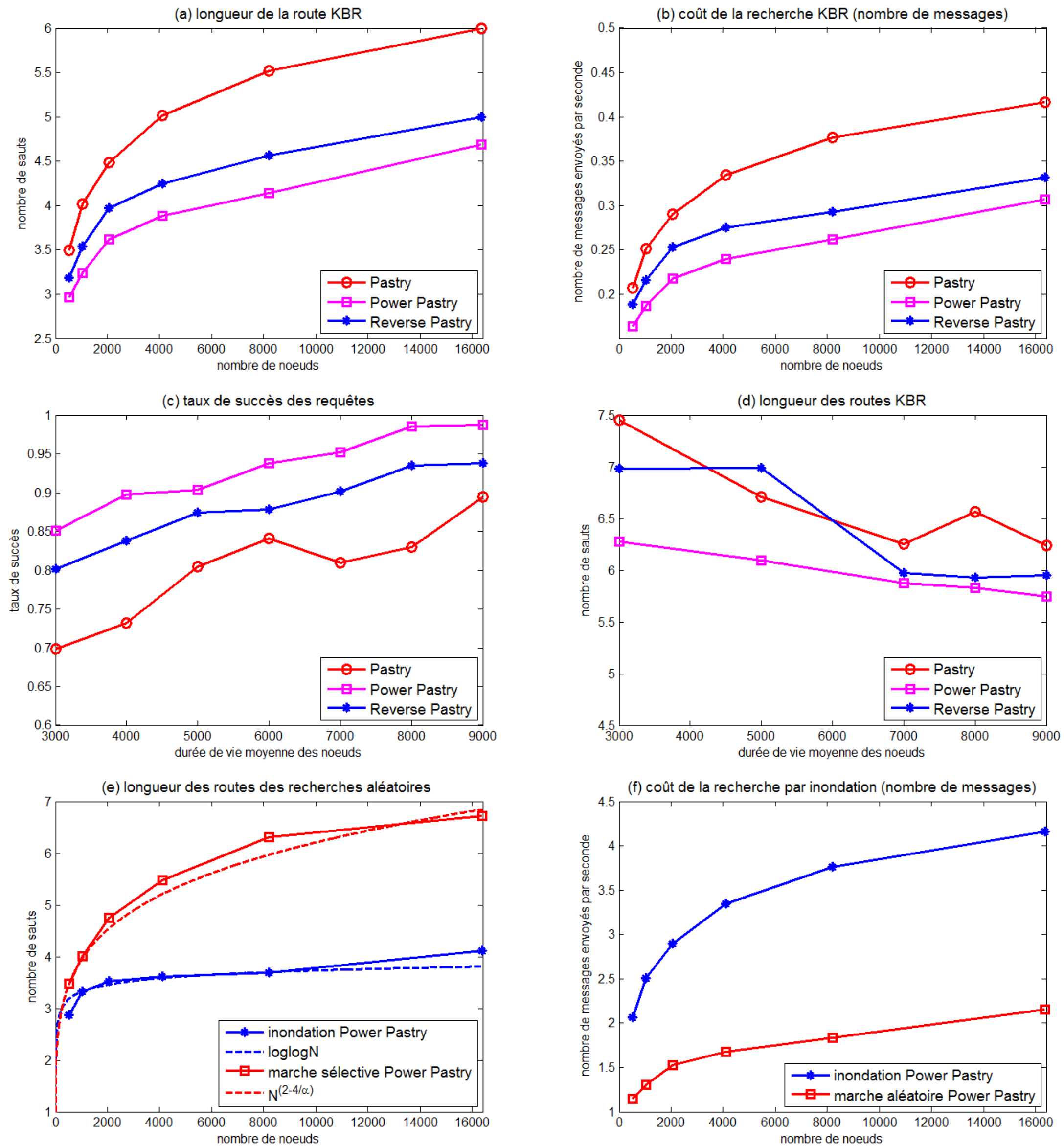


Figure 3. 11 Résultats de Power_Pastry

Enfin, nous mesurons pour Power_Pastry, la fréquence d'utilisation de chacune de ses tables lors du routage. Le routage dans Power_Pastry se fait comme suit : à la réception d'une requête de recherche, si le nœud intermédiaire trouve la destination, la requête est renvoyée directement au nœud destination trouvé dans la table des *reverse* ou parmi des nœuds feuilles. Sinon, le nœud renvoie la requête au nœud le plus proche qui se trouve dans la table de routage ou dans la table des *reverse*. Pour chacune des tables, nous calculons le nombre de fois qu'elle est employée pour trouver le prochain saut ou la destination finale. Le résultat est présenté dans la Figure 3. 12(f). Nous remarquons que Power_Pastry utilise sa table de *reverse* au minimum dans 30% des cas. Nous remarquons également que l'utilisation des nœuds *reverse* pour trouver la destination finale augmente avec la taille du réseau (plus de 20% pour

le réseau le plus large). De plus, la fréquence d'utilisation de la table $reverse_{table}$ pour trouver la destination finale augmente avec le nombre de nœuds, tandis que celle des nœuds feuilles diminue.

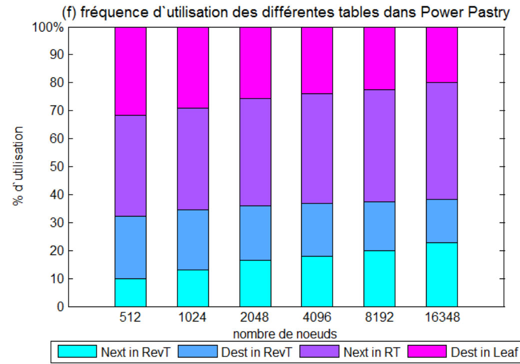


Figure 3. 12 Fréquence d'utilisation des tables dans Power Pastry

Résultats Power_Kad

Nous présentons maintenant les résultats de l'approche Power_DHT appliquée à Kademlia (k=8). Le nombre de nœuds *siblings* est fixé à 4. Nous utilisons les mêmes paramètres et scenarii de simulation que Power_Chord et Power_Pastry.

La Figure 3.13 (a) présente le nombre moyen de sauts nécessaires pour le routage. Nous remarquons d'abord que les routes de Kademlia sont plus courtes que celles de Chord et Pastry. Ceci est du au fait que Kademlia emploie les recherches parallèles sur tous les nœuds du buckets ce qui réduit les chemins de recherche. Power_Kad produit les chemins les plus courts (17,5% de sauts en moins pour le réseau le plus large) ainsi que la plus petite signalisation. Aussi, en variant le *churn*, Power_Kad affiche le meilleur taux de succès dépassant les 90%. Contrairement à Power_Pastry, nous remarquons que l'amélioration apportée par Reverse_Kad est faible par rapport à celle de Power_Kad. Vu que Kademlia est naturellement bidirectionnel, l'amélioration de Reverse_Kad est seulement apportée par l'information sur les voisins séquentiels.

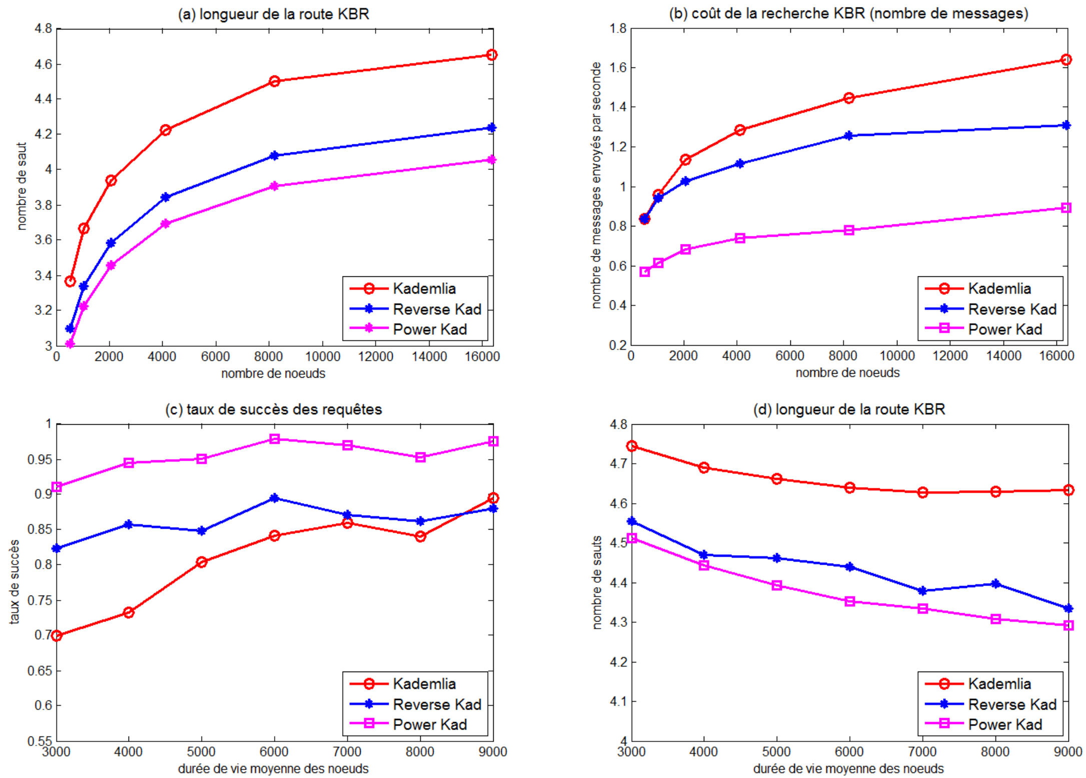


Figure 3. 13 Résultats de Power_Kad

Comme pour Power_Pastry, nous mesurons la fréquence d'utilisation des tables de routages et des nœuds *reverse*. Nous remarquons (Figure 3. 14) que Power_Kad utilise ses tables des *reverse* dans au moins 50% des cas. Il faut toutefois noter que la procédure de routage multi chemins dans Kademlia est différente. En effet, à la réception d'une requête de recherche, un nœud Kademlia retourne un ensemble de voisins qui regroupe les nœuds les plus proches de la clé recherchée. Ces nœuds peuvent être des voisins du *bucket*, des voisins *reverse* ou des voisins *siblings*. Par conséquent, dans 50% des cas au moins, cet ensemble contient un nœud *reverse*.

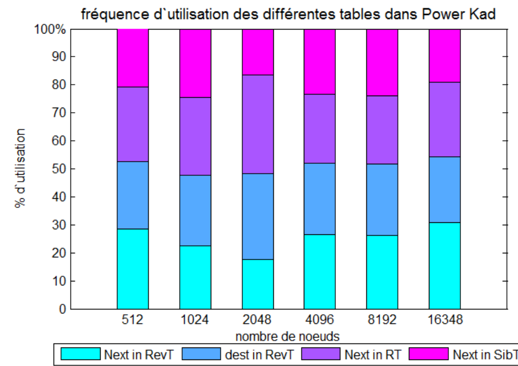


Figure 3. 14 Fréquence d'utilisation des tables dans Power_Kad

Synthèse

L'application de l'approche Power_DHT a produit une amélioration des performances par rapport aux structures classiques et par rapport aux structures équilibrées (Balanced_Chord). Pour la méthode de routage KBR, Power_Pastry affiche les meilleures performances. En effet, les routes sont raccourcies de 15 à 20% et le nombre de messages est réduit d'au moins 20% (Figure 3. 15 (a et b)).

En revanche, l'amélioration pour Power_Kad est moins perceptible notamment en nombre de sauts nécessaires pour le routage. Toutefois, Power_Kad reste toujours meilleur que Kademlia. Vu que Kademlia emploie déjà le routage bidirectionnel, l'application de l'approche Power_DHT sera moins importante. De plus, la taille importante des tables de routage en $O(k * \log_{2^b} N)$ (section 2.2.3) assure au préalable un grand voisinage pour les nœuds Kademlia. Contrairement à Chord et à Pastry, l'information additionnelle sur les voisins *reverse* et les voisins séquentiels augmentent peu la vision des nœuds Power_Kad. De plus, l'absence de procédure de maintenance dans Kademlia rend difficile la construction du graphe en loi de puissance.

Enfin, Power_Chord améliore de 10 à 17% le nombre de sauts moyen et jusqu'à 22% la charge du réseau. Cependant, face au *churn*, ses performances se dégradent et le taux de succès chute en deçà de 65%.

L'extension des messages de maintenance *fix_finger*, *row_request* et *find_node* avec l'information sur les voisins séquentiels augmentent la taille totale (en moyenne) des messages de maintenance pour Power_Chord et Power_Kad. A l'inverse, pour Power_Pastry, la taille totale des messages de maintenance est réduite de 15% en moyenne (Figure 3. 15(d)). La

diminution du nombre de messages transmis compense par conséquent l'augmentation due à l'extension des messages.

Pour les techniques de recherche aléatoire, Power_Chord et Power_Pastry ont montré des performances similaires à celles d'un graphe en loi de puissance. Le routage par inondation se fait en environ $\log \log N$ sauts et la marche sélective s'effectue en environ $N^{2-4/\alpha}$. Les méthodes de recherches aléatoires induisent aussi une signalisation conséquente, notamment pour la recherche par inondation.

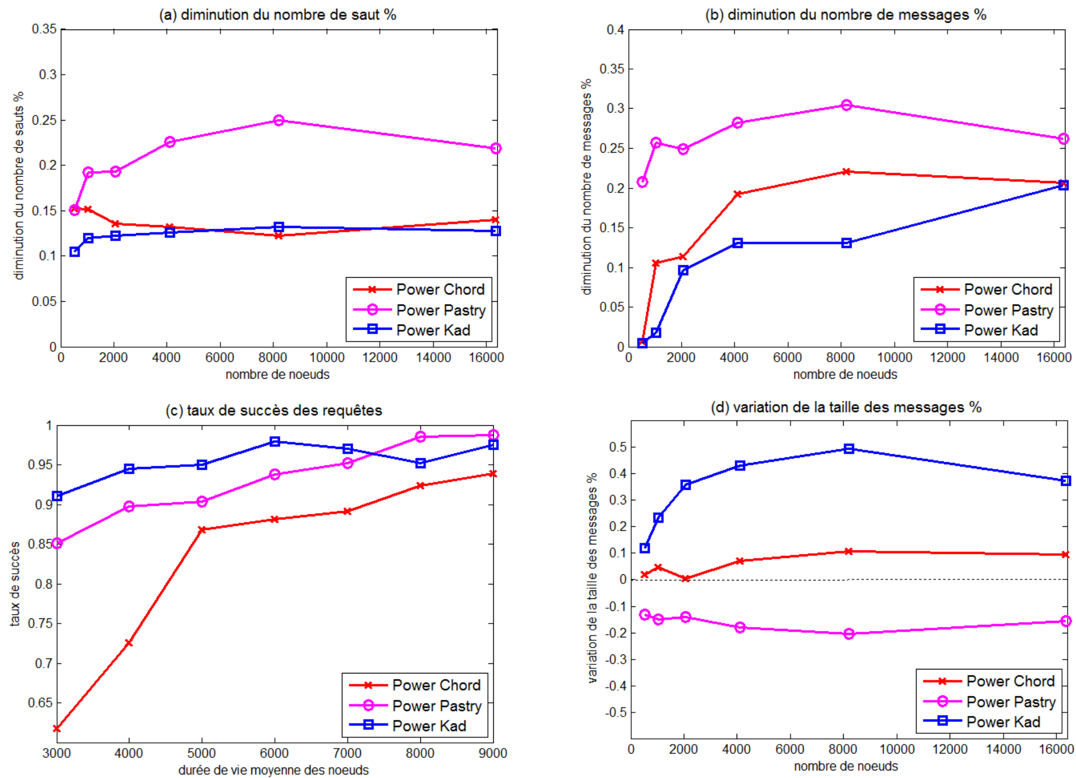


Figure 3. 15 Comparaison des Power_DHTs

Nous n'avons pas implémenté les techniques de recherche aléatoire dans Power_Kad vu que les tables de routage maintenues sont de grandes tailles et que Kademlia emploie déjà les recherches parallèles. En effet, l'inondation à chaque saut des $k * \log_{2^b} N$ voisins plus les nœuds *reverse* dans Power_Kad produirait une très forte signalisation.

3.4 Conclusion

Dans ce chapitre nous avons présenté Power_DHT, une nouvelle structure d'interconnexion et de routage. La motivation de nos travaux part de l'observation d'une grande dichotomie dans la distribution des degrés dans les DHTs déployées. Notre idée est d'exploiter cette hétérogénéité en transformant le réseau vers une structure décentralisée. Pour cela, dans un premier temps, nous avons proposé Reverse_DHT, une structure de routage bidirectionnelle. Les résultats de l'évaluation de Reverse_DHT nous ont encouragés à pousser encore plus loin notre idée. Pour exploiter davantage cette hétérogénéité, nous avons ré-interconnecté le graphe de la DHT selon un graphe sans échelle.

Notre structure Power_DHT transforme dynamiquement la DHT en une structure décentralisée, exhibant les propriétés d'un graphe sans échelle (distribution des degrés, diamètre des recherches aléatoires). Les propriétés de Power_DHT nous a permis l'intégration efficace du routage KBR et des méthodes de recherches aléatoires, à la fois.

Cette nouvelle structure décentralisée constitue une alternative de choix pour la construction spontanée et dynamique de structures décentralisées non hiérarchiques basées sur les super nœuds. Les nœuds fortement connectés qui émergent naturellement du graphe en loi de puissance peuvent être considérés comme des super nœuds, vu que les procédures de recherche et de routage se font essentiellement par ces points centraux.

Nous avons appliqué Power_DHT à trois DHTs flexibles : Chord, Pastry, et Kademlia. En changeant le comportement des nœuds localement, Power_DHT construit une topologie décentralisée tout en conservant la structure. D'autres structures moins souples seraient donc moins adaptées à Power_DHT vu qu'elles n'offrent pas de flexibilité dans la sélection du voisinage. Par ailleurs, l'information sur les voisins séquentiels n'est pas disponible dans toutes les DHTs. Cependant, comme pour Kademlia elle peut être intégrée facilement dans plusieurs structures.

L'application de Power_DHT à Chord, Pastry et Kademlia lors des recherches KBR a produit une amélioration des performances notamment pour la DHT de Pastry. L'amélioration de Power_Kad par rapport à Kademlia s'avère moins importante. Power_Chord raccourcit les routes de recherche et allège la signalisation en termes de nombre de messages. Cependant, l'amélioration devient moins perceptible dans le cas des réseaux à *churn* élevé. D'un autre côté, les méthodes de recherche aléatoire, bien que réalisant un diamètre de recherche court,

produisent une signalisation conséquente : c'est le prix à payer pour une diffusion dans le réseau complet.

Pour améliorer davantage notre structure Power_DHT, plusieurs aspects restent à étudier. Nous explorons dans le chapitre suivant quelques pistes complémentaires, susceptibles d'améliorer les performances de Power_DHT. Pour ce faire, nous proposons d'étudier les différents mécanismes de réplication employés dans les DHTs afin d'améliorer la robustesse du protocole. Nous proposons par la suite, un nouveau mécanisme de recherche par inondation basé sur la réplication symétrique, qui réduit la longueur moyenne des recherches tout en réduisant les coûts de signalisation.

CHAPITRE 4.

OPTIMISATION DE L'INONDATION DANS LES DHTS PAR REPLICATION SYMETRIQUE

Un des problèmes majeurs des systèmes P2P concerne la manière de découvrir et d'accéder à la ressource. Les modèles non structurés, bien que simples et efficaces, ont montré leurs limites (section 1.3.2). Leur architecture est en effet fragile, et la recherche aléatoire par inondation est coûteuse et non fiable. Les tables de hachage distribuées sont des infrastructures qui répondent à ces problèmes, en fournissant une infrastructure distribuée. Ces structures DHTs présentent des propriétés intéressantes telles que la maîtrise du nombre de sauts nécessaires au routage d'une requête. Cependant, une limitation majeure de la recherche KBR dans les DHT est l'incapacité de faire des recherches inexactes : il faut connaître la clé exacte d'une donnée pour pouvoir trouver le pair responsable. Or, en pratique, les utilisateurs des systèmes P2P n'ont qu'une information partielle pour identifier les ressources. La méthode de recherche KBR s'avère peu pratique dans ces cas.

Afin de rallier les deux modèles, nous avons proposé dans le chapitre trois une nouvelle structure décentralisée, nommée Power_DHT. Notre structure marie les deux architectures et intègre à la fois, le routage KBR et les méthodes de recherche aléatoire. La recherche par inondation dans Power_DHT, bien que rapide et efficace, a engendré un coût de signalisation conséquent. Nous adressons cette problématique dans le présent chapitre.

La difficulté de l'inondation réside dans son optimisation (section 2.3.4) : cette opération doit consommer un minimum de ressources tout en assurant que tout nœud soit atteint. L'inondation classique est en effet une méthode coûteuse en signalisation et n'est efficace

que pour les objets populaires et donc suffisamment répliqués (84). De ce fait, nous proposons d'optimiser l'inondation dans les DHTs moyennant la réplication.

Ainsi, nous présentons dans ce chapitre une nouvelle méthode d'inondation pour les DHTs nommée Sym_Flood. Nous proposons d'abord d'étudier les différents mécanismes de réplication employés dans les DHTs, pour sélectionner la technique de réplication à employer. Nous détaillons ensuite notre approche Sym_Flood.

Dans cette partie, nous avons choisi de travailler avec la DHT de Chord. Nous allons voir par la suite, que notre proposition peut être aussi appliquée à Pastry et Kademlia.

4.1 1ère étape : Evaluation des méthodes de réplication dans les DHTs

La plupart des algorithmes DHTs emploient différentes méthodes de réplication afin de pallier les problèmes liés à la faible durée de vie de certains nœuds dans le réseau. Les avantages d'employer une méthode de réplication peuvent être nombreux. Outre un meilleur taux de succès en cas de pannes, certains paramètres de performances peuvent être améliorés grâce à la réplication, tels que le diamètre et des délais des recherches. En effet, la réplication permet de diminuer le nombre de sauts nécessaires pour trouver une copie de l'objet si les copies sont judicieusement placées dans le réseau. De plus, la réplication permet d'augmenter la vitesse de récupération d'un objet en augmentant le nombre de sources potentielles. Il est en effet possible de récupérer en parallèle des parties différentes d'un même objet.

Il existe plusieurs algorithmes de réplication des données qui stockent les données et maintiennent périodiquement des copies de données. Nous classons ces méthodes de réplication selon trois grandes catégories :

- la réplication dans les voisins séquentiels
- la réplication par multi-publication des clés
- la réplication dans le chemin

Toutefois, ces différentes approches de réplication n'ont pas les mêmes avantages et inconvénients. De plus, chaque méthode de réplication est généralement proposée pour une DHT spécifique. Afin de comparer efficacement ces différentes méthodes, il faut les évaluer selon plusieurs critères de performances (section 2.3), et ce dans une même structure.

Nous proposons dans cette première partie du chapitre d'analyser les différentes méthodes de réplication employées dans les systèmes P2P structurés. Nous détaillons d'abord les différentes approches de réplication proposées dans les DHTs, et comparons par la suite ces propositions appliquées à une même structure. Nous choisissons dans ce travail la DHT de Chord.

4.1.1 Méthodes de réplication dans les DHTs

4.1.1.1 Réplication dans les voisins séquentiels (Neig_R)

Dans les algorithmes DHTs, les nœuds maintiennent périodiquement une liste de voisins séquentiels utilisés pour le routage dans l'overlay. Ces nœuds sont sélectionnés suivant l'algorithme de la DHT. Dans Chord par exemple, chaque nœud maintient la liste de ses successeurs dans l'anneau ; la réplication dans les successeurs peut donc être utilisée. Dans Pastry (130), ou Kademlia (131), le nœud maintient une liste de nœuds feuilles ou *siblings* composée des nœuds les plus proches numériquement. Une réplication dans les feuilles et *siblings* peut également être employée. La sélection et la mise à jour de la liste des voisins séquentiels faisant partie de l'algorithme de la DHT original, aucune information d'état supplémentaire n'est requise. La maintenance des réplicas est par conséquent simple et directe.

Il est aussi possible d'appliquer ces mêmes méthodes de réplication aux différentes DHTs. Cependant, la gestion des voisins séquentiels additionnels induirait des coûts supplémentaires. Le choix de la méthode de réplication à employer est directement lié aux caractéristiques de la DHT. Par conséquent un mécanisme de réplication est plus adapté à certaines structures qu'à d'autres.

Réplication dans les successeurs (succ_R)

La réplication dans les successeurs place l'objet à répliquer dans les successeurs du nœud racine. Ce dernier maintient la liste des identifiants de ses successeurs dans sa table de voisins séquentiels. L'avantage de cette méthode est que si le nœud racine tombe en panne, l'objet sera immédiatement disponible dans le successeur direct de l'objet, qui devient à son tour le nouveau nœud racine. Le degré de réplication r est un paramètre variable à fixer selon les besoins. D'un autre côté, la réplication dans les successeurs concentre l'objet autour du nœud racine et crée un déséquilibre dans le réseau, surtout si un nœud possède plusieurs objets populaires. De plus, Succ_R ne permet pas de raccourcir les chemins de recherche puisque la requête de recherche est d'abord routée vers les nœuds racine*s*. Si l'on veut accéder aux différentes copies de l'objet, il faut d'abord atteindre le nœud racine pour avoir la liste des nœuds répliquant : les successeurs. La réplication dans les voisins successeurs est employée dans les systèmes Bunshin (132) et CFS (40), tout deux basés sur l'algorithme Chord (44).

Réplication dans les nœuds feuilles (leaf_R)

L'ensemble des nœuds feuillent regroupent les nœuds successeurs et prédécesseurs. Dans ce cas, l'objet est répliqué dans $r/2$ voisins successeurs et $r/2$ voisins prédécesseurs.

Dans les algorithmes DHTs où l'acheminement des messages se fait dans deux directions (en amont et en aval), il est plus intéressant de répliquer l'objet dans le voisinage du nœud racine et donc, dans ses successeurs et ses prédécesseurs. La recherche peut donc aboutir avant d'atteindre le nœud racine en récupérant l'objet recherché depuis un prédécesseur de ce nœud. Dans ce cas, la route de l'overlay est raccourcie et le délai de recherche est réduit. Contrairement à la méthode de réplication dans les successeurs, où toutes les requêtes se dirigent en premier vers le nœud racine, dans ce schéma, les requêtes de recherches sont desservies aussi par les nœuds prédécesseurs, ce qui permet une meilleure distribution des requêtes.

La réplication dans les voisins feuilles ou *siblings* est utilisée dans les DHTs de Pastry et Kademlia. Dans PAST (18), un système de stockage distribué basé sur Pastry, l'objet est répliqué dans les r nœuds feuilles les plus proches numériquement. Afin de mieux répartir les réplicas, PAST propose en plus de répliquer les objets dans les voisins (les feuilles) des nœuds feuilles si ces derniers sont surchargés. Le système de notification Scribe (48), basé sur Pastry, emploie aussi la réplication dans les nœuds feuilles avec un degré de réplication fixé à 5.

Nous remarquons que les méthodes de réplication dans les nœuds feuilles et dans les successeurs sont similaires. Cependant, chacune est mieux adaptée à une DHT particulière. L'ensemble des nœuds feuilles regroupe à la fois les successeurs et les prédécesseurs du nœud racine. La réplication dans les feuilles apparait donc comme une version plus générale de la réplication dans les successeurs.

La réplication Neigh_R peut être appliquée à d'autres géométries au prix de la maintenance additionnelle de voisins séquentiels. Dans un hypercube de dimension d (CAN par exemple), la réplication dans les voisins peut être appliquée en copiant l'objet dans les prédécesseurs et successeurs du nœud racine dans les d directions, et donc dans tout le voisinage de nœud racine.

4.1.1.2 Multi-publication des clés : Publications sous plusieurs clés

Il s'agit d'associer à un objet donné plusieurs identifiants et donc plusieurs clés. L'objet est inséré r fois sous r clés/identifiants différents.

La réplication de l'objet par multi-publication des clés se fait indépendamment de la DHT utilisée. Le nœud qui publie l'objet calcule simplement différentes clés puis insère l'objet dans les nœuds correspondants. Cette méthode peut être appliquée à n'importe quelle DHT. Les nœuds doivent toutefois être informés de la technique de calcul des clés employée afin de savoir où et comment chercher un objet. Il existe plusieurs approches pour calculer ces clés. Nous présentons dans cette partie les principales méthodes.

Multiple fonctions de hachage (multi_hash)

Afin de publier r fois l'objet dans le réseau logique, son identifiant est haché avec les r fonctions de hachage, résultant en r identifiants différents. Chaque copie de l'objet est ensuite insérée dans le nœud responsable de sa clé. Lors de la recherche, un nœud a le choix parmi r destinations, associées chacune à une clé différente. Le nœud peut choisir par exemple l'objet le plus proche numériquement, réduisant ainsi les délais de recherche.

La réplication par multi-hachage permet une meilleure tolérance aux pannes et une localisation rapide des copies de l'objet. Cependant, il faut noter que le nombre de clés attribuées à chaque nœud augmente avec un facteur r en moyenne. Cette méthode a été proposée à la base comme optimisation de la DHT de CAN (133), mais son principe peut s'appliquer à n'importe quelle DHT.

Hachage corrélé (corr_hash)

Une deuxième méthode proposée dans (134) (135) et (136) propose d'utiliser une seule fonction de hachage en attribuant à chaque instance un index de corrélation. L'objet est placé dans les nœuds correspondant aux adresses déterminées par la fonction $\text{h}\ (\text{i}\ ;\ \text{ID})$ avec $1 \leq i \leq r$. La fonction d'allocation h est définie comme suit :

$\text{h}\ (1;\ \text{ID}) = \text{ID};\ \text{h}\ (\text{i};\ \text{ID}) = \text{h}\ (\text{i+ID})$, '$+$' est la fonction de concaténation, i l'index de l'instance et h la fonction de hachage.

Dans les deux approches (multi_hash et corr_hash), la localisation des différents réplicas est immédiate, puisqu'il suffit de calculer les différents IDs pour pouvoir accéder aux réplicas. L'accès aux différents réplicas peut se faire d'une façon séquentielle ou parallèle.

Cependant, contrairement à la réplication dans les voisins séquentiels, la maintenance des réplicas par multi_hash est complexe puisqu'un nœud répliquant l'objet ne connait pas la localisation des autres réplicas. Par conséquent, si le taux de pannes dans le réseau est élevé, il devient probable que les nœuds stockant les différentes copies d'un même objet tombent en panne, ce qui rendra l'objet inaccessible. La solution consiste à associer à chaque objet un nœud responsable qui gère et maintient ses réplicas : le nœud racine.

Réplication symétrique (sym_R)

La réplication symétrique a été proposée dans le système DKS (39). DKS propose aussi d'associer à chaque objet r identifiants ID_i, mais différemment. La particularité de cette méthode est que les identifiants sont choisis de manière à pouvoir calculer n'importe quel ID_i, dans l'ensemble des r réplicas, à partir de n'importe quel ID_j. Ceci se fait comme suit : l'espace des identifiants est partitionné en r segments de taille m/r (m étant la taille de l'espace des identités). Ensuite, les identifiants ID_i $(2 \leq i \leq r)$ sont calculés à partir de la clé $ID_1 = \text{hash (objet)}$ par la fonction suivante :

$$ID_i = (ID_1 + (i * \frac{m}{r}))\, mod\, m$$

L'intérêt de cette méthode de calcul est que chaque nœud répliquant l'objet peut connaitre les IDs des différents réplicas et peut donc les maintenir. Chaque nœud contenant un réplica peut accéder aux autres copies et vérifier périodiquement que le taux de réplication est maintenu et que la disponibilité de l'objet est assurée.

4.1.1.3 Réplication dans le chemin (path_R)

Dans Path_R, lorsqu'un nouvel objet est inséré, il est automatiquement copié dans le chemin de recherche de sa localisation. Ainsi, l'objet est répliqué dans tous les nœuds de la route séparant le nœud qui a inséré l'objet (la source) et le nœud qui l'héberge (racine). Cette méthode de réplication a été proposée dans l'algorithme Tapestry (137).

Dans les DHTs, le routage se fait saut par saut en s'approchant numériquement de la destination. Les chemins de recherche d'un même objet, provenant de nœuds différents convergent avant ou dans la destination racine. La réplication sur le chemin réplique l'objet dans les nœuds du chemin précédant la destination finale, et permet ainsi de raccourcir les chemins de recherche. En revanche, cette méthode ne permet pas l'équilibrage de charge puisque les requêtes de recherche vont toutes aboutir près du nœud racine.

La réplication dans le chemin peut être appliquée à des géométries différentes d'une façon similaire. Toutefois, le problème de maintenance devient plus compliqué si plusieurs routes possibles existent entre la source et le nœud racine (CAN (133)) ou si le routage se fait par des requêtes émises en parallèles (Kademlia (131)).

L'application multicast Bayeux (138), qui utilise la DHT de Tapestry, emploie la réplication dans le chemin. Le nœud qui détient l'objet envoie un message publish (object ID) au nœud racine. Après la localisation de la destination racine, le nœud insère l'information de localisation (object ID, racine ID) dans chaque nœud intermédiaire. Bayeux stocke dans ces nœuds des pointeurs vers le nœud racine qui détient l'ID de l'objet et non l'objet lui même.

4.1.1.4 Synthèse

Les techniques de réplication dans les voisins et dans le chemin sont directement liées à la méthode de routage utilisée, et doivent être adaptées à la géométrie de la DHT en conséquence. Les méthodes de réplication par multi-publication des clés, en revanche, s'applique à toutes les DHTs. Cependant, cette approche exige que les nœuds soient informés au préalable de la technique de réplication employée. Nous résumons dans le tableau suivant les principales caractéristiques des méthodes de réplication déjà citées.

Tableau 4. 1 Comparaison des méthodes de réplication dans les systèmes structurés

	Réplication dans les voisins		Multi-publication des clés		Réplication dans le chemin	
Classification	Succ_R	Leaf _R	Sym_R	Multi-hash Corr_hash	Une route	Routes multiples
Flexibilité	réplication transparente : réplication implicite intégrée dans le routage DHT.		basée sur les nœuds : peut être appliquée à toutes les DHT, mais les nœuds doivent être informés de la technique de replication utilisée.		basée sur le routage: replication implicite intégrée dans le routage DHT.	
Maintenance	assurée par le nœud *racine* puisqu'il connait la liste de ses voisins		assurée par les nœuds répliquant	assurée par le nœud *racine* seul à pouvoir calculer les # clés	assurée par les nœuds du chemin	maintenance plus complexe
Coût	pas de coût additionnel : la liste des successeurs est pré configurée.		le nœud répliquant doit localiser les réplicas à mettre à jour		les nœuds répliquants doivent sauvegarder le chemin	
Proximité	la requête est d'abord dirigée vers le nœud *racine*	la requête peut aboutir avant le nœud *racine*	localisation du réplica le plus proche numériquement		la requête peut aboutir avant le nœud *racine*	

En résumé, certaines méthodes de réplication peuvent s'appliquer à toutes les DHTs (réplication symétrique, sur le chemin). D'autres, au contraire, sont directement liées à la géométrie de la DHT et sont souvent proposées en association avec une géométrie bien définie. Nous proposons dans la suite de comparer ces différentes méthodes de réplication

dans une même géométrie DHT et sous les mêmes paramètres. Nous avons choisi de travailler avec l'anneau de Chord. Nous comparons dans ce travail la réplication dans les voisins successeurs, la réplication sur le chemin et la réplication par multi-publication de clés symétrique.

4.1.2 Evaluation des méthodes de réplication

Nous avons remarqué dans la littérature l'absence de mécanisme de maintenance propre à chacune des méthodes de réplication. Pour cela, avant de commencer notre évaluation, nous définissons pour chacune de ces méthodes un mécanisme de maintenance simple qui vise à maintenir le degré de réplication r.

4.1.2.1 Spécification des mécanismes de maintenance

Maintenance pour la réplication dans les successeurs

Nous proposons dans cette approche un mécanisme de maintenance relativement simple. Pour maintenir de degré de réplication à r, il suffit de prendre en compte deux évènements, à savoir : le départ d'un successeur (panne ou départ volontaire) et l'arrivée d'un nouveau successeur (qui prend la place d'un ancien successeur). Dans le premier cas, on risque de perdre tous les réplicas de la donnée insérée si tous les successeurs sont amenés à quitter le réseau. Dans le second cas, l'effet inverse peut se produire, et on peut se retrouver avec beaucoup plus de réplicas qu'initialement prévu.

Pour empêcher que de tels cas se produisent, le nœud racine vérifie périodiquement que ses successeurs possèdent un réplica, et insère la donnée si nécessaire. De plus, chaque nœud répliquant vérifie périodiquement si chaque réplica qu'il sauvegarde correspond à une donnée sur un nœud racine dont il est successeur.

Maintenance pour la réplication symétrique

Pour chaque objet avec un ID_i stocké, le nœud répliquant calcule la clé ID_{i+1} correspondant à la clé du réplica suivant. A intervalles réguliers, le nœud vérifie si le nœud responsable de la clé ID_{i+1} possède bien l'objet et l'insère si besoin. Ainsi le nœud qui a le premier réplica s'occupe du second, le second du troisième...et le dernier du premier.

Maintenance pour la réplication sur le chemin

Nous utilisons un mécanisme de maintenance similaire à celui de la réplication symétrique. Chaque nœud répliquant dans la route vérifie que le prochain nœud sur le chemin possède le réplica en question. Le prochain nœud répliquant correspond au voisin logique qui mène vers la destination racine. Un nœud intermédiaire vérifie si le prochain saut possède l'objet et l'insère si besoin. Le nœud racine, quant à lui, vérifie si l'objet se trouve encore dans le nœud qui a inséré l'objet.

4.1.2.2 Résultats

Topologie, scenarii de trafic et critères de performances

Afin d'évaluer les différentes méthodes de réplication dans une même architecture DHT et sous les mêmes paramètres, nous avons choisi d'implémenter ces méthodes avec la DHT de Chord. Dans cette partie du travail, nous avons choisi de procéder par émulation (nous disposons d'une implémentation en C de l'algorithme Chord).

L'´emulation permet d'étudier un système de manière plus réaliste. Il s'agit, en effet, d'utiliser le logiciel réel dans un environnement que nous contrôlons et de tester ses réactions face à cet environnement. L'intérêt de cette approche est qu'elle nous permet d'étudier notre système dans toute sa complexité, sans avoir à faire de simplifications sur son fonctionnement interne.

Nous avons donc intégré les différentes approches de réplication plus les mécanismes de maintenance à notre implémentation existante. Nous émulons un réseau fonctionnant avec la DHT de Chord et utilisant une des trois méthodes de réplication à comparer, à savoir : la réplication dans les successeurs, dans le chemin et la réplication symétrique.

La taille du réseau est fixée à 500 nœuds, chaque nœud stocke 10 objets. Le nombre de successeurs est fixé à 8 et le nombre de voisins *finger* est fixé à 8. Une fois le réseau stable, des nœuds s'insèrent et quittent le réseau suivant une loi de poisson de moyenne variable (0 *churn*/60 sec, 5/60, 10/60, 20/60). Parallèlement, 100 nœuds choisis aléatoirement envoient toutes les 10 secondes un message vers un nœud choisi aléatoirement suivant une distribution uniforme. Le timeout et le nombre d'essaie pour chaque requête sont fixés respectivement à 5 sec et 2. Enfin, la maintenance des réplicas s'effectue toutes les 10 secs selon l'algorithme de maintenance défini plus haut.

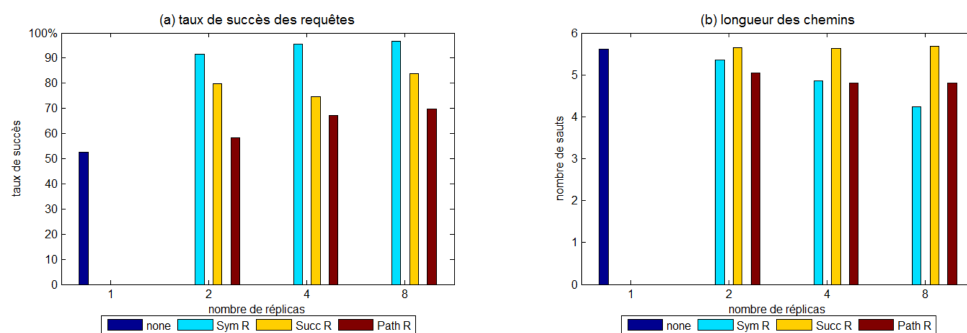
Nous évaluerons les différentes approches de réplication par émulation faisant varier deux paramètres : le degré de réplication *r* et le *churn*.

Effet de la variation du degré de réplication

Dans un premier temps nous faisons varier le degré de réplication pour déterminer le nombre de réplicas à employer dans le reste des émulations. L'augmentation du degré de réplication permet d'offrir une meilleure robustesse face au *churn*. Cependant, ceci requiert plus de mémoire et signalisation pour le stockage et la maintenance des réplicas. Le degré de réplication doit donc être paramétré en conséquence. Pour notre évaluation, nous testons 4 valeurs : 1 (pas de réplication), 2,4 et 8 (nombre de successeurs). La moyenne du *churn* est fixée à 20/min.

La Figure 4. 1(a) montre que le taux de succès des requêtes augmente avec le nombre de réplicas pour la réplication symétrique, qui produit le meilleur taux de succès (plus de 90% pour tous les degrés de réplication).

Pour la réplication dans le chemin, cette amélioration est moins visible. En effet, le nombre de réplicas est majoré par la longueur du chemin. De ce fait, lorsque le nombre de réplicas passe de 4 à 8 et que la longueur de la route ne dépasse pas *k* sauts ($k<8$), le nombre de réplicas insérés est limité à *k*. Le nombre de réplicas dépend donc de la longueur de la route et du degré de réplication. Ce résultat est également visible pour les autres critères évalués : la charge du réseau et les délais de recherches varient très peu quand le nombre de réplicas passe de 4 à 8.



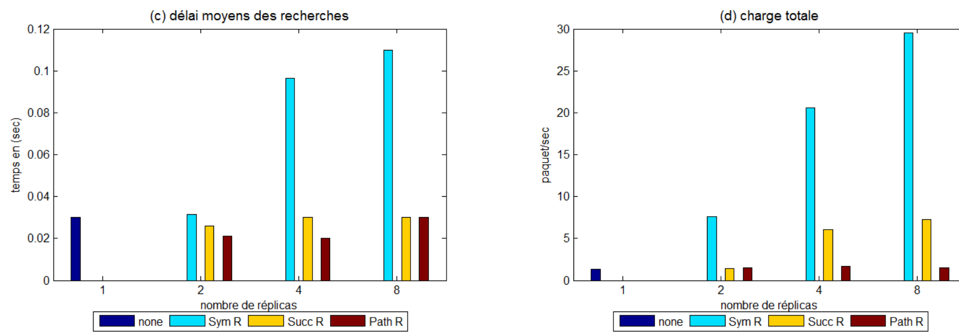


Figure 4. 1 Effet de la variation du degré de réplication

Lorsque le degré de réplication croît, le nombre de sources pour un même objet augmente, par conséquent la signalisation requise pour l'insertion et la maintenance des réplicas augmente aussi. Cette augmentation est plus visible pour la réplication symétrique, notamment lorsque le nombre de réplicas passe de 4 à 8 (Figure 4. 1(d)). La maintenance des réplicas symétriques suppose que chaque nœud vérifie périodiquement le prochain réplica de chaque objet qu'il partage. Cette vérification périodique augmente davantage la signalisation. Dans le cas de la réplication dans les successeurs, chaque nœud vérifie également que ses objets sont bien présents dans ses successeurs. Cependant, la liste des successeurs est déjà connue, chose qui facilite et allège la procédure de maintenance.

Concernant la longueur des routes (Figure 4. 1 (b)), nous remarquons que la réplication dans les successeurs produit des chemins aussi longs que sans réplication. Ceci est du au fait que tous les réplicas sont placés après le nœud racine, et donc le plus court chemin vers l'objet et celui vers le nœud racine. Appliquée à Pastry, la réplication dans les voisins feuilles réduirait donc le nombre de sauts vu que les réplicas seraient placés de part et d'autre du nœud racine.

Nous observons ce résultat dans la réplication sur le chemin. En effet, dans ce cas, les routes de recherches sont plus courtes puisque les réplicas sont placés avant le nœud racine et donc les requêtes de recherche aboutissent avant d'arriver à la destination racine.

Le meilleur résultat est toutefois enregistré par la réplication symétrique. Le placement à intervalles réguliers des réplicas fait que chaque requête localise l'objet le plus proche, qui se situe bien souvent avant le nœud racine.

Pour la réplication symétrique et la réplication dans les successeurs, nous remarquons que les délais de recherche augmentent avec le degré de réplication (Figure 4. 1 (c)). Ces délais

augmentent vu que le nœud n'arrive pas à récupérer le réplica le plus proche en cas de *churn*. Pour la réplication symétrique, la recherche aboutie au bout de la $i^{ème}$ tentative de recherche ($i^{ème}$ réplica le plus proche), alors que pour la réplication dans les successeurs, la recherche se termine dans un successeur du nœud racine.

L'augmentation du nombre de réplicas améliore donc le taux de succès et la longueur des routes mais augmente aussi la charge du réseau et les délais de recherche. Il serait possible d'émettre des requêtes de recherche en parallèle afin de réduire les délais. Toutefois, cette solution augmenterait davantage la charge du réseau. D'un autre côté, nous observons que l'amélioration des performances dans un réseau de 500 nœuds, est moins visible lorsque le nombre de réplicas passe de 4 à 8. Pour ces raisons, nous considérons dans la suite de notre évaluation un degré de réplication égale à 4.

Effet du churn

Dans la deuxième partie de notre évaluation, nous faisons varier le *churn* en fixant le degré de réplication à 4. La Figure 4. 2 illustre les résultats.

Dans un réseau statique et pour toutes les approches, plus que 99% des requêtes aboutissent avec succès. Quand le taux de *churn* augmente, le taux de succès diminue considérablement sauf pour la réplication symétrique qui garde une valeur supérieure à 85%. Moins de 65% des requêtent aboutissent pour la réplication dans les successeurs et moins de 45% pour la réplication sur le chemin.

La réplication symétrique produit aussi le trafic de signalisation le plus important. Cette quantité de trafic augmente avec le *churn*, notamment au delà de 20 connexions/déconnexions par minute. Pour les autres approches, cette augmentation est moins visible.

Concernant la longueur des routes, nous remarquons que la réplication sur le chemin produit les chemins et les délais les plus courts pour toutes les valeurs de *churn*. Cependant, le taux de succès enregistré reste très faible (inférieur à 60%). Les chemins courts enregistrés correspondent seulement aux 60 % requêtes abouties. Par conséquent, le fait que les routes et les délais de recherche soient courts ne garantit pas les bonnes performances globales.

La réplication dans les successeurs génère, quant à elle, des routes aussi longues que celles de Chord sans réplication. Enfin la réplication symétrique produit des routes de longueur

comparables à celle de la réplication sur le chemin, tout en maintenant un taux de succès supérieur à 85%.

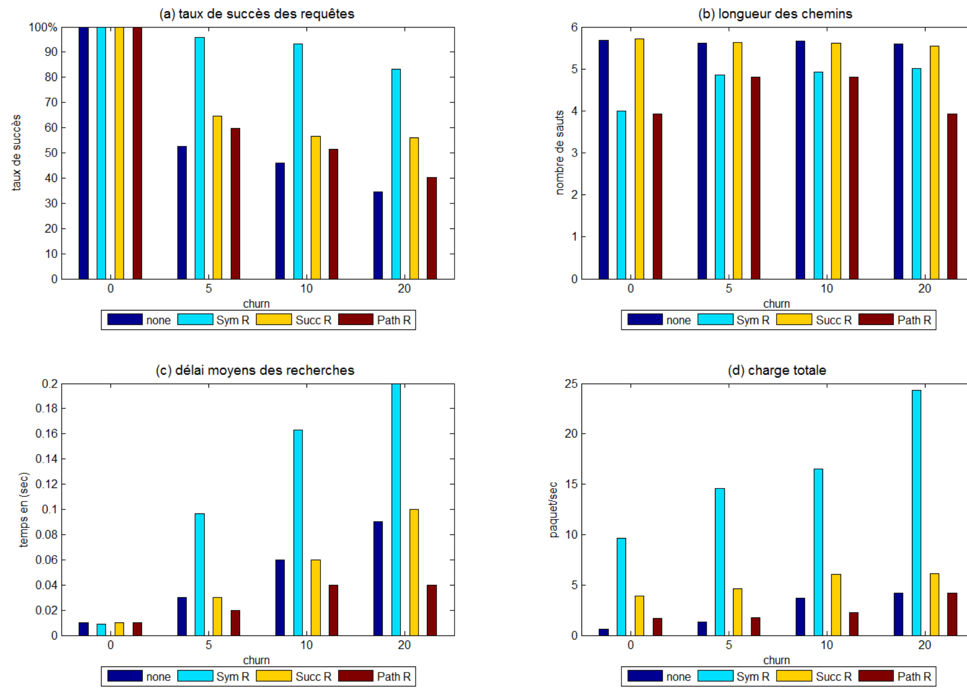


Figure 4. 2 Effet du churn

4.1.3 Conclusion

Dans cette première partie du chapitre, nous avons présenté les différentes techniques de réplication proposées dans les DHTs. Dans la littérature, chacune de ces approches est proposée en association avec une géométrie spécifique. Nous décidions donc de comparer ces différentes méthodes de réplication dans une même structure DHT et avec les mêmes paramètres et critères d'évaluation.

Conformément aux résultats de (139), les résultats de nos émulations ont montré l'avantage de la réplication symétrique par rapport à la réplication sur le chemin et la réplication dans les successeurs en termes de taux de succès et longueur des routes. En moyenne, la réplication symétrique délivre 20% de requêtes en plus, avec 10 % de sauts en moins.

La charge du réseau est cependant alourdie, et les délais sont allongés en cas de *churn*. La charge supplémentaire est due à la maintenance des réplicas qui, périodiquement, vérifie le prochain réplica et l'insère si besoin. Les délais sont rallongés, vu que le nœud n'arrive pas à récupérer le réplica le plus proche numériquement en cas de *churn*. Les simulations des

méthodes de réplication dans (140) (139) montrent que dans un réseau de 500 nœuds, la réplication symétrique produit moins de messages de signalisation comparée à la réplication dans les successeurs. Cependant, dans ces travaux, la maintenance des réplicas se fait réactivement. Dans (140) par exemple, chaque nœud A envoie un message *replicate,* avant de quitter le réseau, pour informer son successeur de son départ. Le successeur du nœud A se charge par la suite de récupérer toutes les données auparavant stockées dans A. Or, dans la pratique, un nœud quitte souvent le réseau sans informer au préalable ses voisins. A moins d'employer un mécanisme de détection de pannes, la réplication devrait se faire d'une manière proactive pour garantir le degré de réplication r (141).

Nos résultats ont montré aussi les limites de la réplication sur le chemin. Bien que les routes soient moins longues et les délais raccourcis, face au *churn*, le taux de succès chute considérablement (en dessous de 45% au delà de 20 *churn*/min).

Enfin, la réplication dans les successeurs a montré des performances moyennes. L'avantage principal de cette approche est la simplicité d'implémentation et de maintenance des réplicas vu que la réplication se fait dans les voisins séquentiels déjà connus. En revanche, cette méthode génère des routes aussi longues que sans réplication, et surtout un taux de succès faible (moins que 60% au delà de 10 *churn*/min).

En résumé, aucune des trois approches ne satisfait tous les paramètres. Toutefois, la réplication symétrique se distingue parmi les autres. Elle garantit l'acheminement correct des requêtes de recherche par des chemins courts, même dans le cas d'un *churn* fort. Nous choisissons donc d'employer la réplication symétrique dans la suite de notre travail.

4.2 2ème étape : Optimisation de la recherche par inondation

L'inondation est la technique de recherche la plus rudimentaire. Elle consiste à répéter un message dans tout le réseau : chaque nœud qui reçoit le message pour la première fois, le renvoie à tous ses voisins (sauf le nœud expéditeur). Ainsi, le message inonde le réseau de proche en proche.

Inspiré par la méthode d'inondation proposée dans (142) (143), qui organise les nœuds de la DHT selon un *spanning tree,* nous proposons d'adapter cet algorithme à la DHT de Chord associé à la réplication symétrique.

4.2.1 Détail de l'algorithme

Nous détaillons dans un premier temps l'algorithme de l'inondation sans l'utilisation de la réplication symétrique.

4.2.1.1 Emission de la recherche

Le nœud source S_1 commence la recherche par inondation en envoyant la requête à tous ses voisins *finger.* La table de *finger* contient généralement des nœuds *finger* redondants (sauf si l'espace est totalement habité). Pour une suite de voisins redondants *(*$finger_i = finger_{i+1}$*),* le nœud émetteur envoie la requête au dernier *finger.*

La requête de recherche contient la clé recherchée et l'argument 'limite' qui délimite l'espace de recherche du prochain nœud c'est à dire le nœud qui va retransmettre la requête. La limite pour le $finger_i(S_1)$ est l'identité du $finger_{i+1}(S_1)$*.* La limite pour le dernier *finger* est le nœud source S_1. Cela veut dire que la limite de l'espace de recherche pour un $finger_i$ est le $finger_{i+1}$: chaque *finger* se charge d'inonder la requête dans l'intervalle qui le sépare du *finger* qui le suit.

4.2.1.2 Traitement de la recherche

Lorsqu'un nœud intermédiaire A reçoit une requête de recherche, il vérifie d'abord s'il possède la clé recherchée. Dans ce cas la recherche s'arrête. Sinon le nœud continue la recherche dans l'intervalle]A, *limite*[défini par l'argument *limite* reçu. Le nœud A retransmet donc la requête de recherche à ses voisins finger dont l'identifiant est inférieur à limite, et modifie l'argument limite dans chaque nouveau message renvoyé. Comme pour le

nœud source, la limite pour chaque *finger* est le *finger* qui suit. Aussi, la nouvelle limite à définir ne doit pas excéder l'ancienne limite reçue dans le message. Par exemple, lorsque le nœud intermédiaire A_j reçoit une requête de recherche avec une limite x_j, il retransmet le message à ses voisins finger dont l'identifiant est inférieur à x_j, et leurs attribue une nouvelle limite telle que la limite du $finger_i(A_j)$ est $min\ [x_j, finger_{i+1}(A_j)]$. Le pseudo code de l'algorithme de recherche est présenté dans la Figure 4. 3.

4.2.2 *Sym_Flood* : Utilisation de la réplication symétrique pour limiter l'inondation

L'algorithme de recherche décrit ci-dessus délimite l'espace de recherche pour chaque nœud et réduit donc le nombre de messages retransmis. En utilisant la réplication symétrique, nous proposons de réduire davantage l'espace des recherches. En effet, le fait de répliquer un objet r fois à des intervalles réguliers, permet de diviser l'espace de recherche en r segments indépendants. L'inondation limitée peut donc s'effectuer sur l'un des espaces (le plus proche) réduisant ainsi le nombre de sauts nécessaires et ainsi le nombre de messages. Si l'objet désiré ne se trouve pas sur le premier segment, la recherche continue dans le second et ainsi de suite jusqu'au dernier. La recherche peut aussi s'effectuer dans les r espaces en parallèle mais au prix d'un surcoût de signalisation.

4.2.2.1 Emission de la recherche

Le nœud source S_1 commence par envoyer la requête de recherche à tous ses voisins *finger* dans le premier segment de l'espace, c'est à dire dans l'intervalle $[S_1,\ (S_1 + m/r)\ mod\ m[$, ($m$ étant la taille de l'espace des recherches). L'argument limite de la requête de recherche doit donc tenir compte de cette nouvelle borne. Ainsi, dans le premier message émis, l'argument limite du voisin $finger_i(S_1)$ est fixé à $min\left[finger_{i+1}(S_1),\ \left(S_1 + \frac{m}{r}\right)\ mod\ m\right]$.

4.2.2.2 Traitement de la recherche

La recherche se poursuit à l'intérieur du segment $[S_1,\ (S_1 + m/r)\ mod\ m[$, comme pour l'inondation sans réplication. Si au bout de n étapes la requête n'aboutit pas, la recherche continue dans le segment suivant $[(S_1 +\ m/r)\ mod\ m,\ (S_1 + 2\ m/r)\ mod\ m[$. Pour

cela, le dernier *finger* F dans l'intervalle $[S_1, (S_1 + m/r) \, mod \, m[$, initie une nouvelle recherche dans l'espace $[F, (F + m/r) \, mod \, m[$.

Ainsi, la requête de recherche parcourt l'espace m segment par segment. Plus le degré de réplication est grand, plus l'intervalle de recherche m/r est petit, et plus la localisation de l'objet sera rapide. En revanche, cela augmente la signalisation nécessaire pour la mise à jour des réplicas et multiplie le nombre de clés attribuées à chaque nœud.

```
Flooding (key, Limit)
for i in 1 to n-1  do                    //n nombre de finger
{     if (Finger_i # Finger_{i + 1})     //sauter un finger redondant
      {     if (Finger_i < Limit)
            {     NextHop = Finger_i
                  if (Finger_{i + 1} < Limit)
                  {     NewLimit = Finger_{i + 1}}
                  else
                  {     NewLimit = Limit}
                  send(key, NewLimit)
            }
      }
}
```

Figure 4. 3 Algorithme de l'inondation dans la DHT

4.2.3 Synthèse

L'inondation Sym_Flood consiste à segmenter l'espace des recherches, puis parcourir en profondeur cet espace segment par segment. Cela permet de réduire l'espace et le coût de recherche en contrôlant le nombre de messages d'inondation transmis.

Nous avons détaillé dans cette partie notre algorithme Sym_Flood, appliqué à Chord. Cependant, notre approche peut être aussi adaptée à la DHT hybride (arbre/anneau) de Pastry et l'arbre de Kademlia.

L'inondation Sym_Flood traverse l'arbre de Pastry en profondeur, en renvoyant la requête vers les voisins qui partagent un plus long préfixe avec l'identifiant ID de l'objet recherché, et ce, jusqu'à la localisation du réplica le plus proche. Concernant Kademlia, Sym_Flood apparaît comme une version plus générale de la méthode de recherche parallèle déjà employée. En effet, par défaut, un nœud Kademlia renvoie la requête à ses k voisins

partagent un plus long préfixe avec ID. L'inondation Sym_Flood dans Kademlia consiste à renvoyer cette requête à tous les voisins de plus long préfixe.

4.2.4 Evaluation de l'inondation Sym_Flood

Dans notre évaluation, nous fixons le degré de réplication à 4, suite aux résultats obtenus dans notre étude précédente (section 4.1.2.2). Nous évaluons l'approche Sym_Flood par émulations. Nous utilisons l'implémentation de Chord réalisés en C et nous implémentons en plus les différents mécanismes d'inondation plus la réplication symétrique.

Nous comparons les algorithmes suivants :

- Full flooding (Full_F) : désigne l'algorithme d'inondation de base (utilisé dans la première version de Gnutella par exemple). Dans cette approche, chaque nœud retransmet le message reçu à tous ses voisins (sauf l'expéditeur). L'opération se répète récursivement jusqu'à l'expiration du TTL ou la localisation de l'objet.
- Limited flooding (Lim_F) désigne l'algorithme d'inondation borné sans réplication. La recherche se fait donc dans tout l'espace des identités.
- Sym_Flood dénote l'algorithme d'inondation borné associé à la réplication symétrique de degré égale à 4. La recherche s'effectue dans le premier quart, le second…

4.2.4.1 Topologie, scenarii de trafic et critères de performances

Nous émulons un réseau de taille variable 100, 200, 300, 400 et 500 nœuds. Chaque nœud partage 10 objets avec un degré de réplication égal à 4. Le nombre de successeurs et de voisins *finger* est fixé à 8. Une fois le réseau stable, des nœuds rejoignent et quittent le réseau suivant une loi de poisson de moyenne 5 connexions/déconnexions par min. Parallèlement, 100 nœuds choisis aléatoirement envoient toutes les 10 secondes une requête vers une clé choisie aléatoirement, suivant une distribution uniforme. Enfin, la maintenance des réplicas symétriques s'effectue toutes les 60 secs.

Nous évaluerons les différentes approches d'inondation tenant compte des paramètres suivants : le taux de succès des requêtes, la longueur de la route, la signalisation totale du réseau et le nombre de tous les messages d'inondation. Ce dernier paramètre désigne le nombre de requêtes de recherche émises et retransmises dans le réseau.

4.2.4.2 Résultats

Longueur des routes et taux de succès

Les méthodes d'inondation Full_F et Lim_F présentent des résultats similaires en termes de nombre de sauts et taux de succès des recherches (Figure 4. 4 (a,b)). Cependant, le nombre de messages et la charge totale du réseau Full_F sont largement supérieurs. Dans un réseau de 500 nœuds par exemple, la méthode Lim_F produit 29% de messages et 35% de charge totale en moins (Figure 4. 4 (c,d)).

L'algorithme Sym_Flood produit, quant à lui, les meilleures performances. Le nombre de messages d'inondation est nettement réduit et le taux de succès dépasse 96%. La longueur des routes résultantes est comprise entre 2 et 3 sauts ; elle est en moyenne inférieure de 1 saut par rapport à Lim_F et Full_F soit 30% en moins.

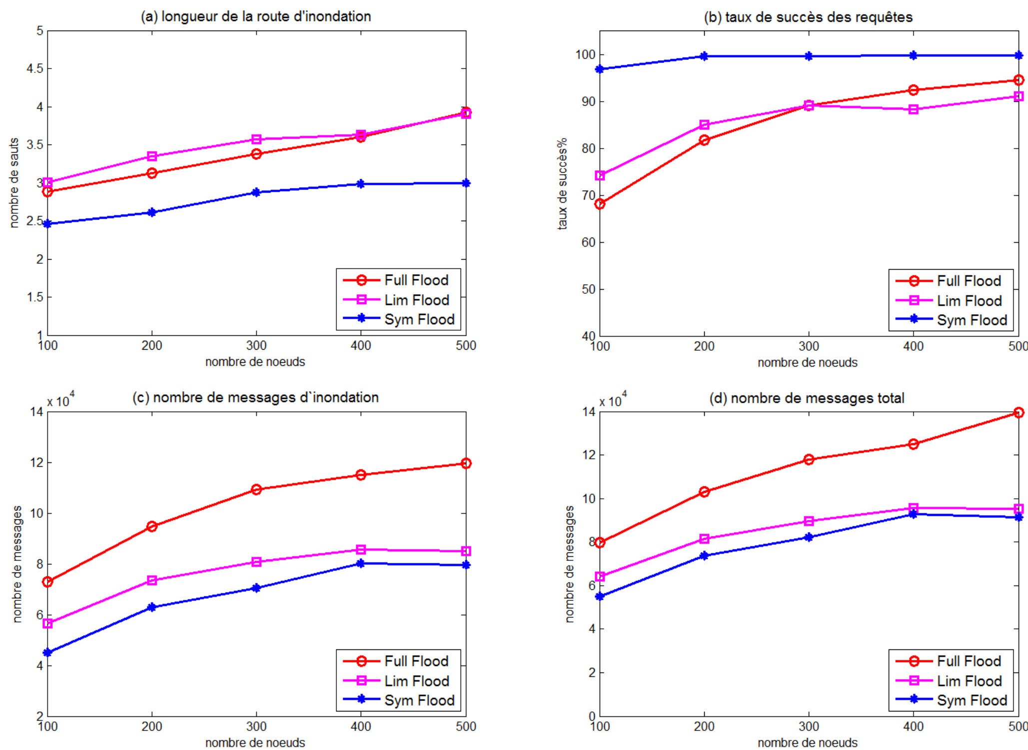


Figure 4. 4 Résultats des émulations des différentes méthodes d'inondation

Il aurait été possible de réduire le nombre de sauts en divisant davantage l'espace de recherche. Cependant, la maintenance d'un nombre plus élevé de réplicas augmenterait la charge totale du réseau (section 4.1.2.2).

Trafic de signalisation

La Figure 4. 4 (c) présente le nombre de messages d'inondation véhiculés dans le réseau. Nous remarquons que l'inondation Sym_Flood produit le plus petit nombre de messages. Comparée à Full_F et Lim_F, Sym_Flood génère respectivement 34% et 13% de messages en moins dans un réseau de 500 nœuds.

Un second résultat intéressant concerne la charge totale du réseau Sym_Flood. Bien que Sym_Flood emploie et maintient 4 réplicas, la charge totale du réseau reste inférieure à celle de l'inondation Lim_F (sans réplication). Le gain en nombre de messages d'inondation véhiculés dans Sym_Flood compense, par conséquent, l'augmentation du nombre total de messages de signalisation. De plus, nous remarquons que la charge totale du réseau Sym_Flood croît moins rapidement avec la taille du réseau. La segmentation de l'intervalle réduit l'espace de recherche et le contrôle du nombre de messages émis et réduisent de ce fait la charge totale.

En conclusion, l'utilisation de la réplication symétrique améliore la résistance du réseau face au *churn* d'une part, et réduit la signalisation et la longueur des routes d'autre part.

4.3 Conclusion

Suite à notre proposition Power_DHT, nous avons exploré dans ce chapitre quelques pistes d'optimisations possibles pour notre nouvelle structure. Nous avons choisi d'aborder la problématique de la réplication et de l'inondation. Ainsi, nous avons présenté une nouvelle méthode d'inondation, nommée Sym_Flood, qui emploie la réplication symétrique pour limiter l'espace des recherches et écourter les recherches.

Dans la première partie du chapitre, nous avons présenté une analyse évaluant les différentes méthodes de réplication employées dans les DHTs. Notre étude a montré l'avantage de la réplication symétrique notamment en termes de longueur des routes et de taux de succès des requêtes. A partir de ces résultats, nous avons choisi d'employer la réplication symétrique dans la suite de notre travail.

Dans la deuxième partie du chapitre, nous avons détaillé puis évalué notre proposition Sym_Flood appliquée à la DHT de Chord. Notre évaluation a montré que l'inondation Sym_Flood réduit le coût et la longueur des recherches. L'approche Sym_Flood s'avère donc une alternative intéressante pour effectuer des recherches par inondation dans les architectures structurées.

Par ailleurs, l'approche Sym_Flood peut être aussi appliquée à plusieurs structures dont Pastry et Kademlia (section 4.2.3). L'emploie de l'approche Sym_Flood dans Power_DHT semble donc une perspective intéressante à approfondir. Cela nous permettrait d'effectuer des recherches aléatoires dans une structure décentralisée à moindre coût.

Nous nous sommes limités dans cette partie à l'évaluation de la technique Sym_Flood dans des réseaux de 500 nœuds. Contrairement l'approche par simulation, l'émulation requiert beaucoup plus de ressources matérielles. Nous étions donc contraints de restreindre la taille du réseau dans notre évaluation. En revanche, il serait intéressant d'étudier dans un travail futur le comportement des mécanismes de réplication et de l'approche Sym_Flood dans des réseaux plus larges, pour vérifier le passage à l'échelle des techniques évaluées.

CONCLUSION GENERALE

Dans ce travail de thèse, nous nous sommes intéressés aux systèmes pair-à-pair de partage et distribution de contenu. Nous nous sommes concentrés principalement sur deux problématiques à savoir : comment interconnecter les pairs entre eux, et comment distribuer les données afin de permettre une recherche plus efficace ?

Pour analyser le problème de l'organisation des nœuds dans l'overlay, nous avons étudié les différentes propositions existantes telles que la hiérarchisation, ou la définition de super nœuds. Ces solutions existantes améliorent certes les performances du réseau, mais apportent à leur tour de nouveaux problèmes (section 2.3.1).

Nous avons traité cette problématique dans la première partie de ce document. Nous avons ainsi proposé Power_DHT. Power_DHT propose une nouvelle structure d'interconnexion et de routage, qui exploite l'hétérogénéité observé dans les DHTs déployées, en construisant dynamiquement une structure décentralisée non égalitaire, exhibant les propriétés d'un graphe sans échelle. Notre approche Power_DHT tire parti des propriétés de ces graphes, et permet ainsi l'intégration efficace du routage KBR et des méthodes de recherches aléatoires, à la fois.

L'approche Power_DHT constitue une alternative de choix pour la construction spontanée et dynamique d'architectures décentralisées et non hiérarchiques, basés sur les super nœuds. En ré-interconnectant le graphe de la DHT, se basant sur les informations locales déjà disponibles, Power_DHT conserve la structure de la DHT tout en décentralisant le graphe. La sélection et la maintenance des super nœuds se fait naturellement, émanant de l'architecture en loi de puissance résultante.

L'originalité de notre approche réside dans son efficacité et son faible coût, malgré sa simplicité : c'est la première méthode à construire dynamiquement une structure DHT décentralisée suivant une loi de puissance.

Pour valider notre approche, nous avons appliqué Power_DHT à trois DHTs différentes, à savoir : Chord, Pastry et Kademlia. L'application de Power_DHT aux différentes structures suit le même principe.

L'évaluation du routage KBR dans Power_DHT a montré une nette amélioration des performances notamment pour la DHT de Pastry et Chord. L'application de Power_DHT

à Kademlia s'avère être moins efficace. D'un autre côté, les méthodes de recherches aléatoires, appliquées à Power_DHT, réalisent un diamètre de recherche court, proche de celui des graphes en loi de puissance, mais produisent aussi une signalisation conséquente.

Dans la deuxième partie de notre travail, nous avons exploré quelques pistes auxiliaires, pour l'optimisation des performances de notre structure. Nous nous somme intéressés en particulier à l'organisation des données dans le réseau logique. Ainsi, nous avons proposé, dans le quatrième chapitre, Sym_Flood, une nouvelle technique de recherche aléatoire à faible coût, basée sur la réplication symétrique. Dans un premier temps, nous avons analysé et comparé les différentes méthodes de réplication proposées dans le cadre des DHTs. A partir de cette évaluation, nous avons choisi d'employer la réplication symétrique. Sym_Flood propose une nouvelle méthode de recherche aléatoire à moindre coût en exploitant la structure de la DHT et les propriétés de la réplication symétrique. L'évaluation de la méthode Sym_Flood, appliqué à Chord a montré un net avantage par rapport à l'algorithme de l'inondation classique en terme de coût (nombre de messages) et de longueur des recherches.

Comme nous l'avons mentionné, Sym_Flood peut être aussi appliqué à Pastry et Kademlia, suivant globalement le même principe. L'intégration de la recherche aléatoire par la technique Sym_Flood à Power_DHT semble donc une alternative intéressante pour réduire le coût des recherches par inondation, et constitue, par conséquent, une perspective directe de ce travail.

Une deuxième perspective principale de notre travail, consiste à faire une implémentation réelle de Power_DHT. L'approche Power_DHT permettrait de construire une DHT flexible pouvant osciller dynamiquement d'une structure DHT égalitaire vers une structure décentralisée et inversement, selon plusieurs métriques de configuration à définir (nombre de nœuds, densité du trafic, capacité des nœuds…). Partant de la DHT originale, nous construisons dynamiquement la structure décentralisée Power_DHT. Dans le sens inverse, on pourrait 'défaire' les ré-interconnexions de Power_DHT, pour revenir à la structure originale, et ce, en réinitialisant les tables des nœuds *reverse*. Dans notre implémentation, un nœud a le choix entre activer ou pas la fonction Power_DHT, en spécifiant une taille maximale à sa $reverse_{table}$ (qui détermine le nombre de connexions entrantes et donc la quantité de trafic qui transite par ce nœud). La fonction Power_DHT est activée si la taille de sa table est supérieure à zéro, et non activée sinon. Ainsi, un nœud de faible capacité

peut choisir de limiter la taille de sa $reverse_{table}$ (voire de la supprimer). A l'inverse, les nœuds de forte capacité optent pour des $reverse_{table}$ de taille importante. De la même façon, on pourrait choisir d'intégrer Power_DHT seulement à certains nœuds capables de supporter un grand nombre de connexions. En changeant simplement le comportement local de quelques nœuds, le réseau pourra évoluer dynamiquement vers une structure décentralisée permettant un routage et une recherche plus efficaces.

L'aspect sécurité reste aussi à explorer dans Power_DHT. Etant une structure décentralisée, divers problèmes de sécurités peuvent apparaitre tels que l'attaque par dénis de services distribués (DDoS) (144) ciblant les super nœuds ou l'attaque Eclipse qui vise principalement l'organisation des pairs et du voisinage dans les systèmes structurés (145). La pollution (146) de Power_DHT constitue aussi un problème délicat notamment lors de la réplication des objets. Cela causerait la propagation de données incorrectes, rendant le système inopérant.

PUBLICATIONS

1. Salma Ktari, Artur Hecker, Houda Labiod. Power-Law Chord Architecture in P2P Overlays. ***CoNext'2008****, Madrid Spain.*

2. Salma Ktari, Artur Hecker, Houda Labiod. Exploiting Routing Unfairness in DHT Overlays. *ISCC'2009, Sousse Tunisia.*

3. Salma Ktari, Artur Hecker, Houda Labiod. A construction scheme for scale free DHT based networks. ***Globecom'2009****, Hawaii USA.*

4. Salma Ktari, Artur Hecker, Houda Labiod, Exploiting Power-Law Node Degree Distribution in Chord Overlays. *NGI'2009, Avero Protugal.*

5. Salma Ktari, Artur Hecker, Houda Labiod. Empowering Chord DHT Overlays. *HPSR2009. Paris France.*

6. Salma Ktari, Mathieu Zoubert, Artur Hecker, Houda Labiod. Performance Evaluation of Replication Strategies in DHT under Churn. *MUM 2007, Oulu Finland.*

7. Salma Ktari, Mathieu Zoubert, Artur Hecker, Houda Labiod, Symmetric replication for efficient flooding in DHTs. ***Mobihoc'2008****, Honkong China.*

8. Salma Ktari, Artur Hecker, Houda Labiod. Structured flooding search in Chord Overlays. *GIIS2009, Hammamet Tunisia.*

9. Salma Ktari, JunHong Huang, Artur Hecker, Houda Labiod. 'Effet de la mobilité MANET sur un système P2P'. *CFIP2007, les arcs France*.

10. Salma Ktari, Artur Hecker, Houda Labiod, 'Exploiting Routing Unfairness in DHT Overlays', *Telecommunication system journal*.

BIBLIOGRAPHIE

1. **Gauron, Philippe.** Interconnexion et routage efficaces pour des procédures de recherche décentralisées dans les systèmes pair-à-pair. *rapport de thèse, Université paris sud 11, Septembre 2006.*

2. **Ipoque.** Internet Study 2007. *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.*

3. **Salma Ktari, Artur Hecker, Houda Labiod.** Power-Law Chord Architecture in P2P Overlays. *CoNext'2008, Madrid Spain.*

4. —. Exploiting Routing Unfairness in DHT Overlays. *ISCC'2009, Sousse Tunisia.*

5. —. A construction scheme for scale free DHT based networks. *Globecom'2009, Hawaii USA.*

6. **Salma Ktari, Artur Hecker, Houda Labiod,.** Exploiting Power-Law Node Degree Distribution in Chord Overlays. *NGI'2009, Avero Protugal.*

7. **Salma Ktari, Artur Hecker, Houda Labiod.** Empowering Chord DHT Overlays. *HPSR2009. Paris France.*

8. **Salma Ktari, Mathieu Zoubert, Artur Hecker, Houda Labiod.** Performance Evaluation of Replication Strategies in DHT under Churn. *MUM 2007, Oulu Finland.*

9. —. Symmetric replication for efficient flooding in DHTs. *Mobihoc2008, Honkong China.*

10. **Salma Ktari, Artur Hecker, Houda Labiod.** Structured flooding search in Chord Overlays. *GIIS2009, Hammamet Tunisia.*

11. **http://www.icq.com/.**

12. **http://www.skype.com.**

13. [En ligne] http://www.pplive.com/en/.

14. **http://setiathome.free.fr/.**

15. **Anthony D. Joseph, Ralf Steinmetz, Ion Stoica and Klaus Wehrle.** Peer-to-peer systems & applications. *Lecture notes in computer science, Vol. 3485.*

16. [En ligne] http://www.bittorrent.com/.

17. **John Kubiatowicz, Davic Bindel , David Bindel , Yan Chen , Steven Czerwinski , Patrick Eaton , Dennis Geels , Ramakrishna Gummadi , Sean Rhea , Hakim Weatherspoon , Westley Weimer , Chris Wells , Ben Zhao.** OceanStore: An Architecture for Global-Scale Persistent

Storage. *In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems.* 2000.

18. **Rowstron A, Druschel P.** Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *In Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles SOSP '01.* New York, NY, 2001.

19. **Courcelle, Bruno.** Introduction à la théorie des graphes :Définitions, applications et techniques de preuves. *rapport de recherche, Université Bordeaux 1, LaBRI (CNRS UMR 5800).* April 2004.

20. **Yang, B. and Garcia-Molina, H.** Comparing Hybrid Peer-to-Peer Systems. *In Proceedings of the 27th international Conference on Very Large Data Bases (September 11 - 14, 2001).*

21. **Olivier Dalle, Frédéric Giroire, Julian Monteiro, Stéphane Pérennes.** Analysis of Failure Correlation in Peer-to-Peer Storage Systems. *Rapport de recherche N°6771, INRIA December 2009.*

22. **Stefan Saroiu, P. Krishna Gummadi, Steven D. Gribble.** Measuring and analyzing the characteristics of Napster and Gnutella hosts. *Multimedia Syst, Aug. 2003.* Vol. 9.

23. **viennot, Laurent.** Autour du graphe et du routage. *rapport de thèse (HDR), Université paris 7, Novembre 2005.*

24. **Malek Rahoual, Patrick Siarry.** *Réseaux informatiques : conception et optimisation, 2006.*

25. **Qin Lv, Pei Cao, Edith Cohen, Kai Li, Scott Shenker.** Search and Replication in Unstructured Peer-to-Peer Networks. *Proceedings of the 16th international conference on Supercomputing.* 2002.

26. **Andrei Broder, Michael Mitzenmacher , Andrei Broder I Michael Mitzenmacher.** Network Applications of Bloom Filters: A Survey. *In Internet Mathematics.* 2002, Vol. 1.

27. **S. A. Baset, H. G. Schulzrinne.** An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. *Proceedings In INFOCOM 25th IEEE International Conference on Computer Communications.* 2006.

28. **v0.4, The Gnutella Protocol Specification.** Clip2. *Document Revision.* [En ligne] 2001. http://www.clip2.com/GnutellaProtocol04.pdf.

29. Gnutella 0.6. [En ligne] June 2002. http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html.

30. [En ligne] http://rfc-gnutella.sourceforge.net/Proposals/Ultrapeer/Ultrapeers.htm.

31. **Kirsch, Jonathan.** An Implementation and Analysis of the Skip Graph Data Structure . December 2003.

32. **Aspnes. J, Shah. G.** Skip graphs. *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, Baltimore, Maryland.* January 2003.

33. **Ion Stoica, Robert Morris , David Karger , M. Frans Kaashoek , Hari Balakrishnan.** Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *In Proceedings of SIGCOMM.* San Deigo, CA, 2001.

34. **Antony Rowstron, Peter Druschel.** Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science.* 2001.

35. **Petar Maymounkov, David Mazières.** Kademlia: A Peer-to-peer Information System Based on the XOR Metric. *in IPTPS.* 2002.

36. **Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Schenker, S.** A scalable content-addressable network. *In Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications (San Diego, California, United States). SIGCOMM '01. ACM, New York, NY.*

37. **T. Klingberg, R. Manfredi.** Gnutella 0.6. [En ligne] 2002. http://rfc-gnutella.sourceforge.net/draft.txt.

38. **Saikat Guha, Neil Daswani , Ravi Jain.** An Experimental Study of the Skype Peer-to-Peer VoIP System. *In IPTPS'06: The 5th International Workshop on Peer-to-Peer Systems.* 2006.

39. **Ghodsi, Ali.** *Distributed k-ary System: Algorithms for Distributed Hash Tables.* s.l. : PhD dissertation KTH-Royal Institute of Technology, October 2006.

40. **Dabek F, Kaashoek M F, Karger D, Morris R, Stoica I.** Wide-area cooperative storage with CFS. *In Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles SOSP '01.* New York, NY, October 2001.

41. **Russ Cox, Athicha Muthitacharoen, Robert T. Morris.** Serving DNS using a Peer-to-Peer Lookup Service. *International Workshop on Peer-To-Peer Systems.* 2002.

42. **Greenfield, David.** SIP Goes Peer-to-Peer. *Network Magazine.* May 2005.

43. **J. Maenpaa, J. Maenpaa, G. Camarillo.** A Self-tuning Distributed Hash Table (DHT) for REsource LOcation And Discovery (RELOAD). *Internet draft.* February 2009.

44. **Ion Stoica, Robert Morris , David Karger , M. Frans Kaashoek , Hari Balakrishnan.** Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications.* San Diego, California, United States, 2001.

45. **C. G. Plaxton, R. Rajaraman, A.W. Richa.** Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems.* 1999.

46. **Ben Y. Zhao, John Kubiatowicz, Anthony D. Joseph,.** Tapestry: An Infrastructure for fault-tolerant Wide-area Location and Routing. *Technical report,.* 2001.

47. **Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, Harlan Yu.** OpenDHT: A Public DHT Service and Its Uses. *Proceedings of ACM SIGCOMM.* August 2005.

48. **Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, Peter Drusche.** SCRIBE: The design of a large-scale event notification infrastructure. *in Proceedings of the Third International Workshop on Networked Group Communications (NGC2001).* London, UK, November 2001.

49. **Moritz Steiner, Taoufik En-Najjary, Ernst W. Biersack.** Exploiting KAD: possible uses and misuses. *ACM SIGCOMM Computer Communication.* October 2007, Vol. 37.

50. [En ligne] http://www.exeem.com/.

51. **Ingmar Baumgart, Sebastian Mies.** S/Kademlia: A Practicable Approach Towards Secure Key-Based Routing. *Proceedings of the 13th International Conference on Parallel and Distributed Systems (ICPADS '07).* Hsinchu, Taiwan, 2007.

52. **Dahlia Malkhi, Moni Naory, David Ratajczak.** Viceroy: A Scalable and Dynamic Emulation of the Butterfly. *Proceedings of the twenty-first annual symposium on Principles of distributed computing, Monterey, California.* 2002.

53. **Datar, M.** repose sur un graphe en papillon . *Proceedings of the 10th Annual European Symposium on Algorithms - ESA'02, 2002.*

54. **Abhishek Kumar, Shashidhar Merugu , Jun (Jim) Xu , Xingxing Yu.** Ulysses: A Robust, Low-Diameter, Low-Latency Peer-to-Peer Network. *11th IEEE International Conference on Network Protocols (ICNP'03), Atlanta, Georgia.* 2003.

55. **de Bruijn, N. G.** A combinatorial problem. *In Koninklijke Nederlands Akademie van Wetenschappen Proceedings.* 1946.

56. **Peyrat, C. Bermond and C.** De bruijn and kautz networks: a competitor for the hypercube? *in Proc. of the 1st Europ. Workshop on Hypercubes and Distributed Computers.*

57. **Gai, Anh Tuan Viennot, Laurent.** Broose: A Practical Distributed Hashtable Based on the De-Brujin Topology. *Rapport de recherche de l'INRIA RR-5238.* Juin 2004.

58. **Gauron, Philippe.** 'Interconnexion et routage efficaces pour les procédures de recherche décentralisées dans les systèmes pair à pair'. *rapport de thèse, Université Paris Sud - Paris XI.* 2006.

59. **Marin Bertier, Anne-Marie Kermarrec, Vincent Leroy and Sathya Peri.** On the Fly DHT using Gossip Protocols. *8th International Conference on Peer-to-Peer Computing 2008.*

60. **G. Manku, M. Bawa, P. Raghavan.** Symphony: Distributed hashing in a small world. *USENIX Symposium on Internet Technologies and Systems.* 2003.

61. **Kleinberg., J.** *The small-world phenomenon: an algorithmic.* s.l. : cornell computer science technical report 99-1776., 2000.

62. **K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, I. Stoica.** The impact of DHT routing geometry on resilience and proximity. *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications.* August 2003.

63. **Loguinov, D., Kumar, A., Rai, V., and Ganesh, S.** Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, SIGCOM'03, New York.*

64. **J. Li, Stribling J., T.M. Gil, R.Morris, andM. F. Kaashoek.** Comparing the performance of distributed hash tables under churn. *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS’04).*

65. **D. Liben-Nowell, H. Balakrishnan, and D. Karger.** Analysis of the evolution of peer-to-peer systems. *In Proceedings of the 21st annual symposium on Principles Of Distributed Computing - PODC’02.*

66. **K. Gummadi, R. Gummadiy, S. Gribblez, S. Ratnasamyx, S. Shenker, I. Stoica.** The Impact of DHT Routing Geometry on Resilience and Proximity. *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications.* Karlsruhe, Germany 2003.

67. **Chawathe. Y, Ratnasamy. S, Breslau. L, Lanham. N, Shenker. S.** Making gnutella-like P2P systems scalable. *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications.* Karlsruhe, Germany, 2003.

68. **Garcés-Erice, Luis.** *Un Réseau P2P Hiérarchique:Design et Applications.* s.l. : Rapport de thèse, Telecom ParisTech.

69. **Liu Hui-shan, Xu Ke, Xu Ming-wei, Cui Yong,.** S-Chord: Hybrid Topology Makes Chord Efficient. *Networking - ICN .* 2005, Vol. 3421/2005.

70. **Miguel Castro, Manuel Costa, Antony Rowstron.** Debunking some myths about structured and unstructured overlays. *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation.* 2005.

71. **Stefan Zöls, Rüdiger Schollmeier.** The Hybrid Chord Protocol: A Peer-to-Peer Lookup Service for Context-Aware Mobile Applications. *Networking - ICN 2005.* Vol. 3421/2005.

72. **Yatin Chawathe, Sylvia Ratnasamy, Lee.** GIA: Making Gnutella-like P2P Systems Scalable. *SIGCOM.* 2003.

73. **Georgios P, Panayiotis P, Lindsay M.** A Policy for Electing Super Nodes in Unstructured P2P Networks. *AP2PC.* 2004.

74. **Y. Zhu, H. Wang, Y. Hu,.** A super-peer based lookup in structured peer-to-peer systems. *In Proceedings of the 16th International Conference on Parallel and Distributed Computing Systems (PDCS).* Reno, Nevada, Aug. 2003.

75. **Yuhong Liu, Chris Gauthierdickey, Jun Li.** Scalable supernode selection in peer-to-peer overlay networks. *In Proceedings of the 2nd International Workshop on Hot Topics in Peer-to-Peer Systems.* July 2005.

76. **Li. Z, Feng. Z.** *Understanding Chord Performance and Topology aware Topologyaware.* s.l. : http://www.cs.berkley.edu/~zl/doc/chord_perf.pdf, Project Report, 2003.

77. **Mesaros. V, Carton. B, Van Roy. P.** "S-Chord: Using Symmetry to Improve Lookup Efficiency in Chord. *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications,.* USA, June 2003.

78. **Junjie Jiang, Ruoyu Pan, Changyong Liang, Weinong Wang.** BiChord: An Improved Approach for Lookup Routing in Chord. *Lecture Notes in Computer Science.* Springer Berlin / Heidelberg, Vol. 3631/2005.

79. **Haitao Dong, Weimin Zheng, Dongsheng Wang.** Janus: Build Gnutella-Like File Sharing System over Structured Overlay. *Lecture Notes in Computer Science.* 2004, Vol. 3251.

80. [En ligne] http://wua.la.

81. **Hakim Weatherspoon, John D. Kubiatowicz.** Erasure Coding vs. Replication: A Quantitative Comparison. *Lecture Notes in Computer Science.* 2002, Vol. 2429.

82. **Rodrigo Rodrigues, Barbara Liskov.** High Availability in DHTs: Erasure Coding vs. Replication. *Lecture Notes in Computer Science.* 2005, Vol. 3640.

83. **J. Risson, T. Moors.** Survey of Research towards Robust Peer-to-Peer Networks: Search Methods. *Computer Networks: The International Journal of Computer and Telecommunications Networking.* December 2006, Vol. 50.

84. **Miguel Castro, Manuel Costa and Antony Rowstron.** Should we build Gnutella on a structured overlay. *ACM SIGCOMM Computer Communication Review.* January 2004.

85. **El-Ansary, Sameh and Brand, Per and Haridi, Seif, et al.** Efficient broadcast in structured P2P networks. *In: 2nd International Workshop On Peer-To-Peer Systems (IPTPS).* Berkeley, CA, USA, Feb 2003.

86. **Patrick Reynolds, Amin Vahdat.** Efficient Peer-to-Peer Keyword Searching. *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware.* 2003.

87. **Keshav, M. zaharia S.** Gossip-based search selection in hybrid peer to peer networks. *in Proceeding of IPTPS.* 2006.

88. **B. T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, I. Stoica.** Enhancing P2P File-Sharing with an Internet Scale Query Processor. *In Proceedings of VLDB.* Sep 2004.

89. **Nicolas Bonnel, Fabienne Moreau.** *Quel avenir pour les moteurs de recherche ?* s.l. : MajecSTIC 2005 : Manifestation des Jeunes Chercheurs francophones dans les domaines des STIC , 2005.

90. **Frans Kaashoek, Omprakash D. Gnawali , Omprakash D Gnawali.** *A Keyword-Set Search System for Peer-to-Peer Networks.* s.l. : Master's thesis, MIT, June 2002.

91. **Kannan. J, Yang. B, Shenker. S, Sharma. P.** SmartSeer: Using a DHT to Process Continuous Queries Over Peer-to-Peer Networks. *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings.* Barcelona, Spain, April 2006.

92. **Hui Zhang, Ashish Goel, and Ramesh Govindan.** Improving Lookup Latency in Distributed Hash Table Systems Using Random Sampling. *IEEE/ACM Transactions on Networking (TON) .* 2005, Vol. 13.

93. **Salma Ktari, JunHong Huang, Artur Hecker, Houda Labiod.** Effet de la mobilité MANET sur un système P2P. *CFIP2007, les arcs France. .*

94. **A.Qayyum, L. Viennot,and A.Laouiti.** Multipoint Relaying: An Efficient Technique for flooding in Mobile Wireless Networks. . *Research Report RR-3898, INRIA, February 2000.*

95. **Fuhrmann, Thomas.** Combining Virtual and Physical Structures for Self-organized Routing. *IWSOS/EuroNGI 2006.*

96. —. Scalable Routing for Networked Sensors and Actuators. *Proceedings of the Second Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, Santa Clara, California, September 26-29, 2005 .*

97. **Schiller, Thomas Zahn and Jochen.** MADPastry: A DHT Substrate for Practicably Sized MANETs. *In Proc. of 5th Workshop on Applications and Services in Wireless Networks (ASWN2005) (June 2005).*

98. **Schollmeier R., Gruber I., Finkenzeller M.** Routing in Mobile Ad Hoc and Peer to-Peer networks. A comparison. *Lecture Notes in Computer Science, Vol. 2376, p. 172-186 (2003).*

99. **Matthew Caesar, Miguel Castro, Edmund B. Nightingale, Greg O'Shea, Antony Rowstron.** Virtual Ring Routing: Network Routing Inspired by DHTs. *In SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications.*

100. **Emmanuel Baccelli, Thomas Zahn , Jochen Schiller.** DHT-OLSR. *rapport de recherche INRIA-00148347:2.*

101. **Hu, Himabindu Pucha Saumitra M. Das and Y. Charlie.** Ekta: An Efficient DHT Substrate for Distributed Applications in Mobile Ad Hoc Networks. *in Proceedings of the 6th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2004).*

102. **Y. Charlie Hu, Saumitra M. Das, and Himabindu Pucha.** Exploiting the Synergy between Peer-to-Peer and Mobile Ad Hoc Networks. *Proceedings of the 9th Conference on Hot Topics in Operating Systems, Hawaii, 2003).*

103. **R. Cuevas, Manuel Urueña, Albert Banchs.** Routing fairness in Chord: Analysis and Enhancement. *InfoCom 2009.* Rio de Janeiro, 2009.

104. **Keno Albrecht, Ruedi Arnold, Michael Gahwiler, Roger Wattenhofer.** Aggregating information in Peer-to-Peer Systems for Improved Join and Leave. *Proceedings of the Fourth International Conference on Peer-to-Peer Computing.* 2004.

105. **Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, Ion Stoica,.** Load Balancing in Dynamic Structured P2P Systems. *Performance Evaluation.* Mar. 2006, Vol. 63.

106. **Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, Ion Stoica,.** Load Balancing in Structured P2P Systems. 2003.

107. **Jussi Kangasharju, Keith W. Ross, David A. Turner.** Adaptive Content Management in Structured P2P Communities. *Proceedings of the 1st international conference on Scalable information systems.* Hong Kong, 2006.

108. **S. Serbu, Silvia Bianchi , Peter Kropf , Pascal Felber.** Dynamic Load Sharing in Peer-to-Peer Systems: When Some Peers Are More Equal than Others. *IEEE Internet Computing.* Los Alamitos, CA, USA, 2007, Vol. 11, 4.

109. **Ruben Cuevas Rumín, Manuel Uruena.** Routing fairness in Chord: Analysis and Enhancement. *28th IEEE Conference on Computer Computer Communications - INFOCOM.* 2009.

110. [En ligne] http://www.omnetpp.org/.

111. **Ingmar Baumgart, Bernhard Heep , Stephan Krause.** http://www.oversim.org/. [En ligne]

112. **Ingmar Baumgart, Bernhard Heep, Stephan Krause.** OverSim: A Flexible Overlay Network Simulation Framework. *Proceedings of 10th IEEE Global Internet Symposium.* Anchorage, AK, USA, 2007.

113. **Waltz, Edward.** Understanding Confidence Intervals. *Support de cours.*

114. **Jiang, J. Pan, R. Liang, C. Wang, W.** BiChord: An Improved Approach for Lookup Routing in Chord. *LECTURE NOTES IN COMPUTER SCIENCE.* Germany 2005.

115. **Castro Miguel, Costa Manuel , Rowstron Antony.** Debunking some myths about structured and unstructured overlays. *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation.* 2005.

116. *Emergence of scaling in random networks.* **Albert-László Barabási, Réka Albert.** s.l. : Science, Vol. 286.

117. **Faloutsos M, Faloutsos P, Faloutsos C.** On power-law relationships of the Internet topology. *SIGCOMM '99 In Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols For Computer Communication.* 1999.

118. **Adamic, Lada A.** The Small World Web. *Proceedings of the Third European Conference on Research and Advanced Technology for Digital Libraries.* 1999.

119. **Andrei Broder, Ravi Kumar, Farzin Maghoul.** Graph Structure of the Web: A Survey. *Computer Networks.* June 2000, Vol. 33.

120. **Jean-Loup Guillaume, Matthieu Latapy, and Clémence Magnien.** Comparison of Failures and Attacks on Random and Scale-Free Networks. *OPODIS : international conference on principles of distributed systems.* Grenoble , FRANCE, 2005.

121. **Watts, Duncan J.** Small Worlds: The Dynamics of Networks between Order and Randomness. 2003.

122. *Scale-Free Networks Are Ultrasmall.* **Cohen Reuven, Havlin Shlomo.** s.l. : Phys. Rev. Lett., 2003, Vol. 90.

123. **Beom J. Kim, Chang N. Yoon, Seung K. Han, Hawoong Jeong.** Path finding strategies in scale-free networks. *Physical Review E.* Jan 2002, Vol. 65.

124. **Adamic, Lada A. and Lukose, Rajan M. and Puniyani, Amit R. and Huberman, Bernardo A.** Search in power-law networks. *Phys. Rev. E.* Sep 2001, Vol. 64.

125. **Nima Sarshar, P. Oscar Boykin , Vwani P. Roychowdhury.** Percolation Search in Power Law Networks: Making Unstructured Peer-To-Peer Networks Scalable. *In Proceedings of IEEE P2P’04.* 2004.

126. **Masahiro Sasabe, Naoki Wakamiya , Masayuki Murata.** LLR: A Construction Scheme of a Low-iameter, Location-Aware, and Resilient P2P Network. *International Conference on Collaborative Computing: Networking, Applications and Worksharing.* Atlanta, GA, 2006.

127. **Rita H. Wouhaybi, and Andrew T. Campbell.** Phenix: Supporting Resilient Low-Diameter Peer-to-Peer Topologies. *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies.* 2004.

128. **Matei Ripeanu, Ian Foster , Adriana Iamnitchi.** Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal .* 2002.

129. **Lu Liu, Jie Xu , Duncan Russell , Paul Townend , David Webster.** Efficient and scalable search on scale-free P2P networks . *Peer-to-Peer Networking and Applications.* Springer New York, 2009.

130. **Antony Rowstron, Peter Druschel,.** Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms.* Heidelberg, November 2001.

131. **Petar Maymounkov, David Mazières.** Kademlia: A Peer-to-peer Information System Based on the XOR Metric. *In IPTPS.* 2002.

132. **Rubén Mondéjar, Pedro García y Carles Pairot.** Bunshin: DHT para aplicaciones distribuidas. Granada, Spain, September 2005.

133. **Sylvia Ratnasamy, Paul Francis , Mark Handley , Scott Shenker.** A Scalable Content-Addressable Network. *ACM SIGCOMM.* 2001.

134. **Predrag Knezevic, Andreas Wombacher, Thomas Risse.** Enabling High Data Availability in a DHT. *in 2nd International Workshop on Grid and Peer-to-peer Computing Impacts on Large Scale Heterogeneous Distributed Database Systems.* 2005.

135. —. DHT-based Self-adapting Replication Protocol for Achieving High Data Availability. *in The InternationalConference on Signal-image Technology and Internetbased Systems (SITIS).* 2006.

136. **Qin Lv, Pei Cao, Edith Cohen, Kai Li, Scott Shenker.** Search and Replication in Unstructured Peer-to-Peer Networks. *In Proceedings of the 16th annual ACM International Conference on supercomputing.* 2002.

137. **Ben Y. Zhao, John Kubiatowicz, Anthony D. Joseph.** *Tapestry: An Infrastructure for fault-tolerant Wide-area Location and Routing.* UC Berkeley : s.n., 2001.

138. **Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, John D. Kubiatowicz.** *Bayeux : An Architecture for Scalable and Fault tolerant Widearea Data Dissemination.* s.l. : In

Proceedings of the 11th international lworkshop on Network and operating systems support for digital audio and video, 2001.

139. **Matthew Leslie, Jim Davies, Todd Huffman.** A Comparison Of Replication Strategies for Reliable Decentralised Storage. *Proceedings of the The First International Conference on Availability, Reliability and Security, ARES 2006, Austria.*

140. **Ali Ghodsi, Luc Onana Alima, Seif Haridi.** Symmetric Replication for Structured Peer-to-Peer Systems. *In: The 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing, July 2005, Trondheim, Norway.*

141. **Emil Sit, Andreas Haeberlen, Frank Dabek, ByungGon Chun, Hakim Weatherspoon.** Proactive replication for data durability. *In Proceedings of the 2007 ACM CoNEXT Conference, New York, December 2007).*

142. **El-Ansary Sameh, Brand Per, Haridi Seif.** Efficient broadcast in structured P2P networks. *In: 2nd International Workshop On Peer-To-Peer Systems (IPTPS).* Berkeley, CA, USA, 2003.

143. **P. Trunfio, D. Talia, A. Ghodsi, S. Haridi.** *Implementing Dynamic Querying Search in k-ary DHT-based Overlays.* s.l. : In: Grid Computing: Achievements and Prospects, S. Gorlatch, P. Fragopoulou, T. Priol (Editors), Springer, USA, pp. 275--286, 2008. , 2008.

144. **MARLIER, Patrick.** sécurité du peer to peer. [En ligne] www.labo-asso.com.

145. **A. Singh, M. Castro, A. Rowstron, P. Druschel,.** Defending against eclipse attacks on overlay networks. *Proceedings of the 11th ACM SIGOPS European Workshop, Leuven, Belgium, ACM Press, New York (2004) .*

146. **R, Sit. E and Morris.** Security Considerations for Peer-to-Peer Distributed Hash Tables. *In Revised Papers From the First international Workshop on Peer-To-Peer Systems (March 2002).*

147. **Erice, Louis Garces.** *Un réseau P2P hiérarchique : Design et Applications.* s.l. : École nationale supérieure des télécommunications, 2005.

148. *On hierarchical DHT systems - An analytical approach for optimal designs.* **Stefan Zoels, Zoran Despotovic, Wolfgang Kellerer.** s.l. : Computer Communications, 2008, Vol. 31.

149. *A policy for electing super-nodes in unstructured P2P networks.* **PITSILIS Georgios, PERIORELLIS Panayiotis , MARSHALL Lindsay ,.** s.l. : Lecture notes in computer science, 2004, Vol. 3601/2005.

150. **Virginia Lo, Dayi Zhou , Yuhong Liu , Chris Gauthierdickey , Jun Li.** Scalable Supernode Selection in Peer-to-Peer Overlay Networks. *In Proceedings of the 2nd International Workshop on Hot Topics in Peer-to-Peer Systems.* 2005.

151. *giFT-FastTrack home page.* [En ligne] http://developer.berlios.de/projects/gift-fasttrack/.

152. *An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol.* **S. A. Baset, H. G. Schulzrinne.** s.l. : 25th IEEE International Conference on Computer Communications. INFOCOM, 2006.

153. **Arturo Crespo, Hector Garcia-Molina.** *Routing indices for peer-to-peer systems.* Stanford University, CS Department : Technical Report , November 2001.

154. **Schollmeier, R.** A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. *First International Conference on Peer-to-Peer Computing (P2P'01) Sweden.* August 2001.

155. *Search and Replication in Unstructured Peer-to-Peer Networks.* **Qin Lv, Pei Cao, Edith Cohen, Kai Li, Scott Shenker.** s.l. : Proceedings of the 16th international conference on Supercomputing, 2002.

156. **Thomas Fuhrmann, Pengfei Di, Kendy Kutzner, and Curt Cramer.** Pushing Chord into the Underlay: Scalable Routing for Hybrid MANETs. *Universität Karlsruhe (TH), Fakultät für Informatik, Technical Report 2006-12, June 2006 .*

157. **Delmastro, Franca.** From Pastry to CrossROAD: CROSS-layer Ring Overlay for AD hoc networks. *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications Workshops.*

158. **Qi Meng, Hong Ji.** MA-Chord: A New Approach for Mobile Ad Hoc Network with DHT Based Unicast Scheme. *n: Wireless Communications, Networking and Mobile Computing, WiCom, Shanghai 2007.*

159. **George Porter, Kevin Lai, Ion Stoica, Jeremy Condit.** The Chord Ad-hoc Routing Protocol. *technical report, 2007-08-12.*