Université de Montréal

Équilibrage de Charge dans un Réseau Pair-à-Pair Structuré : une Méthode Dynamique Intégrée

par

Viet Dung LE

Département d'Informatique et de Recherche Opérationnelle
Faculté des Arts et des Sciences

Thèse présentée à la Faculté des Études Supérieures
en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en informatique

octobre 2006

Université de Montréal
Faculté des études supérieures

Cette thèse intitulée :

Équilibrage de Charge dans un Réseau Pair-à-Pair Structuré : une Méthode Dynamique Intégrée

présentée par :

Viet Dung LE

a été évaluée par un jury composé des personnes suivantes :

Jean VAUCHER
président-rapporteur

Peter KROPF
directeur de recherche

Gilbert BABIN
codirecteur

Abdelhakim HAFID
membre du jury

Pierre FRAIGNIAUD
examinateur externe

Vincent GAUTRAIS
représentant du doyen de la FES

Résumé

Le terme *pair-à-pair* indique un type de système distribué dans lequel les participants, appelés pairs, peuvent demander et fournir des services. Aucun pair n'est capable de connaître ou contrôler tout le système. Le manque de contrôle centralisé cause des difficultés pour le routage et la recherche d'objets. Les systèmes pair-à-pair structurés résolvent ces problèmes en définissant une structure de connexion entre les pairs. La connexion structurée limite le coût de routage (souvent à un nombre logarithmique de sauts) mais complique la maintenance du réseau.

Entre autres, l'équilibrage de charge est un problème critique des systèmes pair-à-pair structurés. L'équilibrage de charge implique des mécanismes visant à distribuer la charge sur les pairs de manière dynamique pour assurer la performance du système. Nous nous intéressons à deux charges importantes : charge de stockage et charge de gestion d'index. Tandis que la première charge dénote l'utilisation des ressources pour le stockage des objets, la deuxième charge dénote l'utilisation de la bande passante pour le trafic de routage. Notre contribution comporte des mécanismes d'équilibrage intégré de ces charges dans un système pair-à-pair structuré dont la topologie s'inspire des graphes de De Bruijn.

L'application des graphes de De Bruijn permet au système de fournir un routage efficace et des mécanismes de maintenance à bas coût. L'équilibrage intégré de charge de stockage et de charge de gestion d'index est possible grâce à la séparation entre le stockage des objets, la responsabilité des clés et l'identification des pairs. L'évaluation basée sur des simulations nous confirme la performance du système pair-à-pair et de l'équilibrage intégré des charges.

Mots clés : système pair-à-pair structuré, équilibrage intégré des charges, charge de stockage et charge de gestion d'index.

Abstract

The term *peer-to-peer* indicates a type of distributed system that allows its participants, called peers, to both access and provide services. No participant can know or control the whole system. The lack of a centralized control introduces difficulties for routing and searching. Structured peer-to-peer systems solve this problem by imposing a structure on the connection of the peers. The structured connection scheme ensures efficient routing (which usually involves a logarithmic number of hops) but complicates network maintenance.

Load balancing is a critical problem in structured peer-to-peer systems. It implies methods that aim to dynamically distribute the load on the peers for maintaining the system's performance. We are interested in two types of load : storage load and index management load. While the first denotes the usage of computational resources in object storage, the second specifies the consumption of network bandwidth for the routing traffic. Our contribution consists in integrating balancing methods for these loads in a structured peer-to-peer system whose construction is inspired by de Bruijn graphs.

The application of de Bruijn graphs allows the peer-to-peer system to achieve efficient routing and low-cost maintenance. The system enables integrated balancing of the storage load and the index management load by separating object location, key responsibility, and peer identifier. The evaluation based on simulations confirms the performance of our peer-to-peer system and the integrated load balancing methods.

Keywords : structured peer-to-peer system, integrated load balancing, storage load, and index management load.

Table des matières

Liste des tableaux

Table des figures

Liste des abréviations

ANOVA Analysis Of Variance.

API Application Programming Interface.

BALLS Balanced Load Supported P2P Structure.

BISON Biology-Inspired techniques for Self-Organization in dynamic Networks.

CAN Content-Addressable Network.

CFS Cooperative File System.

CPU Central Processing Unit.

DH DHT Distance Halving Distributed Hash Table.

DHT Distributed Hash Table.

DNS Domain Name System.

id Identifier (ou Identificateur).

IEEE Institute of Electrical and Electronics Engineers.

IP Internet Protocol.

Java VM Java Virtual Machine.

OS Operating System.

P2P Peer-to-Peer (ou Pair-à-Pair).

RAM Random-Access Memory.

SETI Search for ExtraTerrestrial Intelligence.

SQL Structured Query Language.

TCP Transmission Control Protocol.

TTL Time-To-Live.

Remerciements

Pour achever la présente thèse, j'ai reçu beaucoup de l'aide précieuse. J'aimerais remercier les Professeurs Peter Kropf de l'Université de Neuchâtel et de l'Université de Montréal (UdM) et Gilbert Babin de HEC Montréal, mes directeurs de recherche, qui m'ont financé et donné beacoup de l'aide au long de la recherche. Je remercie le Professeur Sang Nguyen de l'UdM. Ses financement, support et conseils m'ont beaucoup aidé à vaincre mes difficultés pendant les études.

Je remercie le Professeur Jean Vaucher de l'UdM et Monsieur Arnaud Dury du CRIM qui m'ont donné des idées précieuses pour l'identification du problème de recherche. J'envoie un merci à la Professeure Brigitte Jaumard de l'Université Concordia. L'acceptation à son laboratoire (au CRT) m'a permis d'accélérer les expériences. Je remercie aussi le Professeur Denis Larocque de HEC Montréal qui m'a guidé dans la réalisation des analyses statistiques sur les résultats d'expérience.

Il me faut remercier les Professeurs Lorne Bouchard et Marc Bouisset de l'UQAM, qui sont aussi les anciens directeur et directeur des études de l'Institut de la Francophonie pour l'Informatique (IFI) au Vietnam. Ils m'ont donné beaucoup de l'aide et leur support inoubliable avant de venir et pendant mes séjours au Canada. Je remercie l'IFI et les Professeurs Nguyen Dinh Tri, Nguyen Thi Hoang Lan et Nguyen Thanh Thuy de leur support pour que je vienne étudier au Canada.

Je tiens à remercier mes collègues du laboratoire Téléinformatique du Département d'Informatique et de Recherche Opérationnelle de l'UdM, à mes amis, les anciens étudiants de l'IFI au Québec. Ils m'ont donné de l'aide généreuse dans la recherche et les études.

Enfin, je réserve un grand merci à mes parents, grand-parents, mon épouse, ma fille, mon oncle et les autres membres de ma famille. Leur support a été une contribution indispensable à mon succès.

Chapitre 1

Introduction

Le paradigme du calcul distribué pair-à-pair (P2P) a énormément attiré la recherche récente. Le présent document synthétise notre contribution à cette recherche. Ce chapitre présente la motivation, l'objectif et un survol de notre travail.

1.1 Motivation

Un système P2P représente un réseau abstrait dont les nœuds (pairs) sont équivalents en fonctionalité ; c'est-à-dire, ils peuvent à la fois demander et fournir des services. Chaque pair établit une connexion avec un ensemble d'autres pairs et il n'y a pas de contrôle central ni hiérarchique de tout le système. Cette décentralisation entraîne un avantage primordial de P2P : l'agrégation des capacités des pairs pour augmenter la performance globale du système. Par ailleurs, un système P2P permet les entrées et sorties dynamiques des pairs.

Grâce à ces propriétés, le paradigme P2P présente une bonne solution pour les calculs intensifs et à grande échelle. Il est largement appliqué dans des systèmes de partage de fichiers, par exemple, Gnutella [gnu01], KaZaA [kaz05, LKR05], eDonkey - Overnet [edo03],

BitTorrent [bit06] et Freenet [fre06]. Un système de partage de fichiers crée un stockage immense dans lequel les usagers peuvent publier et télécharger des fichiers, particulièrement de la musique, des vidéos et des e-books. D'ailleurs, l'application de P2P se fait dans des services de transfert d'information, par exemple, le service de diffusion de données de média PeerCast [pee06] et le service de téléphonie P2P Skype [sky06, GDJ06]. Dans de tels systèmes, des ordinateurs participants contribuent leurs ressources au stockage et à l'échange des données.

En comparaison avec les systèmes classiques du type client/serveur, le routage est plus difficile dans les systèmes P2P. Sans connaissance globale, le routage pour une destination doit traverser des pairs intermédiaires selon un certain algorithme. En fonction de l'algorithme de routage, on catégorise les systèmes P2P en systèmes P2P non-structurés et systèmes P2P structurés. Les systèmes P2P non-structurés (p.ex., Gnutella et Freenet) ne définissent pas de structure de connexion entre les pairs. Le routage de Gnutella cherche la destination par la diffusion de message. Chaque pair sur le chemin d'un message le transfère à tous les voisins (sauf le pair précédent et le pair d'origine, le cas échéant). La diffusion s'arrête à un pair si le pair est la destination ou le *time-to-live* (TTL) défini pour le message expire. Le routage de Freenet utilise l'algorithme de recherche en profondeur. La décision de transfert de message dépend de l'information connue par le pair courant sur le chemin. Freenet définit aussi un champs TTL dans le message pour limiter le nombre de sauts. En général, les systèmes P2P non-structurés emploient des algorithmes de routage simples. Comme la connexion ne suit pas de structure fixe, cette catégorie demande peu de maintenance (c.-à-d., du support pour l'arrivée et le départ des pairs). Cependent, les algorithmes de routage de ces systèmes ne sont pas déterministes ; c'est-à-dire, un routage pourrait ne pas atteindre la destination malgré son existence. De plus, le routage d'un système comme Gnutella se fait à un coût de trafic élevé à cause de la technique de diffusion utilisée.

Les systèmes P2P structurés, dont des exemples sont Tapestry [HKRZ02, ZHS+04],

Chord [SMK+01], CAN [RFH+01], Pastry [RD01a], Viceroy [MNR02], Koorde [KK03], D2B [FG03], Overlapping DHT [NW03b], maintiennent une structure de connexion entre les pairs. Un tel système distribue un espace de clés sur les pairs disponibles. La structure de connexion définit les liens entre les pairs en fonction de l'ensemble des clés dont chaque pair est responsable. Cette connexion structurée augmente la complexité de maintenance. Cependant, elle apporte le routage déterministe et efficace. Le routage d'un système P2P structuré nécessite un petit nombre de sauts, souvent $O(\log n)$ où n est le nombre de pairs.

Dans un système P2P structuré, le déséquilibre de charge signifie une distribution inadéquate de charge sur les pairs. Cela réduit substantiellement la performance du système qui est souvent comprise comme la rapidité de la réponse aux requêtes et la qualité de service du stockage. L'équilibrage de charge est un problème souvent abordé dans la recherche sur les systèmes P2P structurés. Il implique habituellement des solutions de redistribution des charges entre les pairs pour atteindre un état d'équilibre.

Nous considérons deux aspects de charge dans les systèmes P2P structurés : la charge de stockage et la charge de gestion d'index. Dans un système P2P servant le stockage, des pairs contribuent leurs ressources (l'espace pour le stockage des objets, la bande passante pour le trafic du réseau) au fonctionnement du système. Le stockage des objets est réparti d'une certaine façon sur les pairs disponibles. Chaque pair s'occupe d'un ensemble d'objets. *La charge de stockage* sur un pair représente l'utilisation de ses ressources pour cette tâche. D'autre part, le routage d'un message dans un système P2P requiert que le message traverse plusieurs pairs entre le pair d'origine et le pair de destination. Le routage consomme donc de la bande passante des pairs le long du chemin. Nous utilisons l'appellation *la charge de gestion d'index* pour dénoter la consommation de la bande passante servant le routage sur chaque pair.

L'équilibrage de charge de stockage et de charge de gestion d'index dépend des facteurs

suivants :

- la distribution des clés et la distribution des objets sur les pairs,
- la popularité des objets,
- la taille des objets,
- la capacité des pairs à effectuer le stockage et le routage,
- le dynamisme des pairs et des objets (et leur popularité).

Plusieur mécanismes d'équilibrage de charge dans les systèmes P2P structurés ont été proposés. Nous les groupons ici selon les approches principales suivantes :

1. *Équilibrage basé sur l'égalisation de la responsabilité des clés.* Si on suppose que plus un pair s'occupe des clés, plus grande est la charge qu'il subit, cette approche vise à égaliser la taille des ensembles de clés gérés par les pairs. Des exemples de cette approche sont les méthodes de Bienkowski et al. [BKadH05], de Manku [Man04] et des systèmes P2P effectuant la partition d'un graphe de De Bruijn [NW03a, LKRG03, WZLL04]. Ces méthodes proposent différents algorithmes d'arrivée et de départ des pairs qui divisent et fusionnent des ensembles de clés pour les égaliser.

   Cette approche est simple. Cependant, l'égalisation de la responsabilité des clés ne garantit pas l'équilibre de charge dans la plupart des cas pratiques car la distribution de charge est encore affectée par la distribution des objets sur l'espace de clés, la popularité des objets, la taille des objets et les capacités des pairs.

2. *Équilibrage basé sur le transfert dynamique de la responsabilité de clés.* En appliquant cette approche, le système P2P permet aux pairs d'échanger dynamiquement la responsabilité de clés pour redistribuer la charge. La méthode Virtual Servers [RLS+03] décompose chaque pair en plusieurs serveurs virtuels qui fonctionnent comme des pairs individuels dans un système P2P normal. L'échange de responsabilité de clés est fait en réarrangeant des serveurs virtuels parmi des pairs physiques. La méthode de Karger et Ruhl [KR04a, KR04b] définit un protocole d'ajustement dynamique de la position des

pairs dans l'anneau des clés pour diminuer la différence de charge entre les pairs. Un autre exemple est P-Grid [ACMD+03, ADH03], un système P2P structuré qui forme un arbre virtuel dont le chemin de la racine à chaque feuille indique l'ensemble des clés gérées par un pair (la représentation binaire du chemin est un préfixe de clé). L'équilibrage de charge se fait en changeant dynamiquement la forme de l'arbre virtuel.

Cette approche tient compte de l'inégalité dans la distribution de charge et/ou la distribution de capacité des pairs. Cependant, si on l'applique dans l'équilibrage de charge de stockage (comme des méthodes mentionnées), on ne peut pas équilibrer simultanément la charge de gestion d'index. La raison est qu'elle associe la charge de stockage aux clés.

3. *Équilibrage basé sur le chevauchement de la responsabilité des objets.* Cette approche équilibre la charge au niveau des objets. Elle permet à chaque objet de choisir son emplacement entre différents pairs afin d'équilibrer la charge. Quelques méthodes de ce type utilisent des pointeurs de redirection pour faciliter l'accès aux objets. Deux exemples de cette approche sont l'application du paradigme *power of two choices* [BCM03] et l'équilibrage de charge dans PAST [RD01b]. Le premier emploie un ensemble de fonctions de hachage pour déterminer l'ensemble des pairs pouvant stocker un objet. Le deuxième utilise l'ensemble *leaf* de chaque pair (défini au niveau P2P Pastry [RD01a]) pour rendre plus flexible le stockage des objets. Le choix de l'hébergeur d'un objet permet à ces systèmes de distribuer charge entre les pairs.

   Les méthodes comme l'application du paradigme *power of two choices* et PAST restreignent le stockage d'un objet parmi un certain ensemble de pairs. L'application de cette restriction peut engendrer un coût considérable de maintenance dans un système dynamique.

   Le substrat de routage P2P Tapestry [ZHS+04] sépare le stockage d'un objet de sa racine sans restriction. Cette propriété permet à OceanStore [KBC+00], une application

de stockage basée sur Tapestry, d'arranger des répliques d'un objet pour ajuster le service des requêtes d'accès. Cependant, comme les autres méthodes de cette approche mentionnées, la conception de Tapestry ne supporte pas la séparation entre les clés et l'identificateur (id) des pairs. Cela empêche l'intégration d'un équilibrage de charge de gestion d'index efficace qui se base sur le transfert de responsabilité de clés.

4. *Équilibrage basé sur la restructuration des liens entre les pairs.* La méthode Expressways [ZSZ02] construit un système P2P qui gère l'espace de clés (étant un espace cartésien) sous forme d'une hiérarchie virtuelle de zones. Cette hiérarchie accélère le routage en utilisant des liens entre des zones de différents niveaux. Ceci permet au système d'équilibrer la charge de gestion d'index en restructurant dynamiquement la hiérarchie de zones. La restructuration se fait en redéfinissant les liens entre les pairs.

   Cette méthode exige un grand nombre de liens (c.-à-d., un grand degré des pairs). Ceci entraîne un grand coût de maintenance du système. De plus, l'équilibrage de charge dans la méthode Expressways ne réagit qu'après la collection des informations de charge et de capacité du système entier. Ceci peut causer des coûts élevés en temps et propagation de données.

Les méthodes ci-dessus ont le problème suivant en commun : elles ne considèrent qu'un aspect de charge, soit la charge de stockage, soit la charge de gestion d'index, mais pas les deux à la fois.

1.2 Objectif de la recherche

Les systèmes P2P structurés présument deux propriétés indispensables : dynamisme et grande échelle. Dans de tels systèmes, l'équilibrage de charge est nécessaire pour garantir une bonne qualité de service. À notre connaissance, aucun travail existant ne considère

l'équilibrage simultané de charge de stockage et de charge de gestion d'index. Dans le cadre du présent travail, nous cherchons une solution à ce problème. Nous visons à proposer un système P2P structuré qui :

- permet simultanément l'équilibrage de charge de stockage et l'équilibrage de charge de gestion d'index,
- tient compte des facteurs affectant l'équilibre des charges (sect. 1.1),
- minimise le coût de maintenance en conséquence du dynamisme.

1.3 Méthode proposée

Le système, nommé BALLS (*Balanced Load Supported P2P Structure*), s'inspire des graphes de De Bruijn. En raison de ses propriétés intéressantes (degré constant des nœuds et diamètre logarithmique par rapport au nombre de nœuds), l'emploi du graphe de De Bruijn apparaît fréquemment dans la littérature sur les réseaux d'interconnexion et de communication (p.ex., Bermond et al. [BE96, BDE97], Chung et al. [CCRS87], et auf der Heide et Vöcking [adHV99]). Une description des graphes de De Bruijn se trouve à la section 2.3.3.

Notre analyse sur les systèmes P2P structurés (chap. 2) montre que les applications des graphes sont les meilleurs en considérant le degré des pairs (constant) et le coût de routage (logarithmique). Les systèmes se basant sur d'autres approches (tableau de routage à plusieurs niveaux, espace de clés multidimensionnel) ne possèdent pas ces propriétés simultanément (tab. 2.I). L'équilibrage de charge de gestion d'index est un élément de notre objectif. La construction du système en partitionnant un graphe nous offre une approche de solution à ce problème. Elle permet aux pairs de réorganiser l'ensemble des nœuds du graphe (étant des clés du système) et donc de redistribuer le trafic de routage. Un graphe de De Bruijn binaire a un bas degré (sect. 2.3.3). Intuitivement, cela devrait réduire le degré des systèmes P2P construits sur ce graphe. Le petit degré résultant de l'utilisation des graphes de De Bruijn

simplifiera la maintenance du système et donc l'arrangement des clés entre les pairs dans l'équilibrage de charge de gestion d'index.

BALLS partitionne un graphe de De Bruijn binaire parmi les pairs du réseau P2P. L'espace de clés est identique à l'espace des nœuds $[0, 2^m - 1]$ du graphe. Ainsi, nous utilisons souvent le terme clé au lieu de nœud de De Bruijn. L'espace de clés est circulaire. Il forme un anneau de clés dans lequel la clé $2^m - 1$ précède la clé 0. Chaque pair s'occupe d'un seul intervalle dans l'espace de clés. Deux pairs se connectent si leurs intervalles de clés sont connectés par au moins un arc dans le graphe de De Bruijn ou sont adjacents dans l'espace circulaire de clés. Dans BALLS, le routage suit des chemins de routage appropriés du graphe de De Bruijn, qui sont toujours efficaces.

Loguinov et al. [LKRG03], et Naor et Weider [NW03a] utilisent une structure P2P similaire à la nôtre. Cependant, ils présument l'homogénéité de la capacité des pairs et une distribution égale de la charge sur les clés. Nous, au contraire, voulons résoudre le problème d'équilibrage de charge en tenant compte de tous les facteurs d'impact mentionnés à la section 1.1. Ceci exige des mécanismes d'équilibrage plus compliqués.

Les pannes sont critiques dans les systèmes P2P. On verra dans les sections 5.4 et 7.2.3 la possibilité d'implanter un mécanisme de réplication sur BALLS afin d'augmenter la disponibilité des objets en prévenant l'impact des pannes. Cependant, une étude plus profonde des mécanismes de tolérance aux pannes est hors de la portée de cette thèse, notre travail se concentrant sur l'équilibrage intégré de charges.

1.3.1 Équilibrage de charge de gestion d'index

Le calcul de la charge de gestion d'index tient compte des facteurs suivants : distributions des clés et des objets sur les pairs, popularité des objets et dynamisme. La charge sur un pair est le taux d'utilisation de la bande passante (dans le temps) pour router les messages

traversant le pair. Chaque pair a une capacité propre consacrée au routage. Nous définissons la surcharge de gestion d'index comme la différence entre la charge et la capacité (quand la charge dépasse la capacité). Le but de l'équilibrage est de minimiser la surcharge totale de tout le système.

Pour équilibrer la charge de gestion d'index, BALLS sépare les clés gérées par un pair de l'adresse du pair. Ceci permet aux pairs d'échanger de la responsabilité de clés sans affecter leur adresse. Parce que le routage du système se base sur le routage dans le graphe de De Bruijn, le transfert d'un intervalle de clés d'un pair à un autre pair entraîne le transfert d'une quantité de charge de gestion d'index.

L'équilibrage de charge de gestion d'index se fait de façon dynamique et décentralisée. Quand une surcharge existe sur un pair, il ajuste la responsabilité de clés avec un pair adjacent (dans l'anneau de clés) afin de minimiser leur surcharge combinée. La minimisation de la surcharge combinée des pairs contribue à la minimisation de la surcharge globale.

1.3.2 Équilibrage de charge de stockage

Pour équilibrer la charge de stockage concuremment avec l'équilibrage de charge de gestion d'index, il faut que la distribution des objets sur les pairs soit indépendante de la distribution des clés. BALLS sépare complètement le stockage des objets de leur clé. L'emplacement d'un objet n'est pas limité au pair responsable de sa clé (ce pair est appelé la racine de la clé et de l'objet). L'objet peut migrer sur n'importe quel pair ayant suffisamment d'espace. Si un objet réside sur un pair autre que sa racine, la racine maintient un pointeur de redirection (aussi appelé pointeur de stockage) pour garder l'accès à l'objet.

La charge de stockage sur un pair est la somme des tailles de tous les objets stockés par le pair. Chaque pair contribue au stockage global une quantité d'espace que nous appelons la capacité de stockage. La charge de stockage ne doit pas dépasser cette limite. L'équilibrage

de charge de stockage tient aussi compte de la capacité de bande passante consacrée par les pairs pour l'accès et la migration des objets afin de garantir le bon fonctionnement. Nous définissons une autre limite de la charge de stockage, soit la capacité de stockage désirée. La surcharge de stockage dénote la partie de la charge de stockage qui dépasse la capacité désirée. Le but de l'équilibrage de charge de stockage est de minimiser la surcharge de stockage totale du système.

Comme l'équilibrage de charge de gestion d'index, l'équilibrage de charge de stockage fonctionne de façon dynamique et décentralisée. Il se base sur l'échange d'objets entre pairs afin de minimiser la surcharge combinée des pairs. L'échange doit également garantir que la charge des pairs ne dépasse pas leur capacité de stockage. Les minimisations cumulatives de surcharge combinée mènent à la minimisation de la surcharge globale.

1.4 Évaluation

Afin de vérifier la conformité de la méthode théorique à l'objectif de la recherche, une grande partie de notre travail est réservée à l'évaluation.

1.4.1 Critère d'évaluation

Le résultat théorique de la recherche consiste en une structure P2P (BALLS) et deux mécanismes d'équilibrage de charge (de stockage et de gestion d'index) intégrés. En se basant sur l'objectif du travail (sect. 1.2), nous évaluons le résultat théorique selon les critères suivants :

- *Efficacité de la topologie de BALLS* : Le système P2P doit avoir un coût bas de maintenance qui comprend le coût d'arrivée et le coût de départ des pairs. Il est souvent dérivé du degré des pairs. En appliquant le graphe de De Bruijn à la structure P2P, on

s'attend à un degré constant des pairs. D'ailleurs, une vérification du coût de routage est aussi nécessaire. Dans ce cas, on attend une borne logarithmique pour le nombre de sauts du routage.

- *Efficacité et convergence de l'équilibrage de charge de stockage* : L'effet de l'équilibrage se manifeste par la réduction de la surcharge totale. Le mécanisme d'équilibrage de charge de stockage doit prendre effet dans un large intervalle de dynamisme de pairs et un large intervalle de ratio d'utilisation (c.-à-d., la fraction de la charge totale sur la capacité totale). L'équilibrage est convergent s'il prend effet sans jamais augmenter la surcharge totale. L'évaluation doit confirmer la convergence de l'équilibrage à différents ratios d'utilisation (incluant 100%, plus hauts et plus bas que 100%).
- *Efficacité et convergence de l'équilibrage de charge de gestion d'index* : L'évaluation doit confirmer l'effet et la convergence de l'équilibrage de charge de gestion d'index. L'effet et la convergence de cet aspect d'équilibrage ont la même signification que pour l'équilibrage de charge de stockage. Il faut aussi prendre en compte les facteurs qui affectent l'efficacité de l'équilibrage : le ratio d'utilisation, le dynamisme des pairs et le dynamisme de la popularité des clés.
- *Fonctionnement simultané de l'équilibrage de charge de stockage et l'équilibrage de charge de gestion d'index* : Le support de l'équilibrage intégré des charges demande le fonctionnement simultané et effectif des mécanismes proposés. Ce critère exige que l'activation de l'équilibrage d'un aspect de charge ne réduise pas l'efficacité de l'équilibrage de l'autre aspect, et vice-versa.

1.4.2 Outils d'évaluation

L'évaluation se base sur des simulations. Nous avons dévelopé un simulateur [LBK06b] pour l'expérimentation. Le simulateur supporte la construction et l'opération du système.

Pour fin d'évaluation, il permet d'observer différentes métriques en cours d'opération dont les principales sont le degré des pairs, le coût d'arrivée des pairs, le coût de départ des pairs, le coût de routage, le ratio de surcharge de stockage et le ratio de surcharge de gestion d'index.

Le simulateur permet de configurer l'environnement de simulation afin d'explorer et de comparer la fonctionnalité du modèle dans différentes conditions. Les paramètres principaux sont la taille initiale du système, le niveau de dynamisme des pairs, la distribution des capacités des pairs (uniforme ou Zipf), la distribution de la taille des objets (uniforme ou Log-normal), la distribution de la popularité des clés (uniforme ou Zipf), le niveau de dynamisme de la popularité des clés et différents scénarios d'activation/désactivation des mécanismes d'équilibrage.

Nous avons aussi implémenté des simulations simplifiées permettant la comparaison avec d'autres modèles (incluant l'application des serveurs virtuels, PAST et Expressways). Ces simulations servent à valider les avantages de BALLS comparativement à ces systèmes. Ils ne supportent que l'observation des métriques nécessaires pour la validation.

1.5 Organisation de la thèse

La présente thèse suit une organisation de thèse par articles. Après l'introduction, nous recensons la littérature connexe dans les deux prochains chapitres. Le chapitre 2 présente les systèmes P2P. Le chapitre 3 décrit les approches d'équilibrage existantes dans les systèmes P2P structurés.

Le chapitre 4 présente la méthodologie utilisée dans cette recherche. Les trois chapitres suivants contiennent les articles ([LBK06c, LBK06b, LBK06a]) qui décrivent les résultats de nos travaux. La discussion générale du chapitre 8 termine la thèse.

Chapitre 2

Systèmes pair-à-pair

Les systèmes P2P sont construits de différentes manières. Ce chapitre recense la diversité de leur topologie et leur performance.

2.1 Des systèmes client/serveur aux systèmes P2P

Le paradigme client/serveur domine actuellement le calcul distribué. Il désigne un ou un groupe de processus comme serveurs qui fournissent des services (de calcul, stockage et/ou trafic) à un grand nombre de clients (processus). Des exemples de tels systèmes sont les services Web, les systèmes de gestion de base de données (Oracle, Microsoft SQL Server), les systèmes d'exploitation de réseau (Novel NetWare, Linux). Le plus grand inconvénient du paradigme client/serveur est la concentration de charge sur les serveurs. Les serveurs exigent des plate-formes de haute performance pour assurer la qualité du service requise.

Le paradigme P2P vise à résoudre cet inconvénient en répartissant la charge du système sur les différents processus. Pour ce faire, les processus, étant les pairs, peuvent demander et fournir des services à la fois. Un système P2P agrège donc les ressources de tous les

participants pour constituer la puissance globale de calcul.

Il existe une couche intermédiaire entre les paradigmes client/serveur et P2P que nous appelons semi-P2P. Un tel système répartit les services sur les participants (aspect P2P) mais gère le répertoire des services de façon centralisée (aspect client/serveur). Un exemple de système semi-P2P est Napster [nap06, Shi01], un environnement de partage de fichiers musicaux. Le téléchargement d'un fichier se fait directement de l'ordinateur stockant le fichier alors que son emplacement est retrouvé sur le répertoire centralisé.

Le deuxième exemple est le projet Stardust@Home [sta06] qui agrège l'aide des volontaires pour localiser des parcelles de poussière interstellaire. L'engin Stardust a capturé des parcelles de poussière interstellaire dans son trajet spatial à l'aide d'un collecteur en aerogel. Comme les parcelles sont extrêmement petites, la localisation exige un grand travail. Dans ce projet, 1,6 million de « clips vidéo » numériques seront créés qui couvrent la surface du collecteur. En employant un « microscope virtuel », un logiciel basé sur le Web, de nombreux volontaires téléchargent des clips et y cherchent l'empreinte de parcelles. Le résultat des recherches est retourné à Stardust@Home ce qui permet à des scientifiques de concentrer la localisation sur les positions suspectées par les volontaires. Le système distribue donc la charge de travail sur les pairs (les volontaires avec leur microscope virtuel). Le collecteur contenant des parcelles a été retourné sur Terre en janvier 2006. Le projet d'analyse des vidéos vient d'être lancé.

Un projet similaire, plus ancien, est SETI@Home [set06, And01]. Ce système utilise la puissance inutilisée des ordinateurs connectés à l'Internet pour la recherche d'intelligence extra-terrestre. Les données captées par le télescope Arecibo sont divisées en petites unités, téléchargées et traitées par un logiciel fonctionnant comme un économiseur d'écran sur de nombreux ordinateurs. À la fin d'un traitement, le logiciel envoie le résultat au serveur de SETI@Home et continue avec de nouvelles données. On connaît aussi un projet de recherche

sur le cancer [udc04] qui applique une technique similaire à SETI@Home. Ce projet analyse l'interaction des structures moléculaires avec des protéines afin de trouver des drogues pour le traitement du cancer.

Un autre exemple est le Power Server Model [Loo03] qui emploie la puissance de plusieurs ordinateurs pour exécuter des travaux intensifs. Dans ce modèle, un ordinateur coordinateur fonctionne comme un courtier. Des ordinateurs (aussi appelés serveurs) contribuent leurs ressources en s'inscrivant auprès du coordinateur. Un ordinateur (un client) ayant une grosse tâche à faire doit diviser la tâche en sous-tâches, créer des servlets Java correspondants aux sous-tâches et télécharger les servlets sur le coordinateur. Les servlets sont répartis sur des serveurs appropriés. Puis les sous-tâches et les résultats sont échangés par communication directe entre le client et les serveurs désignés.

Les systèmes semi-P2P peuvent résoudre des travaux énormes en distribuant la charge (de stockage ou calcul) à de nombreux ordinateurs. Cependant, le répertoire centralisé a encore la responsabilité de coordination du tout système. Ceci peut concentrer la charge de communication sur le coordinateur et le faire devenir un point critique de défaillance.

Dans un système P2P complètement décentralisé, il n'y a plus de répertoire central de ressources. Aucun pair n'est capable de connaître ou de contrôler tout le système. Ceci permet aux pairs d'entrer et de sortir dynamiquement. La recherche de service cependant devient plus difficile. Cela pose le problème suivant : « étant donné un objet stocké dans le système, comment localiser le pair qui en est responsable afin de trouver cet objet ? » Ce problème porte le nom de routage et localisation d'objet. Il constitue un problème critique commun des systèmes P2P [BKK+03]. En se basant sur la solution du problème de routage et localisation d'objet, on distingue deux catégories de systèmes P2P : *les systèmes P2P non-structurés* et *les systèmes P2P structurés*.

2.2 Systèmes P2P non-structurés

Un système P2P non-structuré est un système P2P qui n'impose pas de règle de connexion entre les pairs. Nous présentons ici deux exemples : Gnutella et Freenet.

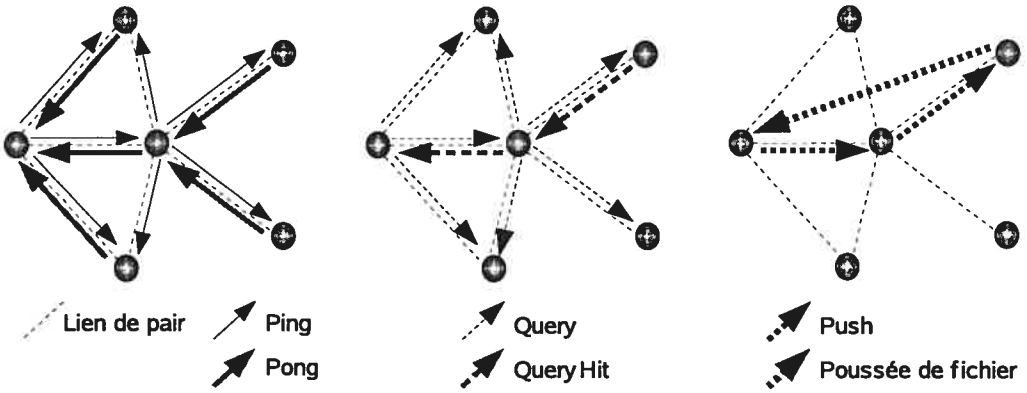

FIG. 2.1 – Routage des messages de Gnutella

Gnutella [gnu01, RFI02, VBKJ02] fournit un service de partage de fichier. Chaque pair fonctionne sur un ordinateur identifié par son adresse IP, stocke des fichiers publiés et maintient le contact avec un groupe de pairs voisins. Le protocole Gnutella définit cinq types de message : Ping, Pong, Query, QueryHit et Push dont le routage est illustré à la figure 2.1. Les messages Ping et Query servent à découvrir des pairs et à chercher des fichiers, respectivement. Gnutella transmet ces messages par une technique d'inondation. Chaque pair recevant un Ping ou Query transmet le message à tous ses voisins sauf le pair duquel vient le message. Les messages Pong, QueryHit et Push sont les réponses à Ping, Query et QueryHit, respectivement. Ils suivent le même chemin (dans le sens inverse) que celui du message auquel ils répondent. Push sert à demander au pair hébergeant un fichier de pousser le fichier au pair le demandant s'il est derrière un pare-feu qui empêche le téléchargement par le demandeur. Pour éviter une transmission infinie, chaque message est muni d'un champs TTL (*time-to-live*) qui limite le nombre de sauts.

Freenet [fre06, CSWH01, CHM+02] est un système de partage de fichier qui protège

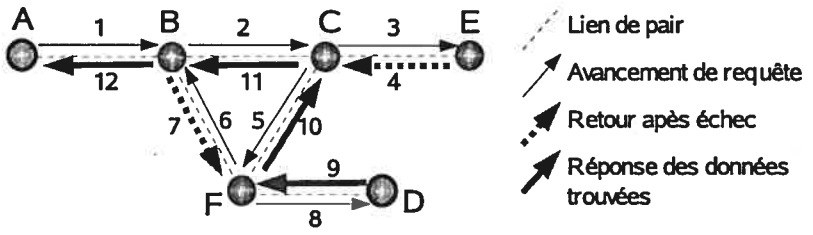

FIG. 2.2 – Chemin de requête dans Freenet (adaptée de Clarke et al. [CHM+02])

la libre publication, le téléchargement et l'anonymat des usagers. Chaque pair de Freenet maintient un tableau de routage contenant l'adresse de quelques autres pairs (voisins) ainsi que les clés des fichiers qu'il croit que ces voisins stockent.

La localisation d'un fichier applique la méthode de recherche en profondeur. La figure 2.2 illustre ce mécanisme. Un message de requête passe d'un pair à l'autre. S'il atteint le pair (D) stockant le fichier recherché, une réponse spécifiant le lieu de stockage est retournée au pair demandeur (A) (en suivant le même chemin que la requête). Sinon, le pair courant achemine le message au voisin qui contient la clé la plus proche de celle cherchée. Si le message atteint un pair déjà visité (B) ou un cul-de-sac (E) (c.-à-d., échec de recherche), il retourne au pair précédent et essaie un autre chemin. Un message a un champs TTL (*time-to-live*) qui limite la longueur du chemin de recherche.

L'insertion d'un fichier suit le même chemin que la recherche de la clé du fichier. Le message d'insertion a un champs TTL qui détermine le nombre de copies du fichier à stocker. Si un fichier avec la même clé existe déjà sur le chemin du message, l'insertion échoue. Sinon (TTL expire sans collision), le dernier pair répond « *all clear* » au pair d'origine du fichier pour accueillir le fichier au stockage des pairs sur le chemin.

Les systèmes P2P non-structurés supportent des opérations simples et efficaces d'arrivée et de départ des pairs parce que les pairs n'ont pas à maintenir une topologie. Cependant, la localisation d'objet n'est pas réalisée exhaustivement. Le champs TTL peut empêcher de trouver le fichier cherché même s'il existe. De plus, l'inondation de message dans Gnutella

cause un grand coût de trafic du réseau. Les systèmes P2P structurés résolvent ce problème.

2.3 Systèmes P2P structurés

Un système P2P structuré doit maintenir une topologie. Il définit un espace de clés et projette les objets sur cet espace. La plupart des systèmes utilisent une fonction de hachage pour effectuer cette projection et donc sont appelés tableaux de hachage distribués (*Distributed Hash Tables* ou DHT). Un système P2P structuré distribue la responsabilité des clés aux pairs. Les sous-ensembles de l'espace de clés occupés par les pairs peuvent être disjoints ou non pour des fins de fiabilité. Le système définit une structure de connexion entre les pairs en fonction des clés dont chacun s'occupe.

La localisation d'un objet se fait en calculant la clé de l'objet et routant une requête au pair voisin le rapprochant le plus de la cible. Malgré l'absence de connaissance globale, chaque pair peut acheminer correctement la requête à l'aide de la structure de connexion connue. En théorie, un système P2P structuré garantit de trouver n'importe quel objet s'il existe. Il supporte habituellement un routage très efficace dont le coût est souvent $O(\log n)$ où n est le nombre de pairs.

Cependant, l'arrivée et le départ d'un pair ne sont plus aussi simples car le système doit réorganiser les pairs pour maintenir la structure de connexion. Différentes topologies entraînent différentes efficacités de routage et de maintenance du système. Nous catégorisons les topologies des systèmes P2P structurés existants en trois groupes : *topologies basées sur un tableau de routage à plusieurs niveaux*, *topologies basées sur un espace de clés multidimensionnel* et *topologies inspirées d'un graphe*.

2.3.1 Topologies basées sur un tableau de routage à plusieurs niveaux

Le tableau de routage d'un pair contient les pointeurs vers certains pairs (voisins) ayant une relation avec le pair courant. Le principe de ce groupe de topologies est de construire le tableau de routage des pairs en utilisant plusieurs niveaux. Rappelons que chaque pair s'occupe d'un ensemble donné de clés (p.ex., les clés les plus proches numériquement de l'id du pair ou les clés partageant un préfixe identique à l'id du pair). On peut donc projeter les pairs sur l'espace de clés et définir la distance entre les pairs dans cet espace. À chacun des niveaux du tableau de routage, le pair maintient un certain nombre de pointeurs vers d'autres pairs. Les niveaux correspondent à des échelles de distances différentes. Dans quelques systèmes, l'échelle de distance croît avec la croissance du niveau. Dans les autres systèmes, elle décroît quand le niveau augmente. Peu importe ces deux cas, la relation entre les échelles est exponentielle. Ainsi, le rapport des distances des pointeurs dans deux niveaux consécutifs est approximativement b où b est souvent la base de la représentation des clés. Les différents niveaux de pointeurs permettent au routage de passer de pair à pair en précisant progressivement la position de la cible. La relation exponentielle des distances induit un coût logarithmique de routage. Des exemples de ce type de topologie sont Tapestry, Chord et Pastry.

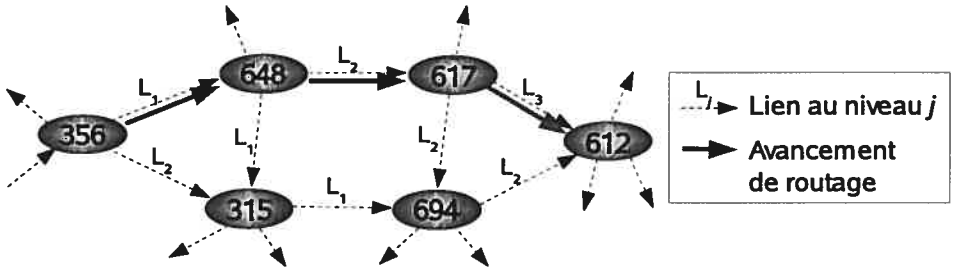

FIG. 2.3 – Exemple de routage du pair 356 au pair 612 dans Tapestry ($b = 10$, $N = 10^3$)

Tapestry [ZKJ01, ZHS+04] définit un réseau P2P dans lequel l'id d'un pair est une chaîne de chiffres extraits d'un alphabet de base b. Dénotons N comme la taille de l'espace

des ids. Le tableau de routage consiste en $\log_b N$ niveaux. Chaque niveau a b entrées. Sur un pair a (nous utilisons l'id d'un pair pour le dénoter), la $i^{\text{ième}}$ entrée du $j^{\text{ième}}$ niveau garde le lien vers le pair le plus proche (selon une mesure de localité) ayant un préfixe identique à $prefix(a, j-1) \bullet i$. Le routage pour un id k cherche simplement dans le tableau de routage le pointeur partageant le plus long préfixe commun avec k pour avancer. Ceci garantit un coût de routage $O(\log N)$. Le degré du pair est aussi $O(\log N)$. La figure 2.3 dépeint un exemple de routage du pair 356 au pair 612. Il suit les liens à différents niveaux et arrive au but après 3 sauts.

Des applications de Tapestry comprennent l'application de multicast Bayeux [ZZJ+01] et les systèmes de stockage Mnemosyne [HR02] et OceanStore [KBC+00, REG+03]. Il y a une extension de Tapestry qui supporte la localisation d'objets similaires [ZZZ+03] (Tapestry ne permet que la recherche exacte).

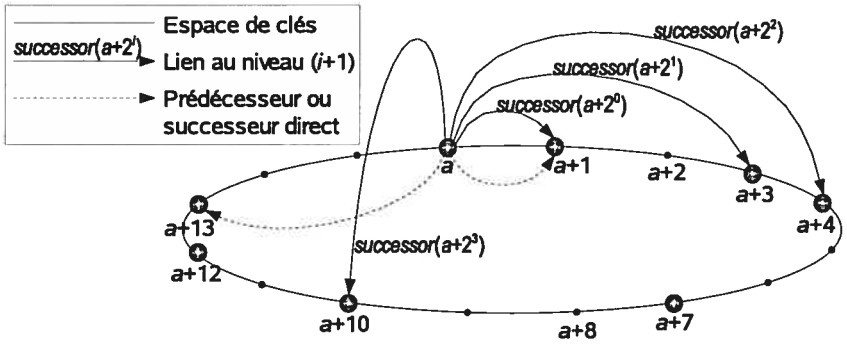

FIG. 2.4 – Exemple des liens d'un pair a dans Chord

Chord [SMK+01] définit un système P2P sur un espace circulaire de 2^m clés. Les ids des pairs sont choisis dans l'espace de clés. Un id est donc une chaîne de m bits. Les pairs immédiatement avant et après un pair a dans le cercle des clés sont appelés, respectivement, le prédécesseur et le successeur de a. Le pair a est responsable des clés qui précèdent a (incluant a), mais qui succèdent le prédécesseur de a. On écrit $a = successor(k)$ si a est responsable de la clé k. Le tableau de routage de a possède m niveaux. Chaque $i^{\text{ième}}$ niveau

ne contient qu'un lien qui pointe au pair $successor(a+2^{i-1})$. Le pair a a aussi deux liens vers son prédécesseur et son successeur, respectivement. La figure 2.4 illustre les liens d'un pair a dans un réseau Chord ayant 8 pairs et $m = 4$. Cette structure permet un routage efficace. Le routage d'une clé k à partir du pair a cherche dans le tableau de routage le $j^{\text{ième}}$ niveau qui donne $(a + 2^{j-1})$ le plus proche mais avant k dans le cercle des clés. Puis le routage suit le lien indiqué. Ce routage coûte $O(\log N)$ sauts dans un système de N pairs. Le degré des pairs est $O(\log N)$.

Il y a des exemples de l'application de Chord qui, entre autres, sont un système de gestion de fichiers P2P (*Cooperative File System* - CFS) [DKK+01], un système de gestion de certificats décentralisé (ConChord) [ACMR02] et un DNS décentralisé (DDNS) [CMM02].

Pastry [RD01a] fournit un substrat de routage et de localisation d'objet P2P. L'espace de clés et l'espace des ids des pairs de Pastry sont identiques à $[0, 2^{128} - 1]$. Le routage traite les clés et les ids comme des chaînes de chiffres d'une base $b = 2^r$ (avec un r prédéfini). Une clé est assignée au pair dont l'id est numériquement le plus proche. L'état de chaque pair a comprend trois tableaux :

- tableau de routage (R) de $\lceil \log_b N \rceil$ niveaux (N est le nombre de pairs). Chaque niveau contient $(b - 1)$ entrées. Les entrées du $j^{\text{ième}}$ niveau pointent vers des pairs dont l'id partage avec a un préfixe de j chiffres, mais le $(j + 1)^{\text{ième}}$ chiffre est différent ;
- ensemble des voisins (M) qui contient les pointeurs vers les $|M|$ pairs les plus proches de a selon une métrique de proximité ;
- ensemble *leaf* (L) qui contient les pointeurs vers les $|L|/2$ pairs numériquement les plus proches et inférieurs à a, et les pointeurs vers les $|L|/2$ pairs numériquement les plus proches et suppérieurs à a.

Le routage vers une clé k à partir d'un pair a cherche d'abord le pair responsable de k dans le tableau L pour y passer. Si la recherche n'est pas réussie, il utilise le tableau R. Supposons que k et a partagent un préfixe commun de longueur c. Le routage cherche dans

R le pointeur vers un pair ayant un préfixe commun avec k d'une longueur au moins égale à $c + 1$. Si un tel pointeur n'est pas trouvé, il choisit le pointeur vers un pair partageant un préfixe de longueur c avec k mais numériquement plus proche de k que a. Le routage passe alors au pair choisi. Pastry a un coût de routage dans $O(\log N)$. De plus, le degré des pairs est dans $O(\log N)$.

Des applications de Pastry sont un système de stockage P2P (PAST) [DR01], un mécanisme de cache décentralisée sur Web (Squirrel) [IRD02] et une infrastructure de multicast (SCRIBE) [CDKR02]. Il y a deux applications développées à partir de cette infrastructure : un système de distribution de contenu (SplitStream) [CDK+03] et une application de anycast [CDKR03].

2.3.2 Topologies basées sur un espace de clés multidimensionnel

Un système P2P de ce type définit l'espace de clés comme un espace Cartésien à plusieurs dimensions. L'espace multidimensionnel permet au routage de s'approcher de la cible (étant un point dans l'espace) en « marchant » sur ces différentes dimensions. Cela raccourcit donc la longueur de routage par rapport à une transmission naïve dans un espace unidimensionnel.

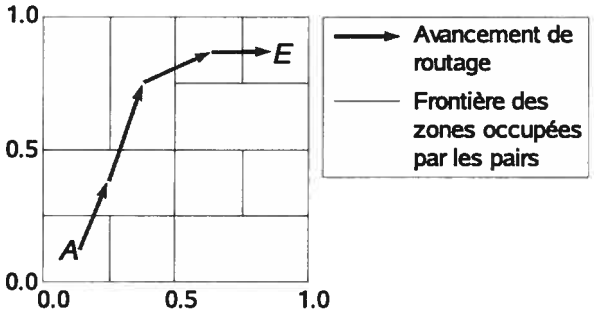

FIG. 2.5 – Exemple de routage du pair A au pair E dans CAN

Un exemple de tel système est le *Content-Addressable Network* (CAN) [RFH+01].

Ce réseau P2P partitionne un espace de clés de d dimensions sur les pairs. Deux pairs se connectent s'ils gèrent deux zones adjacentes (c.-à-d., des zones qui se chevauchent sur $(d-1)$ dimensions et se rejoignent sur 1 dimension). Le routage utilise un algorithme glouton qui traverse les frontières des zones dans les différentes dimensions pour atteindre la cible. La figure 2.5 présente un exemple de routage dans un CAN ayant 11 pairs et $d = 2$. Dans un système de n zones, le coût de routage est $O(dn^{1/d})$ et le nombre de voisins d'une zone est $O(d)$. M-CAN [Rat02] est une application de CAN qui définit un modèle de multicast au niveau de l'application.

Expressways (ou eCAN) [ZSZ02] mixe CAN et une topologie basée sur le tableau de routage à plusieurs niveaux. Le tableau à plusieurs niveaux permet à Expressways d'accélérer le routage (le coût de routage devient ainsi logarithmique). Cependant, le degré des pairs devient aussi logarithmique et n'est donc plus constant comme celui de CAN.

2.3.3 Topologies inspirées d'un graphe

Les topologies de ce groupe s'insprirent de graphes qui ont de bonnes propriétés pour le routage. Nous présentons ici quelques exemples : Viceroy, CAN D2B, Koorde et DH DHT. Tandis que Viceroy s'inspire des graphes *butterfly*, les autres appliquent les graphes de De Bruijn. Dans cette thèse, nous ne décrivons pas les graphes *butterfly*. On trouvera une description de ces graphes dans Cole et al. [CMS95]. Par contre, comme notre système utilise aussi les graphes de De Bruijn, nous présenterons brièvement leurs caractéristiques avant de décrire les systèmes CAN D2B, Koorde et DH DHT.

Viceroy

Viceroy [MNR02] est un réseau P2P construit comme un réseau *butterfly* approximatif. Chaque pair de Viceroy a un id choisi dans l'espace réel $[0, 1)$ et un numéro de niveau l (posi-

tif). Le pair choisit l aléatoirement dans l'intervalle $[1, \log n]$ où n est le nombre approximatif de pairs. Chaque pair a maintient sept liens :

- *predecessor* et *successor* qui pointent vers les pairs avant et après a dans le cercle des ids $[0, 1)$, respectivement,
- *down-right* qui pointe vers un pair au niveau $(a.l + 1)$ et à une distance approximativement $1/2^{a.l}$ de a,
- *down-left* qui pointe vers un pair au niveau $(a.l + 1)$ et proche de a,
- *up* qui pointe vers un pair au niveau $(a.l - 1)$ et proche de a,
- *next-on-level* et *prev-on-level* qui pointent vers les pairs avant et après a sur le même niveau, respectivement.

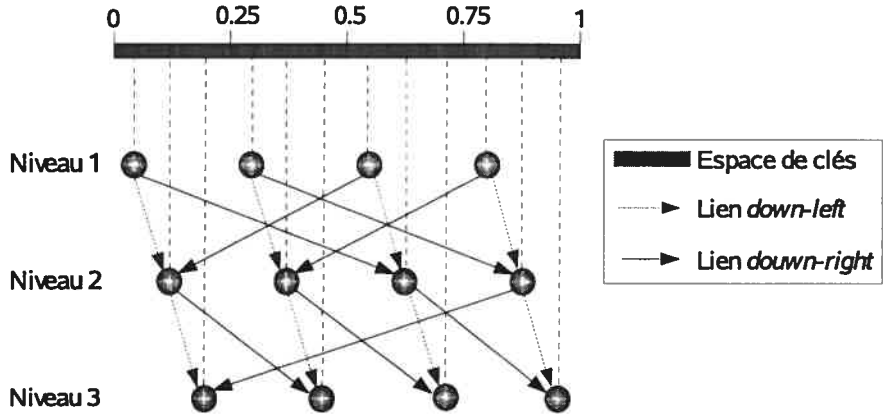

FIG. 2.6 – Exemple de liens dans Viceroy (seuls les liens *down-right* et *down-left* affichés) (adaptée de Malkhi et al. [MNR02])

La figure 2.6 présente un exemple de Viceroy avec 12 pairs. Les pairs s'organisent en 3 niveaux. Pour éviter l'enchevêtrement, elle n'illustre que les liens *down-right* et *down-left*.

Le routage vers une clé k inclut trois étapes : (1) il suit les liens *up* pour monter jusqu'au niveau 1 ; (2) si la cible n'est pas touchée, le routage descend de niveau à niveau en choisissant un lien *down-right* ou *down-left* qui s'approche le plus de k (à chaque niveau i, la distance du pair courant à k doit être au plus $1/2^{i-1}$) ; et (3) si la cible n'est pas encore touchée, le routage l'atteint via les liens *next-on-level*, *prev-on-level*, *predecessor* et *successor*.

Malgré un nombre constant (7) de liens sortant d'un pair, Viceroy fournit un routage avec le coût logarithmique. C'est l'avantage de Viceroy en comparaison avec les systèmes précédents. Une implémentation de Viceroy est décrite par Dobzinski et al. [DT03].

Graphes de De Bruijn

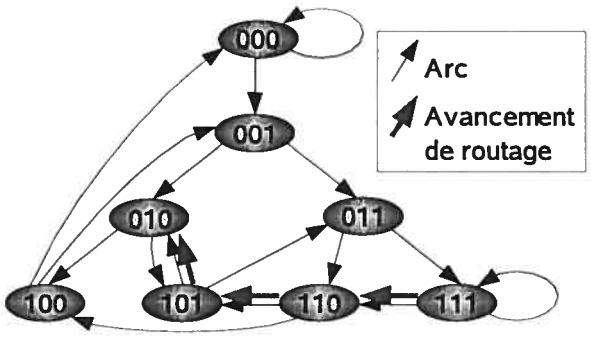

FIG. 2.7 – Graphe de De Bruijn $B(2,3)$ et le chemin de routage de 111 à 010

Un graphe de De Bruijn [dB46], dénoté $B(b,m)$, se compose de b^m nœuds dont l'id, dénoté $x_1 \cdots x_m$, appartient à l'ensemble $\{0,\cdots,b-1\}^m$. Les arcs connectent chaque nœud $x_1 \cdots x_m$ aux nœuds $x_2 \cdots x_m y$ où $y \in \{0,\cdots,b-1\}$. Cette connexion dote le graphe de De Bruijn d'un algorithme de routage très efficace. Pour router un message d'un nœud $X = x_1 \cdots x_m$ à un nœud $Y = y_1 \cdots y_m$, on détermine la chaîne s la plus longue étant un suffixe de X et un préfixe de Y (c.-à-d., $Y = s y_{|s|+1} \cdots y_m$). Puis, on progresse en passant au nœud voisin $x_2 \cdots x_m y_{|s|+1}$. L'algorithme rallonge s à chaque étape jusqu'à Y. La figure 2.7 illustre le graphe $B(2,3)$ et le routage de 111 à 010. Le chemin de routage passe par les nœuds 111, 110, 101 et 010 qui amènent l'évolution de la chaîne s : vide, 0, 01 et 010. Un graphe de De Bruijn possède certaines des propriétés d'un arbre complet : degré constant des nœuds et coût logarithmique de routage. Tandis qu'un arbre a seulement une racine (c.-à-d., le nœud au bout d'un chemin vers n'importe quel autre nœud), dans un graphe de De Bruijn, tous les nœuds sont une telle « racine ». Le graphe de De Bruijn présente des avantages dans

la construction des réseaux d'interconnexion : la longueur du chemin de routage et le degré (sortant) des nœuds sont limités par m et b, respectivement.

Systèmes P2P appliquant les graphes de De Bruijn

CAN D2B [FG03] définit un espace de clés $[0, 2^m - 1]$ (une clé est une chaîne de m bits). Chaque pair a un id (étant une chaîne de bits) de longueur l variant dans l'intervalle $[1, m]$. Un pair a est responsable de toutes les clés ayant un préfixe a. À chaque instant, un pair $x_1 \cdots x_l$ a soit un lien sortant unique vers un pair $x_2 \cdots x_g$ ($g \leq l$), soit quelques liens sortant vers des pairs de forme $x_2 \cdots x_l y_1 \cdots y_h$ ($1 \leq h \leq m - l + 1$). Par cette règle de connexion, D2B partitionne un graphe de De Bruijn binaire $B(2, m)$ de telle façon que chaque partition regroupe tous les nœuds de De Bruijn partageant un certain préfixe. Notez que D2B ne requiert pas la maintenance de tous les liens possibles entre les partitions.

Dans D2B, le routage d'un pair $x_1 \cdots x_l$ vers une clé k suit le routage dans le graphe de De Bruijn partitionné. Il calcule d'abord la plus longue chaîne s qui est un suffixe de $x_1 \cdots x_l$ et un préfixe de k. Si la longueur de s est l, le pair courant gère k. Sinon, le routage passe à un autre pair pour avancer sur le chemin de routage dans le graphe. Si $x_1 \cdots x_l$ n'a qu'un lien sortant unique, le routage prend ce lien. Sinon, il choisit un voisin $x_2 \cdots x_l y_1 \cdots y_h$ tel que $sy_1 \cdots y_h$ est un préfixe de k. Dans un réseau D2B avec n pairs, le coût de routage est $O(\log n)$. Le degré attendu des pairs est constant.

Koorde [KK03] construit une topologie inspirée de Chord et des graphes de De Bruijn. Les ids des pairs partitionnent un espace circulaire de clés. Chaque clé k est assignée au premier pair qui y succède ($successor(k)$). Un pair x maintient deux liens *predecessor* et *successor* vers les deux pairs immédiatement avant et après x, respectivement, dans le cercle des clés (comme Chord). Par ailleurs, le pair x maintient un lien d vers le premier pair précédant $2x$. Ce lien d s'inspire des arcs du graphe de De Bruijn.

Dans Koorde, le routage pour une clé k simule la marche sur le chemin vers le nœud k dans le graphe de De Bruijn correspondant. L'état de routage comprend : le pair courant x et le nœud de De Bruijn imaginaire courant i. Le routage réussit si $x = successor(i) = successor(k)$. À chaque moment, il y a deux cas : (1) si i tombe dans l'intervalle entre x et le successeur de x, i avance d'un pas vers k dans le graphe de De Bruijn et le routage passe via le lien d ; (2) sinon, le routage passe via le lien *successor* pour rejoindre i. L'efficacité de routage de Koorde est prouvée d'être $O(\log n)$ dans un système de n pairs. Chaque pair n'a que trois liens sortants.

DH DHT (*Distance Halving DHT*) [NW03a] emploie aussi le principe du graphe de De Bruijn dans sa topologie. On définit un graphe continu G_c qui généralise le graphe de De Bruijn. L'ensemble des nœuds de G_c est l'intervalle réel $[0, 1)$. Chaque nœud x a deux arcs vers $x/2$ et $(x + 1)/2$. La topologie de DH DHT est un graphe discret $G_{\vec{x}}$ qui partitionne G_c. $G_{\vec{x}}$ comprend n nœuds $\{x_1, x_2, \cdots, x_n\}$ (dénotant aussi les pairs) qui divisent $[0, 1)$ en n segments. Chaque pair x_i est responsable du segment $[x_i, x_{i+1})$. Deux pairs se connectent si leurs segments sont liés par un arc dans G_c ou s'ils sont adjacents. DH DHT définit des algorithmes de routage qui profitent aussi de l'avantage du routage des graphes de De Bruijn. Cette topologie implique un coût de routage logarithmique et un degré moyen des pairs contant.

Overlapping DHT [NW03b] est une extension de DH DHT qui permet le chevauchement de la gestion de clés entre les pairs. L'extension vise à augmenter la fiabilité du système par rapport au dynamisme. Chaque clé dans le système doit être couverte par $\Theta(\log n)$ pairs. Naturellement, le degré des pairs devient $\Theta(\log n)$.

2.4 Synthèse

Les systèmes P2P apportent une bonne solution au problème du calcul à grande échelle. Ils distribuent la charge de travail sur les participants du système afin d'assurer la performance globale vis-à-vis des tâches intensives. La première génération de systèmes P2P, les systèmes P2P non-structurés, n'impose pas de structure sur la connexion des pairs. Il en découle une construction simple et une maintenance efficace mais ne garantissant pas le déterminisme de recherche. La localisation d'un objet peut ne pas atteindre la cible, même si elle existe. La deuxième génération, les systèmes P2P structurés, résout ce problème en appliquant une structure de connexion entre les pairs. Cette structure assure le routage efficace et la localisation déterministe des objets. Cependant, la maintenance du système devient compliquée parce qu'elle doit conserver la structure imposée.

Le principe de maintenance est similaire à travers tous les systèmes P2P structurés. À l'arrivée, un nouveau pair contacte habituellement un pair connu qui fait déjà partie du système, afin de localiser ses voisins (p.ex., en routant à une clé dont il doit s'occuper). Puis le pair construit son tableau de routage et établit les connexions nécessaires. Le nouveau pair doit aussi prendre une part de responsabilité des clés et des objets de ces voisins. D'autre part, au départ d'un pair, les voisins doivent s'occuper des clés (et des objets) du pair sortant et mettre à jour leurs connexions. L'efficacité de maintenance dépend de la topologie de chaque système, dans laquelle le degré des pairs est un facteur important. Plus ce degré est élevé, plus la maintenance est complexe. Le tableau 2.I résume l'efficacité des systèmes P2P mentionnés.

Les systèmes P2P structurés supportent le routage avec une complexité logarithmique (sauf CAN qui a un coût de routage $O(dn^{1/d})$). Le degré des pairs dépend de la topologie du système. Les topologies basées sur un tableau de routage à plusieurs niveaux ont un degré logarithmique (découlant du nombre de niveaux du tableau de routage). Les topologies basées

Système		Routage et localisation déterministe des objets	Coût de routage et de localisation d'objet	Degré des pairs (sortant)
Non structuré	Gnutella	non	$O(TTL)$	
	Freenet	non	$O(TTL)$	
Structuré	Tapestry	oui	$O(\log n)$	$O(\log n)$
	Chord	oui	$O(\log n)$	$O(\log n)$
	Pastry	oui	$O(\log n)$	$O(\log n)$
	CAN	oui	$O(dn^{1/d})$	$O(d)$
	eCAN	oui	$O(\log n)$	$O(\log n)$
	Viceroy	oui	$O(\log n)$	7
	D2B	oui	$O(\log n)$	$O(1)$
	Koorde	oui	$O(\log n)$	3
	DH DHT	oui	$O(\log n)$	$O(1)$

TAB. 2.I – Résumé de l'éfficacité des systèmes P2P

sur un espace de clés multidimensionnel (CAN) ont une borne constante de degré qui est équivalente au nombre de dimensions. Les topologies inspirées des graphes présentent des degrés constants. Ils obtiennent cet avantage en profitant des bonnes propriétés des graphes utilisés.

Les systèmes P2P structurés sont efficaces pour la recherche exacte. Quelques extensions de systèmes P2P structurés (p.ex., [ZZZ+03, AS03]) vont plus loin en fournissant la recherche approximative. C'est-à-dire, la recherche des objets ayant des clés ou autres propriétés proches des valeurs données. La considération de cette recherche est hors du contexte de notre travail.

Pour revenir à notre problème principal, le prochain chapitre discutera des mécanismes d'équilibrage de charge dans les systèmes P2P structurés existants.

Chapitre 3

Équilibrage de charge dans les systèmes P2P structurés

En raison de la décentralisation, du dynamisme et de l'hétérogénéité des participants, les systèmes P2P structurés sont potentiellement vulnérables au déséquilibre de charge. On trouvera dans ce chapitre différentes approches de solution à ce problème.

3.1 Introduction

La charge représente la consommation des ressources du système pour son fonctionnement. Elle peut être l'utilisation de la bande passante du réseau, l'espace de stockage, la puissance de CPU, etc. Dans le cas d'un système P2P structuré, la charge est en fait la charge des participants ou pairs. Parmi les différents types de charge, nous considérons deux aspects : la charge de stockage et la charge de gestion d'index (sect. 1.1).

Un déséquilibre de charge dans un système P2P structuré consiste en une distribution inadéquate de la charge sur les pairs. Il y a différentes façons de voir le déséquilibre de

charge. Si on prend un point de vue simple qui ne tient pas compte de la capacité des pairs, un déséquilibre de charge survient lorsque la charge diffère d'un pair à l'autre. Si par ailleurs on considère la différence de capacité des pairs, un déséquilibre de charge survient lorsque le taux de la charge L_i sur la capacité C_i d'un pair i, dénoté L_i/C_i, est différent du taux global L/C (où $L = \sum_{\forall i} L_i$ et $C = \sum_{\forall i} C_i$). Dans le présent travail, nous considérons un déséquilibre de charge comme l'existance d'une surcharge sur les pairs. La surcharge d'un pair i est la partie de la charge L_i qui dépasse la capacité C_i. Ainsi, si $L_i \leq C_i$, la surcharge est nulle, sinon elle est égale à $L_i - C_i$. Formellement, on peut formuler ceci comme $(L_i - C_i + |L_i - C_i|)/2$. En définissant le déséquilibre de charge par la surcharge, BALLS présume que chaque pair offre une bonne performance si sa capacité couvre la charge. Le système n'a donc pas à réagir tant que la surcharge n'existe pas. Ceci permet d'éviter des réorganisations inutiles dans le système.

Un système P2P structuré, étant un environnement à grande échelle, hétérogène et dynamique, est très vulnérable au déséquilibre de charge. Nous identifions certains facteurs qui affectent l'équilibre de charge de stockage et de charge de gestion d'index :

- la distribution des clés sur les pairs,
- la distribution des objets sur les clés[1],
- la popularité des objets (et/ou des clés),
- la taille des objets,
- la capacité des pairs pour servir le stockage et le routage,
- le dynamisme des pairs (l'ajout et le retrait dans le réseau),
- le dynamisme des objets comprenant le dynamisme de l'emplacement des objets (l'insertion et la migration des objets) et le dynamisme de la popularité des objets (dans quelle mesure la popularité varie dans le temps).

[1]La combinaison de la distribution des objets sur les clés et la distribution des clés sur les pairs implique la distribution des objets sur les pairs.

L'équilibrage de charge vise à diminuer le déséquilibre existant dans le système. Les différents mécanismes d'équilibrage de charge dans les systèmes P2P structurés dépendent de la structure du système et de la définition de déséquilibre choisie. Nous catégorisons les mécanismes d'équilibrage de charge en quatre groupes : (1) *équilibrage basé sur l'égalisation de la responsabilité des clés*, (2) *équilibrage basé sur le transfert dynamique de la responsabilité de clés*, (3) *équilibrage basé sur le chevauchement de la responsabilité des objets* et (4) *équilibrage basé sur la restructuration des liens entre les pairs*.

3.2 Équilibrage basé sur l'égalisation de la responsabilité des clés

Les méthodes qui appliquent cette approche négligent la différence de capacité des pairs. Elles supposent qu'on obtient l'équilibre en égalisant la charge sur les pairs. De plus, cette approche suppose que la charge est distribuée uniformément sur l'espace de clés. Ceci implique qu'elle néglige l'hétérogénéité des objets et aussi leur dynamisme. Ainsi, plus de responsabilité de clés produit plus de charge sur un pair.

Cette approche se base sur la notation de *smoothness*, une métrique d'équilibre définie comme $\min \frac{|k_p|}{|k_q|}, \forall p, q$ où $|k_i|$ est le nombre de clés gérées par le pair i. Notez que la *smoothness* est souvent définie comme $\max \frac{|k_p|}{|k_q|}$ (c.-à-d., l'inverse de $\min \frac{|k_p|}{|k_q|}$), qui paraît moins intuitive quand on veut minimiser la « *smoothness* » plutôt que de la maximiser. Dans les systèmes qui insèrent les pairs aléatoirement, l'inverse de la *smoothness* est souvent logarithmique (par rapport au nombre de pairs). Dans ces cas, l'équilibrage de charge essaie d'augmenter la *smoothness*. Nous présentons trois exemples : la méthode de Bienkowski et al. [BKadH05], la méthode de Manku [Man04] et la méthode de DH DHT [NW03a].

La méthode de Bienkowski et al. [BKadH05] est appliquée dans des systèmes où chaque

pair gère un intervalle de clés et maintient des liens vers son prédécesseur et son successeur dans l'espace circulaire de clés $[0, 1)$. Elle catégorise les intervalles en courts, moyens et longs. La catégorisation doit assurer que le découpage d'un intervalle long ne mène jamais à un intervalle court. Les seuils permettant de déterminer la catégorie d'un intervalle se basent sur le nombre approximatif de pairs, tel que calculé par chaque pair.

L'algorithme d'équilibrage fonctionne par ronde. À chaque ronde, on extrait un certain nombre de pairs ayant un intervalle court avant de les réinsérer en découpant des intervalles longs existants. Cette méthode augmente le nombre d'intervalles moyens en éliminant des intervalles courts et longs. Elle égalise donc progressivement les intervalles.

La méthode de Manku [Man04] poursuit aussi l'égalisation des intervalles dans l'espace de clés $[0, 1)$ occupés par les pairs d'un DHT. Elle projette le système P2P sur un arbre binaire virtuel A dans lequel chaque nœud interne a exactement deux fils (gauche et droit). Les feuilles de l'arbre correspondent aux pairs. La représentation binaire du chemin de la racine à une feuille indique la partition de l'espace $[0, 1)$ attribuée au pair. Les bits 0 et 1 dénotent les moitiés gauches et droites, respectivement. Le principe de l'égalisation est de garder l'arbre A équilibré après les arrivées et départs des pairs.

On définit deux mécanismes : insertion parfaite et suppression parfaite. L'insertion parfaite d'un pair correspond au dédoublement d'une feuille dont le niveau est le plus proche de la racine de l'arbre A. Similairement, la suppression parfaite d'un pair correspond à la suppression d'une feuille. Il y a deux cas : la feuille à supprimer incombe au niveau le plus profond ou pas. Dans le premier cas, le nœud père de la feuille devient une feuille en supprimant ses deux fils. Dans le deuxième cas, la feuille à supprimer échange sa position avec une feuille au niveau le plus profond. Puis, elle continue à partir du premier cas.

Afin d'équilibrer l'arbre A et de limiter le coût résultant simultanément (le coût indique le nombre de messages nécessaires pour trouver une feuille à dédoubler ou à supprimer), les

algorithmes d'arrivée et de départ des pairs imposent une restriction dans l'application des insertions et suppressions parfaites. À l'arrivée, un pair choisit une position aléatoire dans l'espace $[0, 1)$. Supposons que la feuille i correspondant à cette position est à un niveau l. L'algorithme applique l'insertion parfaite dans le sous-arbre qui couvre la feuille i et dont la racine est à un niveau $\phi(l) \leq l$.

De même, le départ d'un pair n'applique pas la suppression parfaite dans l'arbre A globalement. Supposons que la feuille j correspondant au pair partant se trouve à un niveau l. La suppression parfaite se fait dans le sous-arbre qui couvre j et dont la racine est au niveau $\phi(l)$.

La méthode utilise $\phi(l) = \max(0, l - \lceil \log_2 l \rceil - c)$ avec une petite constante c. On prouve que cette méthode limite les feuilles dans au plus trois niveaux différents autour de $\log_2 n$ avec une haute probabilité (où n est le nombre de feuilles). Il en résulte une *smoothness* supérieure ou égale à 1/4.

On peut comprendre cette méthode d'équilibrage d'un autre point de vue. En arrivant, un nouveau pair localise le pair i gérant une position aléatoire dans $[0, 1)$. Puis il cherche dans un groupe de $(c \log n)$ pairs autour de i un pair ayant le plus large intervalle à découper. Lorsqu'il quitte le réseau, un pair j échange sa position avec un pair ayant le plus petit intervalle parmi $(c \log n)$ pairs autour de lui (le cas échéance) et fusionne son intervalle avec le plus petit intervalle adjacent.

Dans DH DHT, Naor et Wieder [NW03a] proposent l'égalisation de la gestion des clés. Leur méthode se base aussi sur les mécanismes d'insertion et de suppression de pairs. Ils décrivent un mécanisme d'insertion similaire à celui de Manku. Le nouveau pair doit estimer le nombre de pairs n, contacter les pairs gérant $(t \log n)$ points aléatoires dans l'espace $[0, 1)$ (avec une constante t) et découper l'intervalle le plus large trouvé. Dans cette méthode, les intervalles des pairs à contacter peuvent être disjoints, contrairement à ce qui est fait dans la

méthode de Manku.

Pour gérer la suppression de pairs, Naor et Wieder introduisent un mécanisme plus compliqué qui exige une structure de données additionnelle. Ce mécanisme distingue les intervalles entre courts et longs. Les intervalles longs doivent être placés consécutivement dans le cercle des clés. La chaîne des intervalles longs est marquée par deux points a (au début) et b (à la fin). Quand un pair arrive, il découpe l'intervalle long adjacent à b. D'autre part, au départ, un pair échange sa position avec le pair occupant le premier intervalle court avant de partir pour conserver la séquence des intervalles longs. Ce mécanisme garantit une *smoothness* de 1/2.

3.3 Équilibrage basé sur le transfert dynamique de la responsabilité de clés

Les méthodes appliquant cette approche tiennent compte de l'hétérogénéité des pairs et/ou de l'inégalité de la distribution de charge sur les clés. Elles associent un certain niveau de charge à la responsabilité de chaque clé. Ainsi, le transfert d'un ensemble de clés d'un pair à l'autre entraîne le transfert de charge. L'équilibre de charge est atteint en effectuant ce genre d'échanges. Nous citons ici trois exemples de cette approche : la méthode de Karger et Ruhl [KR04b], la méthode Virtual Servers [RLS+03] et l'équilibrage de charge dans P-Grid [ACMD+03]

Karger et Ruhl [KR04b] proposent une méthode simple d'équilibrage de charge qui supporte l'inégalité de distribution de charge sur l'espace de clés. Elle s'applique à un système P2P où chaque pair gère un intervalle de clés et maintient les liens à son prédécesseur et son successeur dans l'espace. Le but de l'équilibrage est d'égaliser la charge sur les pairs.

Cette méthode permet à chaque pair de contacter occasionnellement un autre pair. Dé-

notons deux pairs qui se rencontrent comme i et j, et leur charge comme l_i et l_j. Si $l_j \leq \varepsilon l_i$ (avec $0 < \varepsilon < 1/4$), les pairs effectuent un transfert de charge. On distingue deux cas. Cas 1 : si i et j ont adjacents, ils ajustent la frontière de leur intervalle pour que chacun prenne une charge $(l_i + l_j)/2$. Cas 2 : si i et j sont non-adjacents, on vérifie la charge du successeur de j (dénoté $(j+1)$). Si $l_{j+1} > l_i$, le transfert est opéré entre j et $(j+1)$ comme dans le cas 1. Sinon, j donne la responsabilité de ses clés à son successeur $(j+1)$ et prend la moitié de la charge de i en s'occupant de la part correspondante de l'intervalle de clés de i.

De tels transferts de clés égalisent la charge sur les pairs au fur et à mesure. Cette méthode ne tient pas compte de la capacité hétérogène des pairs.

La méthode Virtual Servers [RLS+03] prend en compte la capacité des pairs et l'inégalité de distribution de charge dans l'équilibrage. Elle définit deux états de pair : lourd et léger. Chaque pair i a une charge L_i et une charge désirée (*target load*) T_i. Le pair i est dans l'état lourd si $L_i > T_i$, sinon dans l'état léger. Rao et al. [RLS+03] proposent la façon suivante pour déterminer T_i. Étant donnée C_i la capacité du pair i, la charge désirée est $T_i = (\overline{L}/\overline{C} + \delta)C_i$ où $\overline{L} = (\sum_{i=1}^{n} L_i)/n$, $\overline{C} = (\sum_{i=1}^{n} C_i)/n$, n est le nombre de pairs et δ est un paramètre définissant la compensation entre la qualité et le coût de l'équilibrage. Le but de cette méthode est de diminuer le nombre de pairs dans l'état lourd.

La méthode Virtual Servers décompose un pair physique en plusieurs composants appelés serveurs virtuels. Les serveurs virtuels opèrent comme des pairs individuels dans un système P2P normal. Selon cette méthode, chaque serveur virtuel supporte une certaine charge. Le transfert de serveurs virtuels entre des pairs physiques affecte la distribution de charge et peut donc être utilisé comme mécanisme d'équilibrage.

Le transfert d'un serveur virtuel est l'opération élémentaire d'équilibrage de charge. Il consiste à sélectionner un serveur virtuel v à transférer d'un pair lourd h à un pair léger l. La sélection doit satisfaire deux conditions : (1) v ne rend pas l lourd ; et (2) v est le plus petit

serveur virtuel qui peut rendre h léger ou est le plus grand si aucun serveur virtuel ne peut rendre h léger.

Trois modèles de transfert des serveurs virtuels sont proposés : (1) *un-à-un* qui transfère des serveurs d'un pair lourd à un pair léger ; (2) *un-à-plusieurs* qui transfère des serveurs d'un pair lourd à plusieurs pairs légers ; et (3) *plusieurs-à-plusieurs* qui fait le transfert de plusieurs pairs lourds à plusieurs pairs légers. Dans les modèles (2) et (3), la sélection des destinations (pairs légers) est supportée par une structure de données additionnelle disposée au-dessus du système P2P. Cette structure maintient les informations sur la charge annoncées par les pairs.

P-Grid [Abe01, ACMD+03] est un système P2P basé sur un arbre binaire de recherche virtuel (P). Les pairs du système correspondent aux feuilles de l'arbre. Le chemin d'accès à un pair est la représentation binaire du chemin de la racine de P à la feuille correspondante. Chaque pair gère toutes les clés qui partagent un préfixe identique à ce chemin.

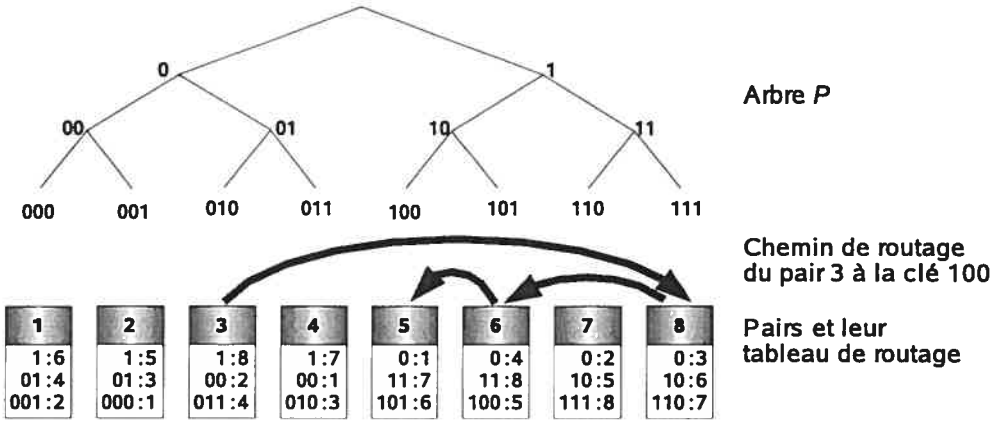

FIG. 3.1 – Exemple de P-Grid et routage du pair 3 à la clé 100

Pour garantir le routage déterministe et efficace, un pair i avec un chemin de longueur l maintient des pointeurs vers l pairs pour chacun des préfixes de longueur de 1 à l, tel que ce préfixe n'est différent de celui du pair pointé que par le dernier bit. Cela permet au routage de toujours trouver un pair plus proche de la cible. La figure 3.1 illustre un exemple de la

connexion d'un P-Grid de 8 pairs. Dans ce système, le routage pour la clé 100 à partir du pair 3 réussit après trois sauts en utilisant les pointeurs (1 : 8), (10 : 6) et (100 : 5) sur les pairs 3, 8 et 6, respectivement.

L'équilibrage de charge dans P-Grid accepte l'inégalité de distribution de charge sur les clés. Il consiste à redistribuer la responsabilité des clés sur les pairs. P-Grid sépare le chemin des pairs de leur id. Cette propriété permet aux pairs de changer leurs chemin. Les pairs reforment donc l'arbre P pour s'adapter à la distribution de charge.

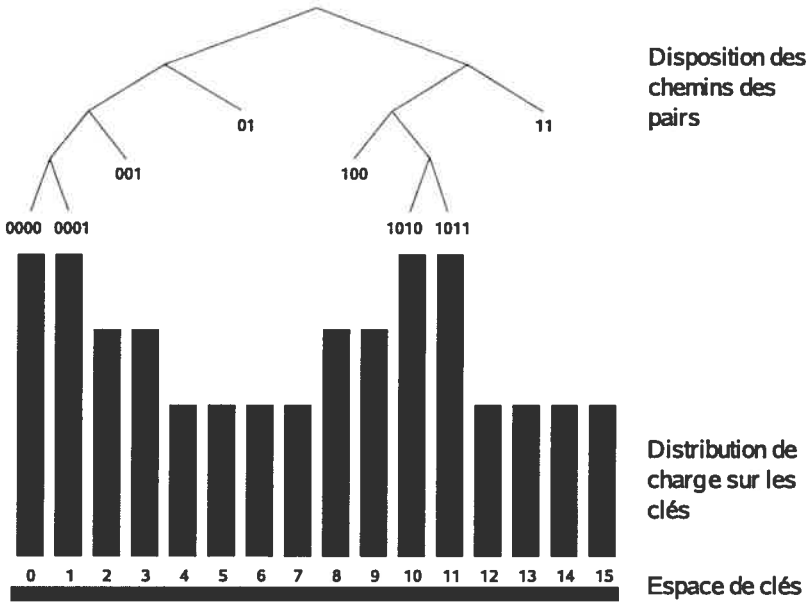

FIG. 3.2 – Exemple de distribution des clés sur les pairs dans P-Grid selon la distribution inégale de charge sur les clés (adaptée de Datta et al. [DGA04])

La figure 3.2 présente une forme de l'arbre P tenant compte de la distribution inégale de charge sur l'espace de clés de 4 bits (comme montrée par l'histogramme). Dans cette disposition, les clés les plus chargées appartiennent aux pairs les plus profonds.

3.4 Équilibrage basé sur le chevauchement de la responsabilité des objets

Cette approche est utilisée pour effectuer l'équilibrage de charge de stockage. Elle commence à redistribuer la charge au niveau des objets (mais pas au niveau des clés comme les approches précédentes). Pour ce faire, le système P2P doit permettre le chevauchement de responsabilité des objets à travers les pairs ; c'est-à-dire, la résidence d'un objet s'ouvre à plusieurs pairs et n'est pas limitée à un seul pair. Par conséquence, les objets peuvent migrer pour choisir une bonne résidence. Le chevauchement profite souvent des propriétés particulières du substrat P2P, par exemple, fonctions de hachage multiples [BCM03] ou ensemble *leaf* de Pastry [RD01b]. Dans ce cas, la relaxation des contraintes sur le lieu de stockage des objets est restreinte. Pour rendre la localisation d'objet efficace, les méthodes de cette approche disposent de pointeurs de redirection. Ces pointeurs établissent une référence du pair responsable d'un objet vers le pair qui stocke effectivement l'objet.

Un exemple de cette approche est présenté par Byers et al. [BCM03]. Cette méthode équilibre la charge dans un DHT en employant le paradigme *power of two choices* [MRS01]. Le but de l'équilibrage est de diminuer la charge maximale de chaque pair avec une haute probabilité. Pour supporter l'équilibrage de charge, le DHT doit disposer de d fonctions de hachage $h_1, \cdots, h_d$ ($d \geq 2$) qui projettent un objet o sur différentes valeurs $h_1(o), \cdots, h_d(o)$ dans l'espace de clés. Chaque objet peut donc se trouver sous la responsabilité de plusieurs pairs. Nous disons que chaque objet a plusieurs racines. L'insertion d'un objet o consiste à localiser les racines de o, à choisir la racine s contenant le plus petit nombre d'objets et à stocker o dans s. La figure 3.3 illustre cette insertion avec $d = 4$.

L'utilisation de d fonctions de hachage multiplie par d le coût de trafic pour une recherche, car elle doit lancer d localisations de clés en parallèle. Pour éviter cet inconvénient,

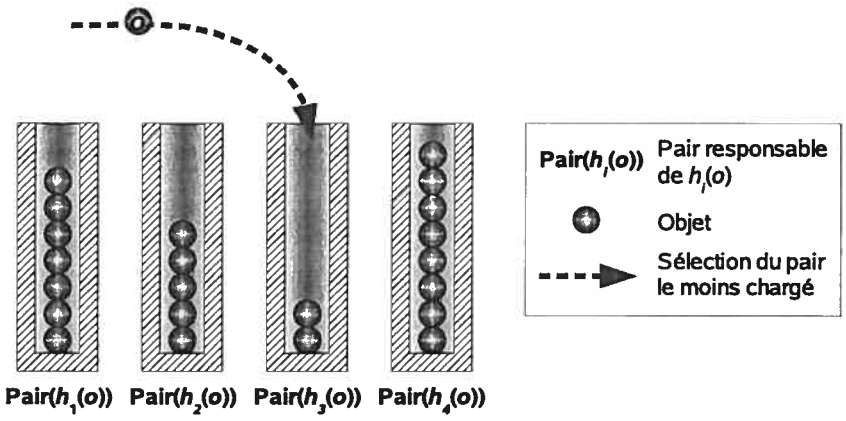

FIG. 3.3 – Exemple d'insertion d'un objet o dans le DHT appliquant le paradigme *power of two choices* ($d = 4$)

Byers et al. [BCM03] ajoutent des pointeurs de redirection dans le système. Si un objet o est stocké par un pair s correspondant à une de ses d racines, les $(d-1)$ racines autres que s maintiennent un pointeur de redirection vers s pour garder la référence à o. Ceci prolonge le chemin de recherche de o d'au plus 1 saut.

Byers et al. prouvent que dans un système de n pairs contenant n objets, la charge maximale d'un pair produite par cette méthode est[2] $\log\log n/\log d + c$ objets (où c est une constante) avec une haute probabilité. Cette charge est réduite par rapport à celle d'un DHT sans équilibrage : $\log n/\log\log n$ [MRS01].

Un autre exemple se trouve dans PAST [RD01b], un service de gestion de stockage P2P basé sur le substrat de routage Pastry [RD01a]. L'équilibrage de charge dans PAST considère des fichiers de tailles différentes (les auteurs utilisent le terme *fichier* au lieu d'*objet*) et des pairs avec des capacités de stockage différentes. Son but est d'équilibrer l'espace disponible des pairs, surtout quand la capacité totale du système est presque atteinte. Il gère donc la charge et la capacité de stockage des pairs.

Dans PAST, l'id d'un pair comprend 128 bits et celui d'un fichier comprend 160 bits.

[2]Dans les mots des auteurs : $\log\log n/\log d + O(1)$.

Un fichier est stocké en κ copies sur κ pairs ayant les ids numériquement les plus proches du préfixe de 128 bits de l'id du fichier. Nous disons que chacun de ces κ pairs est la racine d'une copie. Dans le cas de déséquilibre de charge, l'insertion d'une copie peut être refusée par la racine à cause du manque d'espace sur la racine tandis que de l'espace suffisant existe sur des pairs dans son ensemble *leaf* (voir la définition de l'ensemble *leaf* dans la description de Pastry, sect. 2.3.1).

L'équilibrage de charge de PAST utilise une technique appelée *déviation de réplique* (*replica diversion*). Elle permet à une copie de fichier de demeurer sur un pair B dans l'ensemble *leaf* de sa racine A à condition que B ne participe pas à l'ensemble des $(\kappa - 1)$ autres racines et que B ne contienne pas d'autre copie du même fichier. Après avoir dévié la copie à B, A crée un pointeur de redirection pour garder une référence à la copie.

En utilisant la technique de déviation de réplique, une racine profite de l'espace disponible des pairs autour d'elle pour accepter de nouveaux fichiers si son stockage local n'en est pas capable. Cependant, une copie ne peut pas demeurer à l'extérieur de l'ensemble *leaf* de sa racine. La maintenance de cet invariant entraîne un coût de migration de fichier, car l'évolution dynamique du système doit changer les frontières des ensembles *leaf* des pairs et potentiellement met des fichiers hors de leur région de résidence permise.

3.5 Équilibrage basé sur la restructuration des liens entre les pairs

Cette approche s'applique à l'équilibrage de charge de gestion d'index. En restructurant les liens entre les pairs, elle ajuste le trafic de routage qui passe par les pairs. Un ajustement approprié du trafic devrait mener le système à l'équilibre de la charge.

Expressways [ZSZ02] présente un exemple de cette approche. Son équilibrage de charge

tient compte de la différence de capacité des pairs, de l'inégalité et du dynamisme de la distribution de charge. Il vise à un état d'équilibre dans lequel les ratios de la charge sur la capacité de chacun des pairs sont égalisés.

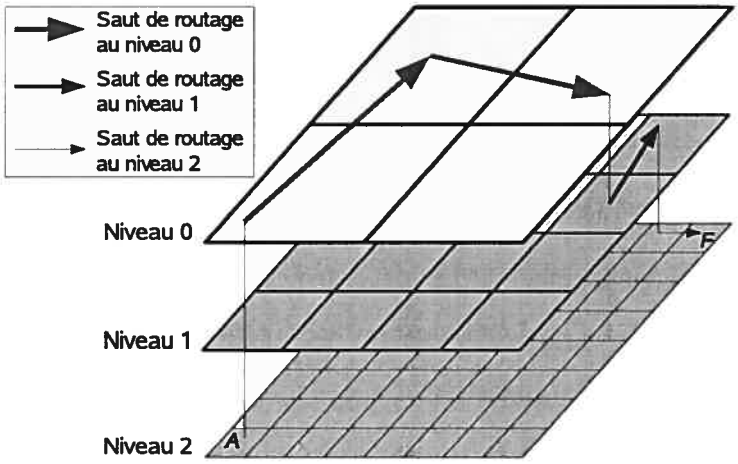

FIG. 3.4 – Exemple de routage (du pair A au pair F) dans Expressways ($d = 2$)

Expressways s'inspire de CAN [RFH+01] qui définit l'espace de clés comme un espace cartésien à d dimensions. Le système organise la gestion des clés comme un arbre virtuel dans lequel les feuilles sont les zones les plus petites correspondant aux zones de CAN et la racine représente l'espace cartésien global. Chaque zone interne dans l'arbre (appelée zone *expressways*) est composée de ces zones fils.

Un pair est responsable d'une zone feuille (dite z_l) et appartient à toutes les zones *expressways* qui couvrent z_l. Deux pairs sont voisins au niveau i si leurs zones à ce niveau sont adjacentes. Le tableau de routage d'un pair a plusieurs niveaux ; chacun maintient $O(d)$ liens vers ses voisins au niveau correspondant dans l'arbre. Les liens entre les zones *expressways* accélèrent le routage dans Expressways par rapport au routage de CAN. La figure 3.4 présente un exemple de routage dans un système Expressays ayant 64 pairs et $d = 2$. Le routage du pair A au pair F prend d'abord deux sauts au niveau 0, puis un saut au niveau 1 et enfin un saut au niveau 2. Le chemin est certainement plus court que celui qui utilise seulement

des sauts au niveau 2, niveau de CAN.

À cause de sa structure hiérarchique, les zones *expressways* aux niveaux le plus hauts supportent le plus de trafic de routage (selon la notation de fig. 3.4, le niveau 0 est le plus haut). On peut régler la charge d'un pair en ajustant les liens pour qu'il participe plus ou moins au trafic *expressways*. L'équilibrage de charge dans Expressways profite de cette propriété. Il consiste à agréger les informations de charge et de capacité des pairs du système entier, à déterminer la distribution appropriée sur les pairs et à ajuster les liens entre les pairs pour que la distribution prenne effet.

3.6 Synthèse

Nous avons énuméré différentes approches d'équilibrage de charge et présenté des méthodes utilisant ces approches. Bien que la plupart de ces méthodes considèrent la charge au sens général (c.-à-d., elle peut représenter la charge de stockage, la charge de trafic ou de gestion d'index, etc.), aucune méthode ne mentionne l'équilibrage de plusieurs types de charge en même temps.

Groupe	Méthode de	Ditribution des clés sur les pairs	Distribution des objets sur les clés	Popularité ou taille des objets	Capacité des pairs	Dynamisme des objets	Dynamisme des pairs
1	Bienkowski et al.	√					√
	Manku	√					√
	DH DHT	√					√
2	Karger et Ruhl	√	√	√		√	√
	Virtual Servers	√	√	√	√	√	√
	P-Grid	√	√	√		√	√
3	Byers et al.	√	√			√	√
	PAST	√	√	√	√	√	√
4	Expressways	√	√	√	√	√	√

TAB. 3.I – Considération des facteurs d'impact par les méthodes d'équilibrage de charche

Par ailleurs, ces méthodes supportent différents aspects des facteurs influençant l'équilibre de la charge. Le tableau 3.I compare les méthodes selon les aspects (sect. 3.1) qu'elles considèrent. Les méthodes du groupe 1 ne font que l'égalisation de la responsabilité des clés. Elles ne tiennent pas compte de la capacité des pairs et les facteurs reliés à la distribution de charge sur les clés.

Les méthodes du groupe 2 tiennent compte de la distribution de charge sur les clés. Bien qu'elles n'indiquent pas explicitement le lien entre la distribution des objets et des clés, la taille ou popularité des objets et le dynamisme des objets, la distribution de charge sur les clés en tient compte implicitement. Les méthodes de ce groupe se basent sur le transfert de clés. Si on les applique à l'équilibrage de charge de stockage, l'emplacement des objets est rattaché à l'emplacement de leurs clés. Ceci réduit la flexibilité et l'efficacité de l'équilibrage car le déplacement d'une clé entraîne obligatoirement le déplacement de tous les objets reliés à cette clé. En pratique, une clé est souvent assignée à un grand nombre d'objets de grande taille.

Les méthodes du groupe 3 équilibrent la charge au niveau des objets. Elles redistribuent les objets plutôt que les clés. Ceci est possible grâce à une relaxation de la relation entre le lieu de stockage et l'emplacement des clés. Dans les systèmes comme celui appliquant le paradigme *power of two choices* ou PAST, cette relaxation est encore restreinte : ils préservent des invariants qui limitent l'emplacement d'un objet sur certains pairs. L'évolution du système exige un coût de maintenance additionnel pour préserver ces invariants.

Tapestry [ZHS+04] est connu comme un substrat de routage qui sépare le stockage des objets de leur racine sans restriction. Un pair indique qu'il héberge un objet en mettant des pointeurs vers cet objet sur les pairs le long du chemin de routage entre lui et la racine de l'objet. Cette séparation permet à OceanStore [KBC+00], une application de stockage construit sur ce substrat, de changer le nombre et l'emplacement des répliques d'un objet.

Elle peut ainsi ajuster les requêtes d'accès à l'objet. Cependant, comme de nombreux autres substrats de routage, Tapestry ne supporte pas la séparation entre la responsabilité des clés et l'id des pairs. En fait, la responsabilité des clés d'un pair dépend de l'id du pair qui y est assigné dès la création et qui reste fixe durant le fonctionnement. Cette propriété empêche l'intégration d'un équilibrage de charge de gestion d'index efficace qui se base sur le transfert dynamique de la responsabilité de clés.

La méthode Expressways (groupe 4) s'attaque à la charge de gestion d'index. Un inconvénient de cette méthode est le grand nombre de liens nécessaires. Normalement, un grand degré des pairs entraîne un grand coût de maintenance lorsque le système est très dynamique.

L'objectif de notre recherche est de proposer une méthode d'équilibrage de charge gérant la charge de stockage et la charge de gestion d'index simultanément. Nous tenons compte de tous les facteurs d'impact mentionnés ci-dessus. Dans les méthodes citées, seuls Virtual Servers, PAST et Expressways considèrent tous les facteurs d'impact. De plus, chacune d'elles présente une approche d'équilibrage. Nous validerons nos résultats en analysant les principes de fonctionnement de ces méthodes et en identifiant leurs limites.

Les prochains chapitres présentent notre approche. Le chapitre 4 décrit notre méthodologie. L'ensemble des résultats suit dans les trois articles composant le cœur de cette thèse.

Chapitre 4

Approche intégrée d'équilibrage de charge : survol et évaluation

Ce chapitre présente la méthodologie que nous avons développée et adoptée. Nous présentons les mécanismes de notre système P2P et la méthodologie d'évaluation utilisée.

4.1 Approche BALLS : survol

Notre recherche vise à résoudre le problème d'équilibrage de charge intégré dans un système P2P structuré. Nous nous intéressons à deux charges importantes des systèmes P2P : la charge de stockage et la charge de gestion d'index. Notre solution doit tenir compte de tous les facteurs d'impact de l'équilibre des charges que nous avons identifiés dans la section 3.1 : distribution des clés sur les pairs, distribution des objets sur les clés, taille et popularité des objets, capacité des pairs, dynamisme des objets et dynamisme des pairs. La charge de stockage d'un pair est calculée en additionnant la taille de tous les objets que le pair stocke. D'autre part, la charge de gestion d'index est déterminée par le volume des messages de

routage passant par le pair au cours d'un intervalle de temps.

L'équilibrage intégré de charge de stockage et de charge de gestion d'index se base sur deux principes : (1) la séparation entre l'emplacement des objets et l'emplacement des clés et (2) la séparation entre les clés et l'id des pairs. Dans cette section, nous résumons nos méthodes d'équilibrage. On trouvera une description détaillée aux chapitres 5 et 7.

Nous décrivons d'abord notre système P2P. La structure P2P que nous proposons, nommée BALLS, partitionne un graphe de De Bruijn dont l'ensemble des nœuds correspond à l'espace de clés $[0, 2^m - 1]$. Chaque pair gère un intervalle continu de clés. Deux pairs quelconques se connectent si leurs intervalles sont liés par au moins un arc du graphe de De Bruijn ou sont adjacents dans l'espace circulaire de clés. Nous appelons « voisins » deux pairs connectés. Ils sont voisins de cercle (*ring neighbours*) si leurs intervalles sont adjacents. Cette structure garantit un algorithme de routage efficace (qui suit les chemins de routage dans le graphe de De Bruijn) et un degré constant des pairs.

DH DHT [NW03a] utilise une topologie similaire à celle de BALLS. Cependant, la contruction de ce système vise à égaliser les segments de clés gérés par les pairs. Pour le faire, DH DHT dispose de mécanismes d'arrivée et de départ des pairs compliqués qui exigent des structures de données additionnelles. Par contre, BALLS supporte l'équilibrage de charge qui accepte l'hétérogénéité des pairs, l'hétérogénéité des objets et le dynamisme. Dans ce contexte, l'égalité des segments ne suffit pas pour équilibrer la charge. Dans notre système, l'important n'est pas l'égalisation de la distribution des clés par les arrivées et départs mais les mécanismes d'auto-organisation en cours d'opération afin d'équilibrer la charge. BALLS garde ses protocoles d'arrivée et de départ les plus simples possibles à des fins d'efficacité et de fiabilité. En fait, l'arrivée d'un pair dédouble simplement la responsabilité des clés d'un pair existant qui gére une clé déterminée aléatoirement. D'autre part, le départ d'un pair transfère sa responsabilité de clés à un de ses voisins de cercle.

L'équilibrage de charge de gestion d'index se base sur le transfert dynamique des clés. La séparation entre les clés et l'id des pairs permet aux pairs de transférer des clés sans affecter leur id. Parce que le routage du système utilise les chemins de routage du graphe de De Bruijn, le transfert d'un intervalle de clés amène le transfert d'une charge de gestion d'index. Le but de l'équilibrage de charge est de minimiser la surcharge globale du système. Supposons qu'un pair p a la charge de gestion d'index T_p et la capacité C_p pour cette tâche. Sa surcharge, $O_p = (T_p - C_p + |T_p - C_p|)/2$, est la partie de T_p qui dépasse C_p. La surcharge globale est la somme des surcharges locales $\sum_{\forall p} O_p$. La minimisation de la surcharge globale se fait par des échanges entre les pairs.

Pour déterminer la charge de gestion d'index à transférer, un pair doit calculer la charge sur différentes portions de son intervalle de clés. L'enregistrement du trafic de routage attribuable à chaque clé est souvent impossible car un pair peut gérer des millions (ou plus) de clés. Pour pallier à ce problème, nous définissons une règle de division d'un intervalle en zones ; l'enregistrement du trafic de routage se fait alors au niveau de la zone. La division des zones doit satisfaire deux critères : (1) la mémoire requise pour l'enregistrement est efficace et (2) on peut déterminer les zones aux extrémités de l'intervalle (zones transférables) dont la taille varie de 1 à $s - 1$ (où s est la longueur de l'intervalle). On considère souvent un coût de mémoire logarithmique par rapport au nombre de clés comme efficace. Une division de l'intervalle en détaillant exponentiellement les zones aux extrémités (jusqu'à la taille égale à une clé) génère un nombre logarithmique de zones et satisfait le deuxième critère. Concrètement, un intervalle[1] $[b, e]$ (dont la longueur $s = e - b + 1$) est décomposé en zones $z_{i,j}$ ($i \in [0, \lfloor \log_2 s \rfloor - 1]$ et $j \in [0, 2]$) : $z_{i,0} = [b, b + l_i - 1]$, $z_{i,1} = [e - l_i + 1, e]$ et $z_{i,2} = [b, e] \setminus (z_{i,0} \cup z_{i,1})$ où $l_i = \lfloor s/2^{i+1} \rfloor$. Dans le cas particulier où $b = e$, les zones sont $z_{0,0} = \{b\}$ et $z_{0,1} = z_{0,2} = \emptyset$. Chaque intervalle a seulement $3\lfloor \log_2 s \rfloor$ zones $z_{i,j}$ ($\lfloor \log_2 s \rfloor$

[1] $[b, e]$ dénote un intervalle dans l'espace circulaire $[0, 2^m - 1]$. Si $b \leq e$, $[b, e] = \{x \in \mathbb{Z} \mid b \leq x \leq e\}$; sinon, $[b, e] = [b, 2^m - 1] \cup [0, e]$. Pour simplifier, toutes les expressions dans l'espace circulaire $[0, 2^m - 1]$ sont implicitement modulo 2^m. Par exemple, $(x + y)$ signifie $(x + y) \mod 2^m$.

est toujours limité par m). En enregistrant le trafic de routage passant par les zones $z_{i,j}$, nous pouvons calculer la charge de gestion d'index sur $4\lfloor \log_2 s \rfloor$ sous-intervalles d'une longueur variant entre 1 et $(s-1)$ aux deux extrémités de $[b, e]$ (c.-à-d., les zones $z_{i,j}$ et $z_{i,j} \cup z_{i,2}$).

Chaque pair p vérifie périodiquement son état de surcharge. Si la surcharge $O_p > 0$, p établit une liste de zones candidates à transférer ($z_{i,j}$ et $z_{i,j} \cup z_{i,2}$) pour diminuer O_p jusqu'à 0. Le pair p propose ces zones à un voisin de cercle q. Si $T_q < C_q$ et q trouve une zone candidate qui minimise la surcharge combinée $(O_p + O_q)$, cette zone est transférée. Sinon, p négocie le tranfert avec l'autre voisin de cercle. La minimisation de surcharge combinée des pairs diminue graduellement la surcharge globale.

Pour permettre à l'équilibrage de charge de stockage de fonctionner concurremment avec l'équilibrage de charge de gestion d'index, ces charges doivent être distribuées indépendamment. Le stockage des objets et leurs clés doivent être séparés. Un objet o n'est pas rattaché à sa racine. Au contraire, il peut se trouver sur n'importe quel pair. Afin de localiser l'objet, sa racine dispose d'un pointeur de redirection (appelé pointeur de stockage). Réciproquement, o maintient un pointeur vers sa racine (appelé pointeur de racine). Cette technique permet aux clés et aux objets de migrer dynamiquement entre les pairs. De plus, la réplication se fait de façon naturelle sans avoir besoin de technique additionnelle. En effet, un objet peut être répliqué en κ copies mises sur κ pairs différents. La racine et les répliques maintiennent des pointeurs entre eux comme avec des objets individuels.

Chaque pair p a un espace limité pour le stockage, appelé capacité de stockage D_p. La charge de stockage S_p ne doit jamais dépasser D_p. Lors de l'insertion d'un objet o sur le pair p, si l'espace D_p ne suffit pas, o doit chercher un autre pair sur lequel il sera stocké. Cela augmente la performance de stockage. En plus de l'espace de stockage, chaque pair utilise d'autres ressources (p.ex., bande passante, CPU) pour gérer l'accès et la migration des objets. On voudra que l'approche d'équilibrage tienne également compte de ces autres ressources.

Nous définissons une autre limite de la charge de stockage, $\overline{D}_p \leq D_p$, que nous appelons la capacité de stockage désirée. La surcharge de stockage, $W_p = (S_p - \overline{D}_p + |S_p - \overline{D}_p|)/2$, dénote la partie de S_p qui dépasse $\overline{D}_p$. Le but de l'équilibrage de charge de stockage est la minimisation de la surcharge globale $\sum_{\forall p} W_p$.

Chaque pair p vérifie périodiquement s'il est surchargé. Si $W_p > 0$, p diffuse une demande d'espace disponible à ses voisins. Les pairs non-surchargés répondent à cette demande. À la réception de chaque réponse d'un pair q, p propose à q un ensemble d'objets R_{pq}. Le pair q choisira un sous-ensemble $R'_{pq} \subseteq R_{pq}$. Si le transfert s'arrête ici, on l'appelle transfert unidirectionnel. Dans certains cas, il sera nécessaire que q propose à p un certain ensemble d'objets R_{qp} à transférer de q à p. Le pair p choisira un sous-ensemble $R'_{qp} \subseteq R_{qp}$. Un tel transfert est appelé transfert bidirectionnel. Les deux types de transfert visent à minimiser la surcharge combinée $(W_p + W_q)$. Ils contribuent graduellement à la minimisation de la surcharge globale.

4.2 Évaluation

L'évaluation de la solution proposée se fait par simulations. Nous présentons ici les simulations, les outils de simulation et les méthodes d'analyse statistique utilisées sur les résultats observés.

4.2.1 Simulations

L'évaluation consiste à vérifier la performance de la topologie de BALLS et des méthodes d'équilibrage des charges. Comme avec d'autres infrastructures P2P, on s'intéresse à la performance du point de vue de l'efficacité de routage et de l'efficacité de maintenance. Nous mesurons le coût de routage (en nombre de sauts requis dans un chemin de routage)

avec différentes tailles du réseau. Pour observer l'efficacité de maintenance, nous mesurons les coûts d'arrivée et de départ des pairs (en nombre de messages expédiés pour maintenir la topologie), ainsi que le degré des pairs, ici encore avec différentes tailles du réseau. Ces mesures nous permettent également d'évaluer la relation entre les coûts d'arrivée et de départ, et le degré des pairs. Cette relation est estimée linéaire en théorie (sect. 7.2.1).

La vérification de la performance des méthodes d'équilibrage de charge s'intéresse à leur efficacité et convergence. Nous définissons des métriques et des scénarios de simulation pour les mesurer. Comme nous l'avons expliqué préalablement, le but de l'équilibrage est de minimiser la surcharge globale. La métrique utilisée doit indiquer la variation de la surcharge globale obtenue par l'équilibrage. Elle doit également tenir compte de la relation entre la surcharge globale et la charge globale du système. À ces fins, nous définissons la métrique comme le ratio de la surcharge sur la charge. Dans l'équilibrage de charge de gestion d'index, le ratio de surcharge de gestion d'index est $\Omega = (\sum_{\forall p} O_p)/(\sum_{\forall p} T_p)$. Dans l'équilibrage de charge de stockage, nous avons le ratio de surcharge de stockage $\Psi = (\sum_{\forall p} W_p)/(\sum_{\forall p} S_p)$.

Nous évaluons l'efficacité et la convergence de l'équilibrage de charge de gestion d'index avec un seul scénario : scénario à trois phases. La phase 1 dure assez long temps sans équilibrage pour obtenir la valeur maximale de Ω. En phase 2, on active l'équilibrage et le laisse fonctionner assez long temps pour qu'il prenne complètement effet. La comparaison de Ω entre les phases 1 et 2 nous permet d'observer l'efficacité de l'équilibrage. En phase 3, on désactive l'équilibrage. La stabilité de Ω à la phase 3 par rapport au niveau établi à la phase 2 détermine la convergence de l'équilibrage.

L'éfficacité et la convergence de l'équilibrage de charge de gestion d'index dépend aussi des facteurs environnementaux : le niveau d'utilisation de la capacité des pairs, le dynamisme des objets et le dynamisme des pairs. Notre évaluation simule ces facteurs via les paramètres suivants : le ratio d'utiliation ($U = (\sum_{\forall p} T_p)/(\sum_{\forall p} C_p)$), le dynamisme de la popularité de

clé (la probabilité τ du changement de popularité d'une clé dans une unité de temps) et le dynamisme des pairs (le pourcentage π de pairs arrivant ou partant dans une unité de temps). La comparaison des résultats produits par différentes valeurs de ces paramètres est nécesaire pour découvrir leur impact sur l'équilibrage.

L'évaluation de l'efficacité et de la convergence de l'équilibrage de charge de stockage est faite de façon séparée. Pour observer l'efficacité, nous comparons le ratio de surcharge Ψ produit au cours de deux simulations, l'une avec équilibrage et l'autre sans équilibrage. Pendant l'opération, le taux d'utilisation de stockage ($Z = (\sum_{\forall p} S_p)/(\sum_{\forall p} \overline{D}_p)$) augmente de temps en temps (de 0% à 150%). La comparaison des deux modes (avec et sans équilibrage) sur les mêmes valeurs de Z révèle l'efficacité de la méthode.

Encore une fois, il y a deux facteurs qui affectent l'efficacité de l'équilibrage : le mode de transfert utilisé (orienté surcharge ou orienté coût, sect. 7.2.4) et le dynamisme des pairs (π). On doit évaluer l'éfficacité par différentes valeurs de ces paramètres.

L'évaluation de la convergence se réalise avec le scénario suivant : on initialise le ratio d'utilisation de stockage à un certain niveau et puis on active l'équilibrage sans changer la capacité totale ni la charge totale. Si l'équilibrage diminue le ratio de surcharge Ψ sans l'augmenter, il montre la convergence. Les métriques de cette évaluation incluent le ratio de surcharge final (appelé le ratio de surcharge stable Ψ_f), le temps pour atteindre Ψ_f (appelé le temps de stabilisation) et le ratio du coût de migration sur la surcharge globale initiale. L'évaluation comprend aussi la comparaison entre deux modes de transfert et entre différents niveaux de Z pour observer l'impact de ces facteurs.

Rappelons que nous résolvons le problème d'équilibrage intégré. L'évaluation séparée des méthodes d'équilibrage ne suffit donc pas. Nous mesurons l'impact de l'équilibrage d'une charge (de gestion d'index ou de stockage) sur l'équilibrage de l'autre aspect et vice versa. En fait, l'évaluation comprend quatre cas : (00) sans équilibrage de charge de gestion d'index

ni équilibrage de charge de stockage, (01) sans équilibrage de charge de gestion d'index mais avec équilibrage de charge de stockage, (10) avec équilibrage de charge de gestion d'index mais sans équilibrage de charge de stockage et (11) avec équilibrage de toutes les deux charges. La comparaison de ces quatre cas sur Ω et Ψ permet de déterminer si les méthodes d'équilibrage sont indépendantes ou non.

Enfin, nous avons besoin de simulations qui vérifient les inconvénients identifiés dans les méthodes connexes. Notre but n'est pas de faire une comparaison directe entre BALLS et les autres systèmes, mais plutôt de vérifier dans quelle mesure la présence de ces inconvénients rend ces systèmes moins performants que BALLS. En théorie, l'application des serveurs virtuels dans l'équilibrage de charge de stockage empêche le fonctionnement concurrent de l'équilibrage de charge de gestion d'index, car les serveurs virtuels attachent le lieu de stockage des objets à celui de leurs clés. Nous démontrons ceci en utilisant les quatre cas identifiés pour l'évaluation de l'équilibrage intégré de BALLS.

Dans notre discussion sur PAST, nous indiquons que la restriction de résidence d'un fichier (dans l'ensemble *leaf* de la racine) et la maintenance de cet invariant mènent à un coût de migration des fichiers quand le système évolue. Des simulations de PAST mesurent ce coût après un certain nombre d'insertions de pairs. Si le coût de migration est considérable, il démontre l'inconvénient de la relaxation restreinte de stockage dans les systèmes comme PAST par rapport à BALLS, qui sépare complètement le stockage de la clé. Ce coût n'existe pas dans notre approche puisque les objets ne sont pas déplacés à l'arrivée d'un pair.

Nous nous intéressons aussi à Expressways. Ce système équilibre la charge de gestion d'index en se basant sur la restructuration des liens des pairs. Cette méthode exige un grand degré des pairs (estimé $O(\log n)$). Des simulations similaires à celles utilisées pour vérifier la topologie de BALLS seront exécutées sur un simulateur d'Expressways. Elles mesurent le degré des pairs sous différentes tailles du système. La dominance du degré mesuré par

rapport à celui de BALLS exprime l'avantage de l'application du graphe de De Bruijn.

4.2.2 Simulateurs

Pour évaluer des systèmes P2P, les simulateurs sont des outils efficaces. Nous avons développé le simulateur de BALLS pour vérifier la performance de cette topologie, des mécanismes d'équilibrage de charge et de l'équilibrage intégré. L'implémentation du simulateur se fait en Java. Nous présentons dans cette section les points saillants du simulateur. Plus de détails sur le simulateur se trouveront dans le chapitre 6.

Le simulateur de BALLS supporte deux modes : *simulation centralisée* et *simulation décentralisée*. Le mode centralisé exécute tout le système (incluant tous les pairs) sur un seul ordinateur physique. Les pairs communiquent via un accès direct à la file des messages des autres pairs. Les mesures globales sont faciles à réaliser et efficaces dans ce mode. Le mode décentralisé exécute chaque pair sur une machine séparée. Les pairs communiquent à travers un réseau réel qui connecte les machines. Ce mode n'est pas efficace pour évaluer un gros système. Cependant, avec un nombre limité de pairs, la simulation décentralisée permet d'évaluer le comportement du système en tenant compte des facteurs en temps réel et de la bande passante réelle d'un réseau. Les résultats que nous présentons dans ce travail n'utilisent pas encore ce mode de simulation. L'implémentation de ce mode constitue le premier pas vers une implémentation du système BALLS.

L'architecture du simulateur consiste en trois couches :

- *couche réseau* qui prend en charge la configuration du système et la communication des pairs. Elle comprend une partie centralisée et une partie décentralisée qui supportent les modes de simulation correspondants. Cette distinction vient de la différence entre les deux modes dans la construction et le fonctionnement du système ;
- *couche des pairs* qui définit les opérations des pairs, par exemple, routage, arrivée,

départ, équilibrage de charge et gestion des objets. Cette couche est commune pour les deux modes de simulation ;

- *couche d'évaluation* qui supporte les fonctions d'évaluation. Elle comprend une partie centralisée et une partie décentralisée selon le mode de simulation. Comme ces modes exigent des stratégies d'évaluation différentes, les deux parties de la couche d'évaluation fonctionnent différemment. La partie centralisée fournit les mesures globales du degré des pairs, du coût d'arrivée, du coût de départ, du coût de routage et des métriques d'évaluation de l'équilibrage de charge (charge, surcharge, capacité, etc.). La partie décentralisée doit disposer d'un serveur central qui agrège les mesures provenant des pairs pour agréer le résultat global. Dans la présente version du simulateur, cette partie supporte seulement l'évaluation de l'équilibrage de charge.

L'architecture à trois couches réduit notre effort dans la mise à jour et l'évolution du simulateur. Les modifications à la couche des pairs prennent effet dans les deux modes de simulation. D'ailleurs, on peut facilement ajouter de nouvelles évaluations et/ou développer des simulateurs pour d'autres modèles P2P.

Nous avons aussi développé des simulateur pour évaluer des inconvénients identifiés dans l'application des serveurs virtuels, PAST et Expressways pour fin de validation de l'avantage de BALLS. Le simulateur des serveurs virtuels implémente l'équilibrage de charge de gestion d'index et l'équilibrage de charge de stockage en se basant sur le transfert des serveurs virtuels. L'équilibrage de charge de gestion d'index utilise un algorithme similaire à celui de BALLS. Le simulateur supporte les mesures globales de surcharge, charge et capacité pour évaluer la performance de l'équilibrage.

Le simulateur de PAST permet aux pairs d'entrer et de sortir dynamiquement afin de faire évoluer le système. Il mesure le volume et le nombre total de fichiers déplacés lorsque leur frontière de résidence bouge. Ceci permet d'observer le coût de migration causé par l'évolution du système.

Le simulateur d'Expressways permet aussi les entrées et départs dynamiques des pairs. Il mesure le nombre de liens maintenus par les pairs sous différentes tailles du réseau.

Ces simulateurs sont les plus simples possibles. Ils ne supportent que les métriques considérées et omettent tous les éléments et fonctionnement non-nécessaires. Nous gardons la simplicité pour assurer l'efficacité et l'exactitude de l'opération.

4.2.3 Analyses statistiques

Pour chaque simulation décrite, nous réalisons au moins 20 exécutions afin d'obtenir des résultats valides. Ainsi, on obtient pour chaque métrique un ensemble de valeurs. Ceci requiert des analyses statistiques.

Nous faisons deux analyses sur les résultats obtenus : *analyse de l'intervalle de confiance* et *analyse comparative*. Le résultat d'une évaluation est souvent présenté sous forme d'un graphe de la moyenne des valeurs mesurées. La première analyse évalue la confiance de la mesure des courbes présentées. L'intervalle de confiance avec une probabilité α d'une variable y est $[a, b]$ tel que $P(\overline{y} \notin [a, b]) \leq \alpha$ où $\overline{y}$ est la moyenne de y. Étant données n valeurs $y_1, \cdots, y_n$ mesurées et la valeur moyenne y^* estimée sur ces valeurs, on obtient $[a, b]$ en utilisant la loi t-student et la variance (S^2) :

$$
\begin{aligned}
a &= y^* - \frac{t_{n-1,\alpha/2} \cdot S}{\sqrt{n}} \\
b &= y^* + \frac{t_{n-1,\alpha/2} \cdot S}{\sqrt{n}} \\
S^2 &= \frac{\sum_{i=1}^{n} y_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} y_i\right)^2}{n-1}
\end{aligned}
$$

où $t_{n-1,\alpha/2}$ est la valeur critique supérieure de la distribution t-student correspondant au degré

de liberté $(n-1)$ et à la probabilité $\alpha/2$.

Revenons à l'analyse de l'intervalle de confiance d'un graphe. Avec chaque valeur sur l'axe horizontal x, nous calculons l'intervalle de confiance de toutes les valeurs y correspondantes mesurées. Plus étroits sont les intervalles par rapport à l'échelle du graphe (l'échelle est la distance maximale du graphe à 0), plus la valeur calculée est proche de la vraie valeur. Nous appliquons cette analyse dans l'évaluation du degré des pairs, du coût d'arrivée, du coût de départ et du coût de routage. Tous les calculs utilisent la probabilité $\alpha = 1\%$.

L'analyse comparative détermine si deux (ou plus) graphes sont différents ou pas. Elle sert à vérifier l'impact des conditions de simulation sur les résultats obtenus. Supposons qu'on compare deux graphes X et Y. Pour chaque valeur i sur l'axe horizontal, nous déterminons l'intervalle de confiance de $(X_i - Y_i)$. Si l'intervalle ne couvre pas 0, X_i et Y_i sont statistiquement différentes.

Nous considérons le taux des paires (X_i, Y_i) pour lesquelles l'intervalle de confiance de $(X_i - Y_i)$ est disjoint de 0, pour toutes les valeurs i considérées. Nous appelons ce taux le taux de différence. Un taux de différence élevé entre deux graphes exprime leur différence statistique. Au contraire, un taux bas (p.ex., inférieur à 10%) ne conclut pas à leur différence. La comparaison entre plus de deux graphes implique la comparaison entre toutes les paires de graphes. Cette analyse est appliquée aux comparaisons suivantes :

- comparaison de l'impact des niveaux de dynamisme de la popularité de clé sur l'équilibrage de charge de gestion d'index,
- comparaison de l'impact des deux modes de transfert sur l'équilibrage de charge de stockage, et
- comparaison entre les scénarios d'évaluation de l'équilibrage intégré (dans BALLS et dans l'application des serveurs virtuels).

Chapitre 5

Article : A structured peer-to-peer system with integrated index and storage load balancing [LBK06c]

Cet article[1] introduit la partie théorique de la solution. Son contenu comprend la description de la structure du système P2P (BALLS) et les mécanismes d'équilibrage de charge de stockage et de gestion d'index. Il sert de jalons pour notre recherche après les définitions théoriques. Les concepts généraux ont été proposés par Viet Dung Le. La collaboration entre Viet Dung Le et les professeurs Gilbert Babin et Peter Kropf a permis de compléter les détails spécifiques. L'article a été présenté lors de la 5^{e} conférence *Innovative Internet Community Systems* (20-22 juin 2005, à Paris, France) et publié dans *Lecture Notes in Computer Science*, volume 3908, Springer Berlin / Heidelberg (avril 2006)[2]. Au moment de la soumission, quelques résultats préliminaires d'évaluation étaient disponibles. Cependant,

[1] Viet Dung Le, Gilbert Babin, and Peter Kropf. A structured peer-to-peer system with integrated index and storage load balancing. In *Innovative Internet Community Systems (I2CS'05), LNCS 3908, Springer Berlin / Heidelberg*, pages 41-52, 2006.

[2] L'article est reproduit dans la thèse avec la permission de « *Springer Science and Business Media* ».

l'évaluation et les points détaillés des méthodes se trouvent dans un article ultérieur.

Abstract

Load balancing emerges as an important problem that affects the performance of structured peer-to-peer systems. This paper presents a peer-to-peer system relying on the partitionning of a de Bruijn graph. The proposed system integrates mechanisms that perform index and storage load balancing. Index load refers to the network traffic incurred by a peer in managing an object index, while storage load refers to the storage space and network traffic required to store objects. The proposed mechanisms allow to effectively distribute both index load and storage load according to the peers' capacities.

5.1 Introduction

A peer-to-peer (P2P) system comprises multiple parties (called *peers*) that can request and provide services at the same time. This decentralized characteristic furthers spreading of workload among all participating peers and thus contributes to solutions for scalability issues in distributed systems. However, in comparison to a centralized system, managing shared objects becomes difficult because of the lack of a central or hierarchical control. Structured P2P systems, such as [FG03, KK03, MNR02, NW03a, RFH+01, RD01a, SMK+01, ZKJ01], introduce efficient mechanisms to store and access these distributed objects. The principle inherent to such systems consists in mapping every object onto a key space or index (e.g., by hashing the object identifier), distributing this key space over the available peers, and maintaining a structured connection among the peers according to the keys each peer holds. The connection structure ensures to guide the search for an object to the peer responsible for the object's key in a small number of hops, often $O(\log n)$ in an n-peer system.

The system performance of a P2P network is critically affected by its overload. Indeed, the storage or processing load of the peers, the communication load and the system management load must be carefully handled to obtain satisfactory system performances which may be regarded as the fastest possible response time to user/application requests. Workload distribution and balancing mechanisms contribute to achieve good system performance. However, they may induce expensive restructuring processes, i.e., maintenance costs. Our approach aims to balance workload in P2P systems while keeping maintenance costs low. We are interested in two workload aspects : index load and storage load. In P2P systems, finding an object usually requires routing requests through intermediate peers before arriving at destination. The bandwidth used for this task makes up the index load on each peer. The storage load, on the other hand, denotes the usage of each peer's resources in object accommodation. Many load balancing approaches have been proposed. However, to our knowledge, none takes into account these two aspects of load simultaneously.

The present paper introduces a solution that simultaneously handles both index and storage load balancing by separating the concerns of peer identifiers (addresses), key management, and object storage locations. In particular, the proposed P2P structure is based on partitioning a de Bruijn graph where the node identifier space is identical to the key space. Therefore, we will use key and de Bruijn node exchangeably. Each peer holds a non-empty interval of de Bruijn nodes and maintains connections to other peers that hold neighbouring de Bruijn nodes. Based on this structure, looking for a specific key in the P2P system follows appropriate routing paths in the de Bruijn graph.

The **index load balancing** method takes into account the network capacity of the peers. It aims to minimize the network overload that may occur while routing requests in the system. This goal is different from that of most other methods which permanently adjust the load to a target. Since the decrease of the overload reacts only when an overload exists, our method saves on the costs of rebalancing. The balancing method involves two tasks : (1) lo-

cally calculating the index load on every peer and (2) dynamically transferring index load from peer to peer by modifying the key interval managed by each peer. We propose efficient mechanisms to perform these two tasks.

The **storage load balancing** method is based on separating the key and the storage location of objects. It eliminates the restriction of an object's residence to its root, where the root refers to the peer responsible for the key interval which includes the object's key. Instead, the root needs only to keep pointers to the location of its objects. This separation enables and facilitates the index load balancing since the move of a key interval from peer to peer entails moving only the involved object pointers (very small in size) instead of the objects themselves. Thus, moving keys does not affect the storage load. Without restriction to the root, the accommodation of objects chooses the storage location such that the storage load on every peer does not exceed the contributed storage capacity. In addition, we take into account the capacity of the peers in serving object requests and migration. We propose a balancing algorithm that minimizes the peer's overload with regards to its capacity. Like the index load balancing, the consideration of overload in this algorithm minimizes rebalancing costs. The algorithm is based on exchanging appropriate objects among pairs of peers in order to decrease the overload whenever it occurs. Finally, a fair advantage of separating key and storage location is the replication facility. The root of an object can maintain a set of pointers to its replicas (placed on different peers). Thus, the object availability is enhanced without further replication techniques.

The rest of this paper is organized as follows. Section 5.2 summarizes some recent work on load balancing in structured P2P systems. Sections 5.3 and 5.4 respectively describe the methods of index load and of storage load balancing that can operate simultaneously. The last section provides some discussion.

5.2 Related work

A straightforward approach to load balancing in a structured P2P system is the equalization of the key occupation among the peers (e.g., [BKadH05, Man04]). The equalization in [BKadH05] stochastically makes peers with short key intervals leave and rejoin the system by splitting peers with long key intervals. The method proposed in [Man04], on the other hand, balances a virtual binary tree whose leaf nodes represent the participating peers. In practice, load balance depends also on the distribution of objects on the peers, the object size, and the storage, processing, and communication capacity of the peers. Equalizing key occupation does not ensure an even load distribution when taking into account all these different factors making up the load.

The application of the *power of two choices* paradigm [BCM03] applies multiple hash functions to map each item to multiple peers. This allows to insert an item on the least loaded peer. The methods in [KR04a, RLS+03] achieve load balance by exchanging key responsibility among the peers. The above approaches cannot simultaneously balance the index load and the storage load because they associate the storage location and the key. Balancing one workload aspect can break the balance of the other one, and vice versa. *PAST* [RD01b] uses a replica diversion process to balance the storage load. However, the concerns of storage location and file identifier in *PAST* are not separated. It maintains an invariant that limits the storage location of a file to the *leaf* sets (see [RD01a] for definition) of a number k of peers. The maintenance of this invariant introduces considerable overhead in a dynamically changing P2P system, e.g., a system with index load balancing.

Expressways [ZSZ02], an extension of *CAN* [RFH+01], proposes an index load balancing method. It structures the network (of size n) as a hierarchy of $\log n$ levels, each one operating like a basic *CAN*. The balancing method is based on promoting peers with higher bandwidth to higher levels in the hierarchy. However, the reaction of the system to balance

the load takes place only after aggregating the loads and the capacities of all peers in the system. Moreover, keeping each peer's and the overall system's load/capacity ratio equal can constantly bring the system to restructure itself even if individual peers would not require rebalancing.

The P2P systems introduced in [LKRG03, NW03a, WZLL04] employ the partition of a de Bruijn graph. Like [BKadH05, Man04], they aim at equalizing key occupation. As discussed above, this equalization is not sufficient for load balancing in structured P2P systems.

5.3 Index load balancing

5.3.1 System structure and routing

Our proposed P2P network partitions a binary de Bruijn graph $G(V, A)$ of 2^m nodes. The key space is identical to the de Bruijn node identifier space $V = [0, 2^m - 1]$[3]. Obviously, with a large enough m, the number of peers in a real network does not attain 2^m. Each peer holds and is responsible for a non-empty interval of de Bruijn nodes (also called *key interval*). Every peer is identified by its network (e.g., IP) address. Given a peer p, we denote :

- $p.a$ – the address of p,
- $p.b$ and $p.e$ – respectively the beginning and ending keys of p's key interval,

Two peers p and q must connect, denoted $connect(p, q)$, if there exists at least one arc between any two de Bruijn nodes that fall within the key intervals of p and q respectively, or

[3] $[b, e]$ denotes the interval of integers from b to e (inclusive). If $b \leq e$, $[b, e] = \{x \in \mathbb{Z} \mid b \leq x \leq e\}$, otherwise, $[b, e] = [b, 2^m - 1] \cup [0, e]$.

if their key intervals are numerically adjacent.

$$connect(p,q) = \begin{cases} \text{true} & \text{if } (\exists x, y \mid (x,y) \in A \wedge x \in [p.b, p.e] \wedge y \in [q.b, q.e]) \\ & \quad \vee (p.e = (q.b - 1) \bmod 2^m) \vee (p.b = (q.e + 1) \bmod 2^m) \\ \text{false} & \text{otherwise} \end{cases}$$

These connections are bidirectional, i.e., if $connect(p,q)$ then $connect(q,p)$. Two connecting peers are called neighbours. Each peer maintains a ***neighbour list*** consisting of a triple $(q.a, q.b, q.e)$ for each neighbour q in the list. The separation between peer address and key means that the peers can dynamically change their key interval $[b,e]$ without affecting the address a.

Loguinov et al. [LKRG03], and Naor and Weider [NW03a] introduced a similar structure based on the de Bruijn graph. Their goal is to balance the partitioned zone sizes through different arrival/departure mechanisms. Our focus, however, is in balancing mechanisms taking into account the storage capacity and communication capacity of peers.

The routing function consists in directing a message to the root of a given key x from anywhere in the system. The message follows appropriate de Bruijn routing paths towards x. For convenience, all expressions on the de Bruijn node identifiers are implicitly modulo 2^m, e.g., $x + y$ means $(x + y) \bmod 2^m$. We also refer to the beginning and ending values of interval I as $I.b$ and $I.e$, respectively.

Definition 1 *The distance between two keys x and y, denoted $distance(x,y)$, is the minimum among the length of the de Bruijn routing paths[4] from x to y and from y to x.*

Definition 2 *The distance between a key interval I and a key x, denoted $distance(I,x)$, is equal to $distance(v,x)$ where $v \in I$ and $\nexists v' \in I \mid distance(v',x) < distance(v,x)$.*

[4]Note that the de Bruijn routing path between two nodes in an undirected de Bruijn graph is not always the shortest path.

In the de Bruijn graph of 2^m nodes, each node x has four arcs respectively to nodes $2x$, $2x+1$, $\lfloor x/2 \rfloor$, and $\lfloor (x+2^m)/2 \rfloor$. Let the arcs to $2x$ and $2x+1$ be the fore-arcs and the arcs to $\lfloor x/2 \rfloor$ and $\lfloor (x+2^m)/2 \rfloor$ be the back-arcs. We use notation $foredistance(x,y)$ to specify the length of the routing path following only fore-arcs from x to y. Similarly, the notation $backdistance(x,y)$ specifies the length of the routing path following only back-arcs. By Definition 1,

$$distance(x,y) = min(foredistance(x,y), backdistance(x,y)).$$

Claim 1 *Given a node x, the set of every node y such that $foredistance(x,y) = i$ (with $i \in [0,m]$), denoted $F_i(x)$, is $[x2^i, x2^i + 2^i - 1]$.*

Proof: If $i = 0$, it is clear that $F_0(x) = \{x\}$.

If $i > 0$, suppose that $F_{i-1}(x) = [x2^{i-1}, x2^{i-1} + 2^{i-1} - 1]$ is correct. Following the fore-arcs of all nodes in $F_{i-1}(x)$, we have

$$F_i(x) = \bigcup_{y \in F_{i-1}(x)} F_1(y) = [x2^{i-1}2, (x2^{i-1} + 2^{i-1} - 1)2 + 1] = [x2^i, x2^i + 2^i - 1] \qquad \square$$

Claim 2 *Given a node x, the set of every node y such that $backdistance(x,y) = i$ (with $i \in [0,m]$), denoted $B_i(x)$, is $\{y_0, y_1, \cdots, y_{2^i-1}\}$ where $y_j = \lfloor x/2^i \rfloor + j2^{m-i}$.*

Proof: If $i = 0$, it is clear that $B_0(x) = \{x\}$.

If $i > 0$, suppose that $B_{i-1}(x) = \{y_0, y_1, \cdots, y_{2^{i-1}-1}\}$ where $y_j = \lfloor x/2^{(i-1)} \rfloor + j2^{m-(i-1)}$ is correct. Following the back-arcs of all y_j, we have

$$B_i(x) = \bigcup_{j \in [0, 2^{i-1}-1]} B_1(y_j)$$

where

$$\begin{aligned} B_1(y_j) &= \{\lfloor y_j/2 \rfloor, \lfloor (y_j + 2^m)/2 \rfloor\} \\ &= \{\lfloor (\lfloor x/2^{(i-1)} \rfloor + j2^{m-(i-1)})/2 \rfloor, \lfloor (\lfloor x/2^{(i-1)} \rfloor + j2^{m-(i-1)} + 2^m)/2 \rfloor\} \\ &= \{\lfloor x/2^i \rfloor + j2^{m-i}, \lfloor x/2^i \rfloor + (j + 2^{i-1})2^{m-i}\} \end{aligned}$$

For all $j \in [0, 2^{i-1} - 1]$, the pair $(j, j + 2^{i-1})$ gives all integers in $[0, 2^i - 1]$. □

Given a key interval I and a key x, the $distance(I, x)$ algorithm calculates $F_i(x)$ and $B_i(x)$ for i from 0 to m. If at an iteration d, $F_d(x)$ or $B_d(x)$ has common keys with I, it returns d. This algorithm is efficient because it iterates testing $F_i(x)$ and $B_i(x)$ for at most $m + 1$ times before finding the distance.

Definition 3 *The de Bruijn neighbourhood set of a key interval I, denoted $dbneighbour(I)$, is the set* $([I.b \times 2, (I.e \times 2) + 1] \cup [\lfloor I.b/2 \rfloor, \lfloor I.e/2 \rfloor] \cup [\lfloor (I.b + 2^m)/2 \rfloor, \lfloor (I.e + 2^m)/2 \rfloor]) \setminus I$.

Routing : the following algorithm routes a message from the current peer p to the peer holding key x.

1. if $x \in [p.b, p.e]$, peer p is the destination. Otherwise, continue with step 2 ;
2. calculate the set $U = dbneighbour([p.b, p.e])$. Find $t \in U$ such that $distance(t, x) = distance(U, x)$. Select neighbour q such that $t \in [q.b, q.e]$. Then continue routing to x from q.

The set U may contain several key intervals. We use here the notation $distance(U, x)$ to refer to the minimal distance from the intervals in U to x. The key t satisfying the equality $distance(t, x) = distance(U, x)$ is easily found : we select the key from the intersection of $F_i(x)$ or $B_i(x)$ and the interval (in U) the nearest to x while calculating the distance. This

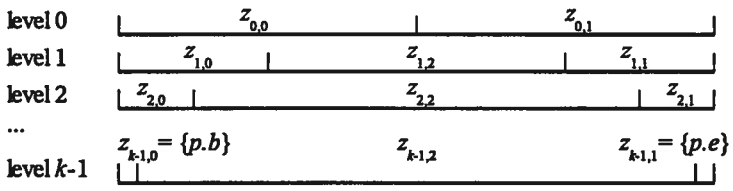

FIG. 5.1 – Zone division at k levels on a peer p

algorithm ensures to reduce the distance from the current position t to x by at least 1 after each hop. The number of routing hops is therefore bound by m.

5.3.2 Index load calculation

The index load of a peer is defined as the sum of routing message sizes passing through the peer in a unit of time. The idea of index load balancing is to transfer key intervals between peers to minimize the overload. It requires to calculate the routing traffic on different subsets (which we call zones) of each peer's key interval. For large key intervals, registering the routing traffic through all keys is inefficient or even unrealizable. To make this monitoring efficient, we restrict key interval movements. First, a peer p will only transfer keys to the peers holding $p.b - 1$ or $p.e + 1$. Second, the size of the interval transferred should range from 1 to $s - 1$ where s is the whole key interval's size. We further simplify the monitoring by dividing each peer p's key interval into k levels, where $k = \lfloor \log_2(p.e - p.b + 1) \rfloor$. Levels are further broken down into 3 zones. Figure 5.1 depicts this division.

At each level i ($0 \leq i < k$), l_i, the length of zone $z_{i,0}$, is given by :

$$l_i = \begin{cases} \lfloor (p.e - p.b + 1)/2 \rfloor & \text{if } i = 0 \\ \lfloor l_{i-1}/2 \rfloor & \text{if } 0 < i < k \end{cases}$$

Then, we have the zones : $z_{i,0} = [p.b, p.b + l_i - 1]$, $z_{i,1} = [p.e - l_i + 1, p.e]$, and

$z_{i,2} = [p.b, p.e] \setminus (z_{i,0} \cup z_{i,1})$. It follows that $z_{k-1,0} = \{p.b\}$ and $z_{k-1,1} = \{p.e\}$. In the special case where $p.b = p.e$, there exists only one level with $z_{0,0} = \{p.b\}$ and $z_{0,1} = z_{0,2} = \emptyset$.

Each peer p constructs a table $G_p[k][3]$. $G_p[i,j]$ registers the routing traffic through zone $z_{i,j}$. This table does not consume much memory space since $k < m$. According to the routing algorithm, when a message λ passes through peer p, λ is oriented via a de Bruijn node $t \in [p.b, p.e]$. For every level i, if $t \in z_{i,j}$ then $G_p[i,j] = G_p[i,j] + |\lambda|$ (where $|\lambda|$ denotes the size of λ). Obviously, the total routing traffic on peer p is $Tr_p = \sum_{j \in [0,2]} G_p[i,j]$, for any i.

Each peer p has a routing traffic capacity C_p. It verifies the index load periodically. We denote the period duration as δt, the beginning time of the current period as t_0, and the current time as t_c. Then, the current index load is $T_p = Tr_p/(t_c - t_0)$. In case $t_c - t_0$ is too small and may produce $Tr_p/(t_c - t_0)$ reflecting an incorrect index load of p, we calculate the load as $T_p = (Tr'_p + Tr_p)/(t_c - t'_0)$ where Tr'_p and t'_0 are, respectively, the routing traffic and the beginning time of the previous period. If $T_p > C_p$, peer p is overloaded. At the end of each verification period, if p is overloaded, it executes the index load balancing algorithm and starts a new period. Any change of $p.b$ or $p.e$ involves also a new period. The beginning of every period resets k and table G_p.

5.3.3 Index load balancing algorithm

When a peer p discovers that it is overloaded ($T_p > C_p$), it should transfer an appropriate key interval $z_{i,0}$, $z_{i,1}$, $z_{i,0} \cup z_{i,2}$, or $z_{i,1} \cup z_{i,2}$ to the corresponding adjacent neighbour (the peer holding $p.b - 1$ or $p.e + 1$). The transfer must : (1) reduce as much as possible the cumulative overload of the two peers involved, and (2) be as small as possible. These criteria maximize the reduction of the cumulative overload while entailing the least changes. Since peer p only has local information, it does not know which key interval the destination peer can receive. Asking the destination peer for its load information before transferring would slow

down the procedure. Furthermore, this may entail an incorrect decision since the status of the destination peer evolves continuously. Our solution allows peer p to propose a set of candidate key intervals to the neighbour. The transfer is completed when the destination peer chooses the most appropriate interval. Such transfer requires only one ask-answer communication between the two peers. Let $w_{h,j}$ (for integers $0 \leq h < 2k$ and $0 \leq j \leq 1$) represent the candidate key intervals to transfer. We determine $w_{h,j}$ using the following rule :

$$w_{h,j} = \begin{cases} z_{k-h-1,j} & \text{if } 0 \leq h < k \\ z_{h-k,j} \cup z_{h-k,2} & \text{if } k \leq h < 2k \end{cases}$$

Thus, the routing traffic load on $w_{h,j}$, denoted $T(w_{h,j})$, is given as :

$$T(w_{h,j}) = \begin{cases} \dfrac{G_p[k-h-1,j]}{t_c - t_0} & \text{if } 0 \leq h < k \\ \dfrac{G_p[h-k,j] + G_p[h-k,2]}{t_c - t_0} & \text{if } k \leq h < 2k \end{cases}$$

Index load balancing algorithm : The index load balancing algorithm (applying the key interval transfer protocol below) on peer p is as follows :

Let $n_0(p)$ denote the adjacent neighbour of p that holds $p.b - 1$ and $n_1(p)$ denote the adjacent neighbour of p that holds $p.e + 1$.

1. select the smallest h such that $\exists j \in \{0,1\}$ and $T_p - T(w_{h,j}) \leq C_p$. Then, execute the key interval transfer protocol for $w_{h,j}$ from p to $n_j(p)$. If the transfer succeeds, the load balancing stops. Otherwise, continue with step 2 ;
2. set $l = (j+1) \bmod 2$. Select the smallest h such that $T_p - T(w_{h,l}) \leq C_p$. Then, execute the key interval transfer protocol for $w_{h,l}$ from p to $n_l(p)$. After this step, the load balancing stops even if the key interval transfer does not succeed.

Key interval transfer protocol : The transfer protocol for the key interval $w_{h,j}$ from peer p to peer $n_j(p)$ tries to move one of the key intervals $w_{0,j}$, $w_{1,j}$,..., $w_{h,j}$ from p to $n_j(p)$ such that the combined overload of p and $n_j(p)$ is minimized. Formally, the overload of p is $O_p = (T_p - C_p + |T_p - C_p|)/2$ and that of $n_j(p)$ is $O_{n_j(p)} = (T_{n_j(p)} - C_{n_j(p)} + |T_{n_j(p)} - C_{n_j(p)}|)/2$. Thus, the transfer must reduce as much as possible $O_p + O_{n_j(p)}$. The key interval transfert protocol involves the following steps :

1. p sends to $n_j(p)$ a key interval transfer proposal including the list ($w_{0,j}$, $w_{1,j}$,..., $w_{h,j}$), the list ($T(w_{0,j})$, $T(w_{1,j})$,..., $T(w_{h,j})$), and O_p ;
2. if $n_j(p)$ is not able to receive a key interval or $T_{n_j(p)} \geq C_{n_j(p)}$, it refuses the transfer. Otherwise,
   (a) it searches for the greatest $g \in [0, h]$ such that $T_{n_j(p)} + T(w_{g,j}) \leq C_{n_j(p)}$;
   (b) if no such g exists, $n_j(p)$ searches for the smallest $g \in [0, h]$ satisfying

$$|T(w_{g,j}) - O_p| + T(w_{g,j}) - O_p + 2(T_{n_j(p)} - C_{n_j(p)}) < 0 \qquad (5.1)$$

      i. if no such g is found, $n_j(p)$ refuses the transfer because $O_p + O_{n_j(p)}$ cannot decrease ;
      ii. if such a g is found, $n_j(p)$ sets the chosen index as g ;
3. if an index g is chosen (by step 2a or 2(b)ii), $n_j(p)$ changes its key interval by $[n_j(p).b, n_j(p).e] \cup w_{g,j}$ and establishes connections to the new neighbours. Then, it sends to p an acceptance message specifying the chosen index g ;
4. upon receiving the acceptance message with the chosen index g, peer p updates its key interval to $[p.b, p.e] \setminus w_{g,j}$ and releases the unnecessary connections to other peers. The transfer then succeeds ;
5. in case $n_j(p)$ refuses the proposal of p, the transfer fails.

Theorem 1 *Given $w_{h,j}$ the interval to be transferred from peer p to peer $n_j(p)$ using the key interval transfer protocol. If $n_j(p)$ chooses an index $g \in [0,h]$, then transferring $w_{g,j}$ will maximize the reduction of the combined overload of p and $n_j(p)$.*

Proof : Peer $n_j(p)$ chooses an index $g \in [0,h]$ in step 2a or 2(b)ii of the protocol to accept the transfer of $w_{g,j}$. Recall that at each peer on the routing path, the routing algorithm limits the choice of the next de Bruijn node t (to direct the message to) in the de Bruijn neighbourhood set of the current peer's key interval. Therefore, if $w_{g,j}$ moves from p to $n_j(p)$, $T(w_{g,j})$ is transferred from p to $n_j(p)$ with high probability[5]. The overloads of p and $n_j(p)$ after the transfer are estimated as :

$$\begin{aligned} O'_p &= (T_p - T(w_{g,j}) - C_p + |T_p - T(w_{g,j}) - C_p|)/2 \\ O'_{n_j(p)} &= (T_{n_j(p)} + T(w_{g,j}) - C_{n_j(p)} + |T_{n_j(p)} + T(w_{g,j}) - C_{n_j(p)}|)/2 \end{aligned}$$

The condition for reducing the total overload of p and $n_j(p)$ is :

$$\Delta O = O'_p + O'_{n_j(p)} - O_p - O_{n_j(p)} < 0 \tag{5.2}$$

If g is set by step 2a, $T_{n_j(p)} + T(w_{g,j}) \leq C_{n_j(p)}$. Thus, $O'_{n_j(p)} = 0$. Since $O'_p < O_p$ and $O_{n_j(p)} = 0$, (5.2) holds.

If g is set by step 2(b)ii, (5.1) holds and $O'_{n_j(p)} = T_{n_j(p)} + T(w_{g,j}) - C_{n_j(p)}$. It is easy to prove that the left hand side of (5.1) is equal to $2\Delta O$ and that the smallest chosen index g induces the largest $|\Delta O|$. □

In the key interval transfer protocol, step 2b is mandatory. Study the case where p is overloaded, $n_j(p)$ is underloaded, and there exists no $g \in [0,h]$ such that $T_{n_j(p)} + T(w_{g,j}) \leq$

[5]Because of the de Bruijn graph structure, it cannot be guaranteed that all trafic "transferred" will effectively be transferred.

$C_{n_j(p)}$. Without step 2b, p cannot transfer any key interval to $n_j(p)$. Since $n_j(p)$ is underloaded, it does not intend to take off any part of its key interval. This situation blocks the transfer of load from p. The presence of step 2b allows peer p, in this case, to transfer the least loaded zone $w_{g,j}$ when it reduces the combined overload of p and $n_j(p)$. The load transfer thereby continues until some steady state.

5.4 Storage load balancing

We define the storage load of a peer as the total of size of the objects it stores. Each peer has a limited capacity available for storage which might be used for object migration. The system's management to store objects requires network bandwidth for object distribution, re-distribution, and associated index management (i.e., routing requests to the network). Consequently, the storage load balancing method has three goals : (1) keeping the storage load under the storage capacity on every peer, (2) adjusting bandwith consumption requirements to bandwidth availability, and (3) minimizing its impact on index load balancing (Sect. 5.3). To achieve these three goals, we propose to separate the location of the key of an object from the location of the object itself. In this way, objects can reside on arbitrary peers. Therefore, roots are only required to keep pointers to the objects under their responsibility. This approach simplifies the mechanisms required to achieve the first two goals. Finally, the independence of object and key locations enables us to achieve the third goal. Indeed, the key interval transfer remains efficient since only object pointers (very small in size) are required to move when a key interval transfer occurs.

A consequence of this approach is that replication of objects is simplified, hence enhancing object availability, without the need for multiple mapping hash functions (such as e.g. in [BCM03]) or for maintaining invariants that constrain replication to nearby peers (e.g., [RD01b]). A root simply needs to keep pointers to the peers that store the replicas of an ob-

ject. In this paper, we consider that up to d ($d \geq 1$) replicas of an object may be stored. When a peer departs the network, it must guarantee the objects' availability. By allowing replication, we facilitate this task, since the departing peer only has to wait for objects with unique replicas to be copied elsewhere.

5.4.1 Object pointer and object insertion

Every peer maintains two tables : *indices* and *storage*. Each entry of table *indices* contains the index of an object under the peer's responsibility. The index includes the object identifier (*oid*), and a list of pointers to the replicas (*replicas*). A replica pointer consists in the replica identifier (*rid* - a number in $[0, d-1]$), the storing peer address (*location*), and the replica's storage counter (*counter*). This counter is initially set to 0 and incremented after each change of location. Its use will be explained below. The *storage* table contains the list of objects stored on the peer. For each object, it records the object identifier (*oid*), the replica identifier (*rid*), the size (*size*), the address of the root (*root*), and the storage counter (*counter*). In order to maintain *indices* and *storage*, we propose two protocols, namely the storage notification protocol and the root notification protocol.

Whenever a peer receives an object, it sends to the root of the object a storage notification which contains its address and (*oid*, *rid*, *counter*) of the object. The *counter* field lets the root know whether the notification is newer than the corresponding pointer it holds. If the notification is new, the root updates the pointer. In the notification, the sending peer attaches the *root* field of the object header, asking whether it keeps the correct root address or not. If the information is incorrect, the root sends back a root notification.

When a peer receives a key interval, it sends root notifications to the storing peer of the objects involved. A root notification contains the root address, (*oid*, *rid*, *counter*) of the object, and the storing peer's address (*location*), as known by the root. On receiving the

root notification, the storing peer updates the corresponding object's header. If *counter* or *location* are incorrect, the storing peer sends a storage notification back.

The maintenance of pointer consistency may seem complicated. However, in comparison to traditional systems which associate storage location and key, the key interval transfer used in our structure requires little effort. It involves the move of a number of pointers and some notifications but does not require any object transfer. The size of an object pointer and of a notification is much smaller than the size of the object.

The object insertion algorithm must ensure that $S_p \leq D_p$ for every peer p, where S_p and D_p are the storage load and capacity of p, respectively. In addition, it tries to store the object on up to d different peers. An insertion request contains the object identifier (*oid*) and size (*size*). The request is routed to the root of the object. If an index for the object already exists, the insertion algorithm stops. Otherwise, it starts diffusing replicas, with *rid* from 0 to $d-1$. The diffusion process tries the root first.

A replica diffusion message λ_r contains *oid*, *size*, and *ridlist*, where *ridlist* is the list of *rid*s remaining to be assigned. The message traverses multiple peers. A *ttl* (time-to-live) field limits the number of peers visited. At each peer q, if $S_q + size \leq D_q$ and q does not store any replica of the same object, q extracts a *rid* from *ridlist*, loads the corresponding replica to the local storage, and sends a storage notification to the root. If *ridlist* is not empty and $ttl > 0$, q decrements *ttl* and forwards λ_r to a neighbour not visited. Message λ_r maintains the list of visited peers to perform this verification. If *ttl* reaches 0 but no replica was stored, the insertion fails. If the number of stored replicas is between 1 and $d-1$, the root starts a new diffusion for the remaining *rid*s.

Object deletion is not considered here since it does not increase the storage load.

5.4.2 Storage load balancing algorithm

Recall that the first two goals of the storage load balancing are to avoid storage overload and to take into account the bandwidth required respectively available to do so. Implicitely, the storage capacity of a peer D_p corresponds to the real storage available for objects. However, for the system to work properly, another boundary must be defined, which we refer to as the desired capacity on a peer $\overline{D}_p$, with $\overline{D}_p < D_p$. When inserting objects into the system, we ensure that $S_p \leq D_p$, hence allowing S_p to temporarily exceed $\overline{D}_p$ but always limiting it to D_p. Consequently, the storage load balancing problem can be specified as the minimization of the storage overload with regards to $\overline{D}_p$ while keeping $S_p \leq D_p$.

Given $A_p = \overline{D}_p - S_p$ the available space on peer p, a peer is overloaded when the overload $O_p = (-A_p + |A_p|)/2$ is positive, otherwise $O_p = 0$. The storage load balancing algorithm aims at minimizing the overload of all system components. It consists in the decentralized exchange of objects between pairs of peers. Suppose that an overloaded peer p exchanges objects with a peer q. In general, p sends to q a set of objects R_{pq} and q sends back to p a set of objects R_{qp}. Given that S_{pq} and S_{qp} are the storage loads of R_{pq} and R_{qp}, respectively, the combined overload of p and q decreases only if $A_q > 0$ and $0 \leq S_{qp} < S_{pq}$.

Definition 4 *Given that peer q receives a storage load S_{pq} from a peer p and selects a storage load S_{qp} to send back to p, the optimal exchange must (1) reduce the combined overload of p and q the most, and (2) minimize S_{qp}.*

Condition (1) guarantees the fastest reduction of the combined overload, while condition (2) minimizes the data volume sent. Hence, this approach not only reduces the storage overload, but also the bandwidth required to perform storage load balancing.

Theorem 2 *Given two peers p, q, with $A_p < 0$ and $A_q > 0$, and S_{pq}, the optimal exchange*

occurs when

$$S_{qp} = \begin{cases} 0 & \text{if } S_{pq} \leq A_q \text{ or } A_q < S_{pq} \leq -A_p \\ \textit{closest to} \quad \min(A_p, -A_q) + S_{pq} & \\ \quad \textit{such that } 0 \leq S_{qp} < S_{pq} & \text{if } S_{pq} > \max(-A_p, A_q) \\ \quad \textit{and } S_{qp} > A_p - A_q + S_{pq} & \end{cases}$$

Because of the limitation of space, we do not present the proof of this theorem. However, it can be found in the full version of this paper.

Storage load balancing algorithm : Each peer p periodically verifies the storage load. If p is overloaded, it starts a balancing session :

1. p diffuses an available space interrogation ϕ, with a limited *ttl* (time-to-live) field, to its neighbourhood. Each peer q that receives ϕ the first time, processes ϕ, decrements *ttl*, and forwards ϕ to its neighbours excluding p and the peer from which ϕ comes. q responds to ϕ by sending $A_q = \overline{D}_q - S_q$ to p if $A_q > 0$;
2. for each reply A_q received, if p is still overloaded, p and q exchange objects such that the combined overload of p and q will decrease the most while the object migration is minimized :
   (a) p selects a set of objects R_{pq} to send to q satisfying one of the following conditions : (1) R_{pq} is the smallest that can underload p without overloading q; (2) if (1) cannot be satisfied, R_{pq} is the largest that cannot overload q; and (3) if both (1) and (2) cannot be satisfied, R_{pq} contains only the smallest object;
   (b) q selects a set of objects R_{qp} to send back to p. The selection is based on the optimal exchange condition stated in Theorem 2.

5.5 Conclusion

We have introduced balancing methods for index load and storage load that can simultaneously operate. The index load balancing is based on the exchange of key intervals among the peers. Unlike the *Expressways* [ZSZ02] method, which must collect the load information of all peers before redistributing load, our method relies only on local information. We thus avoid the overhead of the load information communication.

The storage load balancing method manipulates the system structure at the object level, instead of the key level (such as the *Virtual servers* [RLS+03] method). The manipulation at the key level exhibits less flexibility since the all objects belonging to one key must move together with the key. Moreover, a move of keys in balancing the storage load also affects index load.

The load balancing methods presented operate on the overload instead of the load itself. Most other methods aim to adjust the load or the load/capacity ratio of every peer with a global objective function. This requires to globally calculate the targeted optimization and to continuously reorganize the system. By relying on the local examination of the overload, we need to react only when the overload exists and when it can be reduced. Experiments to evaluate the proposed load balancing methods are currently being conducted. So far, preliminary results have confirmed their anticipated efficiency. These experimentation results will be presented and discussed elsewhere.

Chapitre 6

Article : BALLS simulator : evaluator of a structured peer-to-peer system with integrated load balancing [LBK06b]

Afin d'évaluer les méthodes introduites par l'article précédent, nous effectuons des simulations. Un simulateur de BALLS a été développé parallèlement au développement de la solution théorique. L'interface et les spécifications ont été définies en collaboration par Viet Dung Le et les professeurs Gilbert Babin et Peter Kropf. La conception et la programmation ont été réalisées par Viet Dung Le. Le présent article[1] décrit et valide le simulateur de BALLS. Il a été publié et présenté lors de la 4^{e} conférence internationale en informatique IEEE *Research, Innovation and Vision for the Future* (12-16 février 2006, à Ho Chi Minh Ville, Vietnam).

[1]L'utilisation de l'article dans la thèse est autorisée par les coauteurs.

Abstract

Simulation is an efficient way to evaluate new peer-to-peer models. It requires two implicit properties : large scale and high dynamicity. In the context of our work that proposes a peer-to-peer structure based on partitioning a de Bruijn graph and its load balancing algorithms, we developed a simulator for evaluation purposes. This paper introduces a three-layer architecture of the simulator. This architecture allows to support simulations in two modes : centralized (where all peers are simulated on one physical machine) and decentralized (where the peers run on separate machines communicating through the underlying network).

6.1 Introduction

The recent appearance of structured peer-to-peer (P2P) systems (e.g., [KK03, MNR02, NW03a, RFH+01, RD01a, SMK+01, ZKJ01]) proposed interesting solutions to the routing problem resulting from the lack of a centralized control. Such a system manages objects by distributing the responsibility of the object keys (usually produced by hashing the object id) over the available peers. The peers connect according to the set of keys each one holds.

The distribution of the key responsibility in a structured P2P system introduces a problem of load unbalance, which critically affects the system's performance. In this context of load balancing in a structured P2P system, we have proposed a P2P structure and algorithms for the index management load balancing and the storage load balancing. The index management load represents the rate of bandwidth consumption on each peer for routing, whereas the storage load implies the usage of each peer's resources for object accommodation. The proposed structure (namely BALLS - Balanced Load Supported P2P Structure) partitions a de Bruijn graph layered over the P2P network. It achieves simultaneous index management

load balancing and storage load balancing. Details of this system are found in [LBK06c].

The performance of our proposed P2P system must be evaluated. A P2P system usually displays two properties : large scale and high dynamicity. Consequently, using simulators is an efficient method for evaluating P2P systems.

There exist P2P simulators. Peersim [pee05] supports simulation of several P2P protocols as part of the BISON (Biology-Inspired techniques for Self-Organization in dynamic Networks) project. The simulator manages an array of peers and performs the peer operations sequentially. It provides the following main Java classes and interfaces : Node, CDProtocol, Linkable, Observer, and Dynamics to respectively model the peer, protocol, network, data collection, and dynamicity.

Another P2P simulator is the NeuroGrid simulator [Jos03], which was originally designed to compare the Freenet, Gnutella, and NeuroGrid [Jos02] systems. This simulator supports abstract classes intended to be generic for different P2P simulations. The classes Network, Node, Document, and Keyword represent, respectively, the P2P network, peer, document, and keyword. Nodes interact through instances of class Message. The interface MessageHandler provides actions to process Messages on each peer.

The p-sim simulator [MSZ03] was developed in C. It uses the event-driven simulation technique. The supported events (including Peer Arrival/Departure, Search Query, and Evaluation) have a time-stamp. The simulation executes the events sequentially according to the order of their time-stamp. The simulation components include topology, peer dynamics, file search protocol, and evaluation metrics.

He et al. [HAR+03] introduced a framework for P2P simulation that takes into account the details of the underlying network. Each peer simulated by this framework consists of three layers : Network Simulator (providing packet transfer service), PeerAgent (supporting message forwarding and processing), and PeerApp (performing application actions such as

query processing and peer relationship maintenance).

FreePastry [fre05] is a Java implementation of the Pastry network for deployment in the Internet. The latest version supports two transport communication modes : Direct and Socket. The Direct mode emulates the whole system with Pastry nodes in one Java VM without a physical network. The Socket mode, however, uses TCP for communication (except liveness checks) between Pastry nodes.

The ongoing effort to provide a diversified evaluation of our proposed P2P structure and load balancing algorithms requires centralized simulations (in one physical machine) and decentralized simulations (in a real network). Centralized simulations evaluate the system with a high number (thousands) of peers and high dynamicity. Decentralized simulations, on the other hand, take into account the real-time and parallel factors. The above simulators (except FreePastry) are generic for evaluating different P2P systems but do not support decentralized simulations. FreePastry can answer this requirement due to its two communication modes. It also includes a common API package [DZD+03] for developing different structured P2P overlays. However, the API is suitable for P2P structures having peer ids belonging to the object key space (e.g., [RFH+01, RD01a, SMK+01]). Our P2P structure separates the peer id from the object key to enable index management load balancing. Therefore, basing our simulator on this API is not advantageous.

We therefore implemented the BALLS simulator for our proposed P2P system that can perform simulations under both modes, centralized and decentralized. The present paper describes the architecture of the BALLS simulator, which includes three layers : network layer, peer layer, and evaluation layer.

The three-layer architecture also facilitates the extension of the BALLS simulator for other P2P structures and protocols. This can be done by simply extending and developing classes in the peer layer without affecting the other layers.

The rest of this paper is organized as follows. Section 6.2 outlines the objectives and architecture of the BALLS simulator. Section 6.3 summarizes the BALLS structure and algorithms as the simulated objects. Section 6.4 goes into details of the simulator layers. Discussion in Section 6.5 validates the simulator. Finally, the last section concludes the paper.

6.2 Generic architecture of the BALLS simulator

Evaluating the BALLS structure and the load balancing methods requires a large-scale and dynamic environment, which involves a large number (e.g., thousands) of peers that continuously join and leave. Running such a system in a real network, where each peer occupies a physical node, is very expensive and often only possible in closed experimental environments. As an alternative, we use centralized simulations in which all peers operate on the same machine. In such a way, evaluation of a large and dynamic system becomes feasible and efficient. Moreover, a centralized simulation greatly simplifies the global measurements. Unlike centralized simulations, a decentralized simulation implements the peers on separate machines communicating through the underlying network. Evaluation employing decentralized simulations allows us to take into account the effect of real network and real-time factors. Therefore, one objective of the BALLS simulator is to support both centralized and decentralized simulation modes.

The decentralized simulations supported by the BALLS simulator are not real experiments yet, although they run on a real network. The objects and routing requests in simulations are generated by scenarios selected a priori but not by real users. In a real experiment, the peer properties (e.g., storage capacity, traffic capacity) are set based on the host machine configuration. However, in simulations, they are chosen according to simulation settings. Although being a simulator, this framework is a first step towards the future implementation of the BALLS system.

The evaluation of a P2P system usually considers the routing cost and the maintenance cost. The routing cost means the number of hops involved in routing. Whereas, the maintenance cost denotes the number of messages exchanged to maintain the P2P structure when a peer arrives or departs. The maintenance cost is usually related to the degree of the peers. The BALLS simulator must be able to explore the routing cost, maintenance cost, and peer degree.

On the other hand, evaluating the proposed load balancing methods requires the simulator to provide functionalities that measure the loads (index management load and storage load) and compute the corresponding overloads (see Sect. 6.3 for definitions). This observation allows us to quantify the load balancing performance.

In order to fulfill the above objectives, we designed the BALLS simulator in three layers : network, peer, and evaluation.

The **network layer** is responsible for system configuration and peer communication. These tasks function differently across the two simulation modes. In centralized simulations, the whole system with multiple peers can be initiated at the same time on one physical machine. The peers communicate by direct access to each other. However, in decentralized simulations, each peer is initiated on a separate machine. The peers therefore communicate through network services provided by the underlying network. For this reason, we divide the network layer into two parts : centralized and decentralized. Each part of this layer supports the network functionalities for the corresponding simulation mode.

The **peer layer** provides the operational functions of a peer : joining, departing, routing, object managing, and load balancing. Since the functionality requirement of a peer does not change across the centralized and decentralized simulations, we design this layer commonly for the two simulation modes.

The **evaluation layer**'s task consists in conducting simulations and observing simula-

tion data. The two simulation modes require different evaluation strategies. In centralized simulations, global measurements can be efficiently computed by local functions. Whereas, they become more difficult in the decentralized mode where additional network communications are needed to perform aggregate operations. Consequently, this layer involves two parts : centralized and decentralized. The centralized part supports evaluation of the peer degree, routing cost, arrival cost, departure cost, index management load balancing, and storage load balancing. To ensure the computational efficiency, the decentralized part of the current simulator version only evaluates the load balancing methods.

6.3 BALLS and load balancing algorithms

This section briefly introduces the BALLS structure and load balancing algorithms (see also [LBK06c]), the objects evaluated by the simulator. It presents the simulator's requirements and helps the reader understand concepts that are used in later sections. We define the **index management load** of a peer as the bandwidth consumed in a unit of time for the routing of messages. The **storage load** of a peer is the total size of the objects (files) it actually stores. To enable the simultaneous index management load balancing and storage load balancing, we proposed the BALLS structure that separates the concerns of peer id (address), object key, and storage location.

Due to its interesting properties (low node degree and diameter), the de Bruijn graph appears extensively in the literature of interconnection networks (e.g., [BDE97, CCRS87, adHV99]). We apply the de Bruijn graph in the P2P structure for the goals : low-cost maintenance, efficient routing and load balancing.

The BALLS partitions a binary de Bruijn graph in the P2P network. The key space is identical to the de Bruijn node id space. Each peer p holds (is responsible for) a non-empty

key interval, denoted $[p.b, p.e]$. Any two peers p and q connect if $[p.b, p.e]$ and $[q.b, q.e]$ are connected by a de Bruijn arc or are adjacent in the circular key space. Such p and q are called neighbours. A peer maintains a *neighbour list* consisting of a triple $(q.a, q.b, q.e)$ for each neighbour q, where $q.a$ is the network address (e.g., IP) of q.

The routing follows appropriate routing paths in the de Bruijn graph. Because of the de Bruijn graph's low diameter, the routing efficiency is guaranteed. The BALLS applies simple peer arrival and departure protocols. When a new peer p joins, it lookups for the peer responsible for a random key via a known peer. The peer found splits its key interval and transfers one half to p. When a peer q departs, it selects among its two ring neighbours (i.e., peers holding a numerically adjacent key interval) the one holding the shortest key interval to transfer $[q.b, q.e]$. If the transfer is done, q notifies all its neighbours about the departure and quits after receiving their confirmation.

The index management load balancing aims at minimizing the system's global overload. Given a peer p with index management capacity C_p and index management load T_p, its overload is $O_p = (T_p - C_p + |T_p - C_p|)/2$. Whenever $O_p > 0$, p transfers some portion of $[p.b, p.e]$ to an appropriate ring neighbour (say q) so as to reduce the combined overload $O_p + O_q$ the most. The separation between peer id and object key means that p can dynamically change either $p.b$ or $p.e$. It thus enables the key transfer among peers. The reduction of $O_p + O_q$ contributes to the minimization of the global overload.

Each peer p has a space boundary D_p to limit the storage load S_p. In addition to the storage space, object accommodation also uses network bandwidth for object access and migration. For a proper operation, we define another boundary namely the desired storage capacity $\overline{D}_p$, with $\overline{D}_p \leq D_p$. S_p never exceeds D_p but can temporarily exceed $\overline{D}_p$. The storage overload is defined as $W_p = (S_p - \overline{D}_p + |S_p - \overline{D}_p|)/2$. The storage load balancing aims at minimizing the global storage overload while ensuring $S_p \leq D_p$ on every peer p.

Our approach separates the object key and object location. An object can reside on a peer other than its root (the peer responsible for the object's key). To keep reference, an object and its root maintain pointers to each other. This separation allows objects to migrate among peers. As long as $W_p > 0$, the storage load balancing algorithm on peer p exchanges appropriate objects with another peer (say q) to minimize their combined overload $W_p + W_q$. Obviously, this exchange contributes to the global overload minimization.

6.4 The layers of the BALLS simulator

Section 6.2 gave an overview of the BALLS simulator's architecture, which comprises three layers : network, peer, and evaluation. The present section describes the layers in more details. The implementation of this architecture uses Java. Figure 6.1 depicts the principal classes supported as well as their relationships. In centralized simulations, the peer layer operates with the centralized parts of the network and evaluation layers. In decentralized simulations, the peer layer and the decentralized parts of the other layers are used.

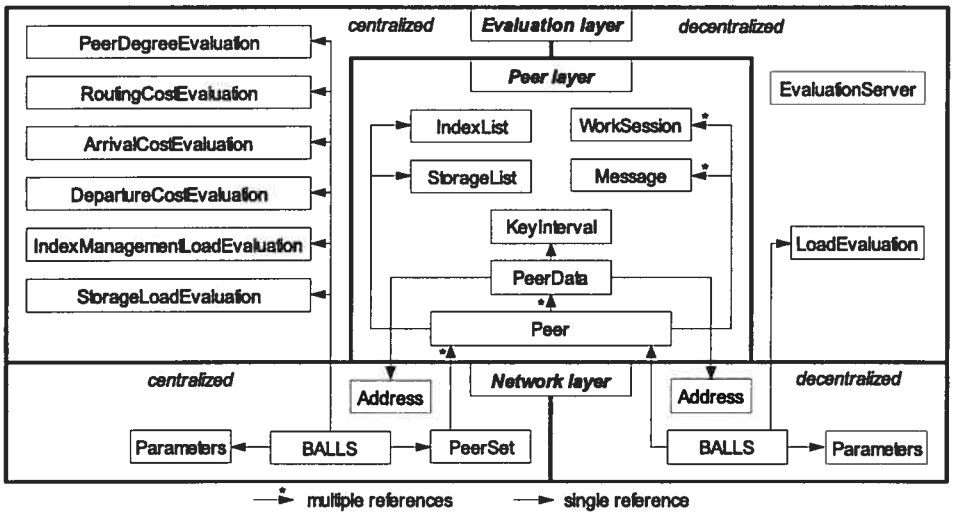

FIG. 6.1 – The simulator architecture

6.4.1 Network layer

This layer is decomposed into two parts : centralized and decentralized. While the centralized network layer constructs the whole system on one machine and realizes communication using direct access, the decentralized network layer initiates only one peer at a time and supports communication via the real network.

The centralized network layer supports the following classes : BALLS, Parameters, PeerSet, and Address. Class BALLS represents the P2P system. It keeps an instance of Parameters (for the configuration) and an instance of PeerSet (for the access to the peers).

The Parameters class is responsible for reading/setting the parameters from/to a configuration file and keeping the parameter values to conduct the system operation. Principal examples of parameters are : the key length, initial number of peers, arrival and departure rates, and simulation scenarios.

The PeerSet class is an array of Peers (representing peers) sorted in ascending order of addresses. This class provides the peer insertion/deletion (in the array) and access functions using binary search. The peers run in a circular schedule. The peers' execution order follows their order in the array.

A complete execution round of all peers constitutes a simulation cycle. In the centralized mode, the simulator does not rely on the computer clock to calculate the time because the load of concurrent processes affects the operation speed. However, we calculate the time as a real number in which the integer part is the current simulation cycle counted from the start of simulation. The fraction part is equal to the fraction of the currently executing peer's index in the current size of the peer array.

The Address class represents the peer address, which is an integer in the centralized mode. We use it as the key for sorting the peers in the array and for binary search. Sending

a message to a peer involves locating it in the array using the given address and inserting the message to its message queue.

The decentralized network layer supports classes : BALLS, Parameters, and Address. The BALLS class here represents the system aspect of one peer. The Parameters class provides similar functionality to that of the centralized part. However, since it configures one peer, the address (TCP/IP) of the peer must be specified. Moreover, it allows to set the address of the EvaluationServer (which we will introduce in Sect. 6.4.3).

The Address class now specifies the TCP/IP address of the corresponding peer. In communication, the BALLS objects of the peers exchange messages using TCP/IP and push them to the message queue of the receiving peers.

6.4.2 Peer layer

The peer layer implements the classes supported for a peer's functionality. The classes are common for both simulation modes. The principal classes includes : Peer, PeerData, KeyInterval, IndexList, StorageList, Message, and WorkSession.

The role of class Peer is to perform the peer activities. Each execution cycle of a peer consists in treating the new messages, performing the ongoing protocols (WorkSessions), and generating random actions (e.g., start of routing, insertion of new objects) depending on the simulation requirement.

Class PeerData represents the triple $(p.a, p.b, p.e)$ of a peer p. Peer p uses a PeerData instance to identify its own $(p.a, p.b, p.e)$ and a set of PeerData instances for the neighbour list. Class KeyInterval represents a key interval. It also provides the necessary calculations in the circular de Bruijn node space, e.g., union, intersection, neighbourhood verification, distance.

In order to manage the objects, a peer maintains an instance of class IndexList and an instance of class StorageList. An IndexList is the list of the pointers to the objects under the current peer's responsibility. It sorts the items in ascending order of (object key, object id). The StorageList is the list of the objects stored on the current peer. It sorts the items based on the object id. The lists are sorted to support binary search.

We develop the Message class for peer communication. The current BALLS simulator version supports 24 extensions of Message to use in different protocols. The principal message types includes : routing message, joining message, departure message, interval notification, interval transfer message, storage transfer message, storage notification, and root notification. In the decentralized simulation mode, the peer periodically sends a message called load notification to the EvaluationServer (see Sect. 6.4.3) for aggregate measurements.

Class WorkSession implements a protocol on a peer. The present simulator version develops 13 extensions of WorkSession. Principal work sessions includes : routing, joining, departure, interval transfer, interval notification, index management load monitoring, index management load balancing, storage load monitoring, and storage load transfer. Each peer keeps instances of the ongoing work sessions. An execution cycle resumes the sessions by calling their continueSession method. When a session ends, the peer frees its instance.

6.4.3 Evaluation layer

The evaluation layer supports measurement of different metrics so as to evaluate the system performance. As explained before, this layer is divided into two parts : centralized and decentralized to perform different evaluation strategies.

The centralized evaluation layer provides the evaluation classes : PeerDegreeEvaluation, RoutingCostEvaluation, ArrivalCostEvaluation, DepartureCostEvaluation, IndexManagementLoadEvaluation, and StorageLoadEvaluation. The first four classes measure the topo-

logy performance. The last two, however, evaluate the load balancing methods. An evaluation instance starts with the simulation, calculates the desired metrics at intended moments, and saves the metric values for later analyses.

PeerDegreeEvaluation measures the average and distribution of peer degree in function of system size. A measurement takes place when an arrival or departure of peer finishes successfully. RoutingCostEvaluation measures the average routing cost. It registers the number of hops taken by each routing when the routing finishes. ArrivalCostEvaluation and DepartureCostEvaluation measure the number of messages sent for peer arrival and departure in function of system size. The number is counted during the joining and departure sessions.

IndexManagementLoadEvaluation and StorageLoadEvaluation evaluate the load balancing methods. They measure the corresponding global overloads, loads, and capacities along the simulation life. To enhance the evaluation quality, the simulator allows to set simulation scenarios. A scenario defines the moments to start or stop load balancing functions, the capacities' utilization, the key popularity dynamicity, etc.

This layer also supports different patterns of load and capacity distribution so as to achieve the most realistic simulations. As suggested by numerous research (e.g., [Dow01a, GFJ+03, LRS02, SGD+02]), the Zipf distribution is appropriate for the peer's capacities and the routing skewness while the log-normal distribution is appropriate for the object size. Table 6.I shows the distribution patterns supported by the BALLS simulator.

Distribution	Random	Zipf	Log-normal
Index management capacity	√	√	
Routing source probability	√	√	
Routing target probability	√	√	
Desired storage capacity	√	√	
Object size	√		√

TAB. 6.I – Supported distribution patterns

The decentralized evaluation layer of the current BALLS simulator version supports

the index management load balancing and storage load balancing evaluations. In order to facilitate aggregating measurements, we use a process called EvaluationServer. The EvaluationServer runs as a server that aggregates measurement values from the peers, periodically calculates the desired global measurements (including the overload, load, and capacity), and writes the results to files.

In the peer part, we group both supported evaluations in one class called LoadEvaluation. It periodically sends a load notification (Sect. 6.4.2) containing the current index management load, storage load, and the peer's capacities to the EvaluationServer. With the limitation of current decentralized testbeds (e.g., no more than 594 nodes in PlanetLab) and the small size of load notifications, this client/server evaluation cannot cause a computational catastrophe.

6.5 Discussion

The previous section has described the design of the BALLS simulator. We now review whether the functionality of the simulator conforms to what we expect. We validate the simulator in the two simulation modes.

6.5.1 Validating the centralized simulations

The centralized simulations are required to observe the peer degree metrics (average and distribution), the arrival cost, the departure cost, the routing cost at different system sizes, as well as the global index management and storage overloads, loads, and capacities along the simulation life.

The measurement of the peer degree is based on the peers' neighbour list size since it reflects the number of connections among the peers. We only count the degree of the peers that actually join the system. The measurement takes place after a successfull peer arrival or

departure - the moment of a change in number of peers. It thus ensures the valid observation of the peer degree relative to the varying system size.

The evaluations of peer arrival and departure determine the number of messages sent (to maintain the P2P structure) when these events occur. Because the arrival and departure involve transferring some key interval from a peer to another, the cost is the number of notification messages sent by the giving peer (say q) and the receiving peer (say p). If the transfer succeeds, p replies an acceptance message to q. It also sends the number of its messages to q via this reply. q registers the total number of messages as the cost. In the case of departure, q also counts the number of confirmation messages from its neighbours before leaving. Each completed arrival or departure makes a change in system size. It ensures the valid arrival/departure cost corresponding to all system sizes in a simulation.

Each routing message maintains a list of the peers it passed. When routing finishes, the length of this list reflects the routing cost. The routing cost evaluation registers this value at the end of every routing. If the routing frequency is higher than the frequency of peer arrivals and departures, we can observe the routing costs relative to all the system sizes.

The index management load balancing evaluation and the storage load balancing evaluation measure and register the corresponding overload, load, and capacity of the whole system at every simulation cycle. This ensures observing the effect of the balancing methods at all simulation moments.

Centralized simulations usually require a huge amount of computer resources to accommodate the peers and to execute their operations. We assessed the scalability of the simulator using two measures : simulation size and simulation time. The simulation size includes the population of peers and the population of objects managed by the peers. The simulation time denotes the time (in milliseconds) needed to run a certain number of simulation cycles. We measured these metrics on a computer with a Pentium 4 CPU, 3.4 GHz, 2GB RAM, running

OS Red Hat Fedora Core 3 and a computer with the same hardware configuration but running Windows XP.

For measuring the simulation size, we executed experiments with different numbers of peers and different numbers of objects. Figure 6.2 shows the numbers of objects a simulation can manage corresponding to system sizes : 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}, and 2^{14} peers. The number of peers satisfies the requirement of large simulations (thousands of peers). It is comparable to the system sizes of numerous experiments on P2P, such as 4096 in [RLS+03], 2250 in [RD01b], or 2048 in [MSZ03]. On the other hand, the object population can attain hundreds of thousands. It is comparable to the numbers of objects used in numerous P2P experiments, e.g., 40000 in [GDS+03], 3000 in [THS05], or 30 per peer in [YJF05]. Denoting the peer population as x and the object population as y, we find that they have a linear relationship : $y = -29.22x + 672222$. This is due to the limited memory size, which is split between the peers and the objects. In all experiments, Windows XP yields a little higher object population.

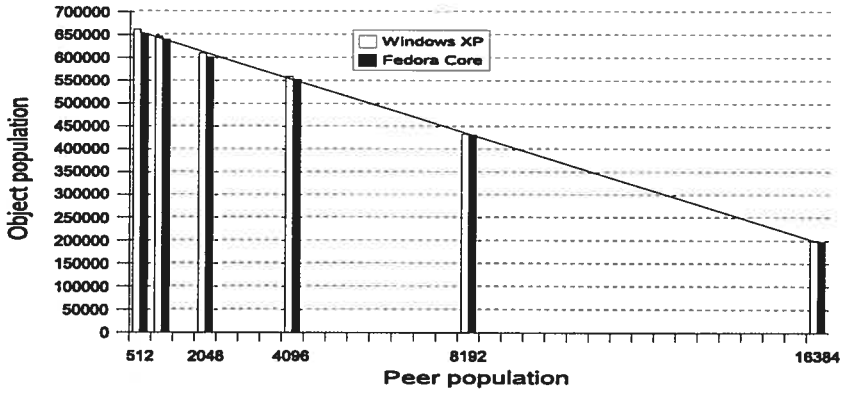

FIG. 6.2 – Simulation size evaluation

To evaluate the simulation time, we executed experiments that measured the time of 30 simulation cycles running in system sizes : 2^9, 2^{10}, 2^{11}, 2^{12}, and 2^{13} peers. In all experiments, the load balancing and the evaluation activities were activated. From the results in Figure 6.3,

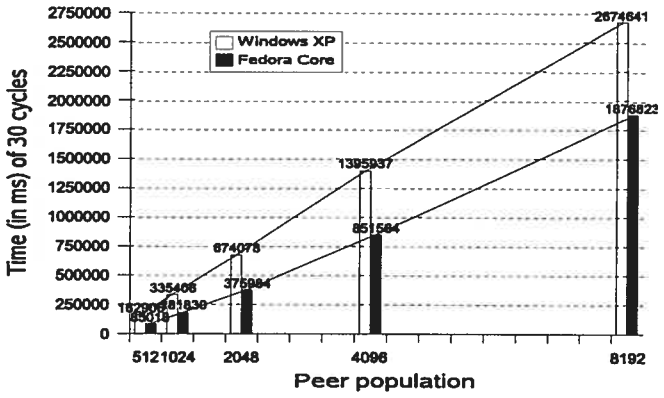

FIG. 6.3 – Simulation time evaluation

we can see a linear relationship between the time and the peer population. Based on the data collected, we calculated the angular coefficient of this relationship as 333.33 in Windows XP and 219.66 in Fedora Core. The execution time of Windows XP is longer than that of Fedora Core. However, even with 2^{13} peers, 30 cycles take only 2674641 ms. This allows a 300-cycle simulation to be completed in no more than 8 hours.

6.5.2 Validating the decentralized simulations

In the decentralized simulation mode, evaluation of the load balancing methods is also required to observe the corresponding global overload, load, and capacity during all the simulation life. However, global measurements cannot be locally calculated. As described in Section 6.4.3, an EvaluationServer aggregates load notifications from the peers and periodically registers the measured values to files. The EvaluationServer maintains a table keeping the load and the capacity of the currently live peers. Each reception of a load notification brings about an update of the corresponding entry in the table. Since the peers periodically post load notifications, if the EvaluationServer does not receive a peer's notification for some time, the corresponding entry is deleted. Peer arrivals and departures also entail updates in the table. The EvaluationServer produces the global measurement (of overload, load, and

capacity) relying on the current state of this table.

This measurement method allows to observe the load balancing metrics of the whole system along the simulation life. The more frequently the peers send load notifications, the more exact results we observe. Since the decentralized mode takes into account the real time factor but not the large-scale factor, our intended experiments will use a modest number of peers (no more than one hundred). This small simulation size will not cause any overload problem to the network.

6.6 Conclusion

Being developed in three independent layers, the BALLS simulator allows us to perform simulations in two environments : centralized and decentralized. The common use of the peer layer for both simulation modes reduces our effort in upgrading the simulator. Moreover, the simulator having this architecture is easy to be extended for other P2P systems. The network and evaluation layers are generic for most P2P structures. An extension therefore is made on the peer layer. We can also add more evaluations by developing new evaluation classes in the evaluation layer.

The evaluation classes provide the ability to explore most performance aspects of the P2P model : peer degree, routing cost, arrival and departure costs, and load balancing. The supported distribution patterns : Zipf and log-normal (for the peers' capacities, the routing source, the key popularity, and the object size) were confirmed by numerous research. Applying them enables us to imitate the most realistic distributions.

We have shown that the measurement methods supported by the simulator allow it to generate valid evaluation data. On the other hand, experiments have verified the simulator's capability in two aspects : simulation size and simulation time.

Chapitre 7

Article : BALLS : a structured peer-to-peer system with integrated load balancing [LBK06a]

Les articles précédents ont introduit la partie théorique de la solution et l'outil de simulation. Nous avons conduits plusieurs simulations avec cet outil pour évaluer la proposition théorique. Le plan expérimental a été établi par les professeurs Peter Kropf et Gilbert Babin en collaboration avec Viet Dung Le. La réalisation des simulations et l'analyse des données ont été faites par Viet Dung Le. L'article[1] suivant présente les résultats d'évaluation. Afin de donner une image complète des résultats de recherche, l'article reprend les méthodes de solution avec plus de détails. Puis, il apporte une description des simulations, des résultats d'évaluation et des analyses statistiques, le tout suivi d'une discussion. L'article a été soumis à un numéro spécial de la revue Annales des Télécommunications sur le thème des nouvelles technologies de la répartition le 15 janvier 2006 et a été accepté le 27 septembre 2006.

[1]L'utilisation de l'article dans la thèse est autorisée par les coauteurs.

Abstract

Load balancing is an important problem for structured peer-to-peer systems. We are particularly interested in the consumption of network bandwidth for routing traffic and in the usage of computer resources for object storage. In this paper, we investigate the possibility to simultaneously balance these two types of load. We present a structured peer-to-peer overlay that efficiently performs such simultaneous load balancing. The overlay is constructed by partitioning the nodes of a de Bruijn graph and by allocating the partitions to the peers. Peers balance network bandwidth consumption by repartitioning the nodes. Balancing of computer resources for storage is enabled by dissociating the actual storage location of an object from the location of its search key. The paper presents and analyzes the protocols required to maintain the overlay structure and perform load balancing. We demonstrate their efficiency by simulation. We also compare our proposed overlay network with other approaches.

Keywords—structured peer-to-peer systems, load balancing

Résumé

L'équilibrage de charge est un problème majeur des systèmes pair-à-pair structurés. Nous nous intéressons plus particulièrement à l'utilisation de la bande passante pour le routage de messages et à l'utilisation des ressources informatiques pour le stockage d'objets. Dans cet article, nous présentons un réseau pair-à-pair structuré permettant l'équilibrage efficace et simultané de ces deux types de

charge. Le réseau est construit en partitionnant les nœuds d'un graphe de De Bruijn et en assignant les partitions aux pairs. Les pairs équilibrent l'utilisation de la bande passante en repartitionnant les nœuds. On équilibre l'utilisation des ressources informatiques pour le stockage en dissociant l'emplacement d'un objet de celui de sa clé de recherche. L'article présente et analyse les protocoles requis pour maintenir la structure de réseau et pour équilibrer la charge. Nous démontrons leur efficacité par simulation. Nous comparons également notre réseau à d'autres approches.

Mots clés—systèmes pair-à-pair structurés, équilibrage de charge

7.1 Introduction

A peer-to-peer (P2P) system consists of multiple parties (peers) similar in functionality that can both request and provide services without a centralized control. This feature allows the system to spread the workload over the participants, hence aggregating their resources to more efficiently provide services or execute computational tasks. Roughly speaking, P2P networks can be classified as either structured or unstructured. Unstructured P2P networks (e.g., the Gnutella [gnu01] and KaZaA [kaz05] file sharing systems) have no particular control over the file placement and generally use "flooding" search protocols. In contrast, structured P2P networks (e.g., Chord [SMK+01], CAN [RFH+01], Pastry [RD01a], P-Grid [ACMD+03], D2B [FG03], Koorde [KK03], Viceroy [MNR02], DH DHT [NW03a], Tapestry [ZHS+04]) use specialized placement algorithms to assign responsibility for each file to specific peers. A structured P2P network distributes the responsibility for a key space over the available peers and maintains the peers' connection structure based on the set of keys each peer holds. Objects are mapped to the key space (e.g., by hashing the object id) and then assigned to

the peers (say roots) responsible for the corresponding keys. Such a system provides efficient routing and object location in which the request, directed by the structured connection, arrives at the destination within a small number of hops, often $O(\log n)$ in an n-peer system. However, the peer dynamicity (or churn, i.e., peer arrivals and departures) introduces expensive restructuring costs. These costs are called maintenance costs.

7.1.1 Resolving the load problem

In structured P2P systems, the performance, such as the response time to user requests or the capacity to store objects is critically affected by workload distribution. Overload in communication, system management, storage, or processing reduces performance. In the following, we consider two aspects of workload : index management load and storage load. P2P routing usually traverses intermediate peers between the source and the destination. This routing process uses the bandwidth of the peers along the path. Index management load refers to the bandwidth consumption for this task. The P2P system spreads objects over the peers. The resource usage for object accommodation on each peer makes up its storage load. To our knowledge, no existing system balances these two workload aspects simultaneously.

This article describes a P2P structure (namely BALLS – Balanced Load Supported P2P Structure) with the ability to simultaneously balance the index management load and the storage load while keeping the maintenance costs low. A brief description of this system was published in [LBK06c]. BALLS partitions a de Bruijn graph among the peers participating in the P2P network. By partitioning a de Bruijn graph, we harness the advantages of this type of graph, that is, low peer degree and efficient routing.

The **index management load balancing** method we propose takes into account the heterogeneity of peers, where each peer has its own capacity (called index management capacity) to accommodate for this load. The goal of index management load balancing is to minimize

the possible overload of the whole system. Unlike other load balancing methods that permanently restructure the system to direct the load on every peer to a target load or a global load/capacity ratio, our method adjusts the system only when overload occurs. It therefore saves on the cost of restructuring.

BALLS separates the peer id (peer address) from the keys it holds. This allows the peers to dynamically modify their key responsibility. Since the P2P routing is based on the de Bruijn routing paths, arrangement of key responsibility among the peers can adjust the peers' index management load, and thus enables load balancing.

The **storage load balancing** method presumes that each peer has limited space available to store objects. To facilitate storage load balancing, BALLS allows objects to be located at peers different from the peer holding the key. Hence, an object can reside on any peer, regardless of its root (i.e., the peer holding the object key). The root needs only to keep a pointer to the location of the object. This separation enables the system to employ available space from any peer to place objects in case the corresponding root has reached its capacity. This enlarges thus the overall storage capacity. It also enables and facilitates simultaneous index management load balancing since modifying the key responsibility of a peer requires that only pointers of the involved objects be moved, but not the objects themselves. Moreover, this separation simplifies object replication, in which case the root of an object keeps the pointers to its replicas (on different peers). It thus enhances object availability without the need for further techniques.

Storage management must also consider resources required for file transfer. To that end, we take into account the network capacity of the peers to support object access and migration. The storage load balancing algorithm aims to minimize the overload with regard to these capacities (storage, access, migration) in the whole system. As it is the case with index management load balancing, the overload minimization goal allows us to save on rebalancing cost.

The load balancing algorithm is based on transferring storage load between pairs of peers if it results in a decrease of the global overload.

7.1.2 Related work

There exist several approaches to load balancing in structured P2P systems. A straightforward approach is the equalization of key responsibility assigned to the peers. It aims to maximize a metric called *smoothness*, defined as :

$$\min_{\forall p,q} \frac{|k_p|}{|k_q|},$$

where $|k_i|$ is the number of keys managed by peer i. Note that smoothness is usually defined as $\max \frac{|k_p|}{|k_q|}$ (i.e., inverse of $\min \frac{|k_p|}{|k_q|}$), which we believe is less intuitive as we would then minimize "smoothness" rather than maximize it. Bienkowski et al. [BKadH05] proposed a balancing method for P2P systems in which each peer occupies a key interval. The method categorizes the peers into *short*, *middle*, and *long* according to their interval's length. In order to increase smoothness, the balancing algorithm continuously makes short peers leave and rejoin by halving the interval of an existing long peer. Manku [Man04] aims at maximizing smoothness using a virtual balanced binary tree where the leaf nodes represent the current peers of the system and the path from the root to each leaf represents the splitting chain of the key space (being $[0, 1)$) to yield the corresponding peer's interval. The author proposed appropriate peer arrival and departure algorithms that maintain the balance of the binary tree. It thereby leads to a balanced key responsibility among the peers. In practice, the balance of load in structured P2P systems also depends on the size of objects, the popularity of objects, the distribution of objects over the peers, and the peers' capacities. If these factors are accounted for, a good smoothness does not ensure load balance.

There are some structured P2P systems [LKRG03, NW03a, WZLL04] using de Bruijn graphs for topology construction. Their load balancing methods also aim to increase smoothness via different arrival and departure algorithms. As discussed above, however, smoothness is not sufficient to balance load.

Several methods [KR04a, KR04b, RLS+03] transfer key responsibility among the peers to balance load. In such methods, heavily loaded peers move part of their key responsibility to lightly loaded peers, thereby shedding some load. The above methods tie the objects' location to the objects' key. They therefore do not enable index management load and storage load to be simultaneously balanced since balancing one aspect can break the balance of the other, and vice versa.

PAST [RD01b] is a P2P storage management system based on the Pastry [RD01a] routing substrate. Its load balancing approach uses a replica diversion technique that allows an object (file) to reside on a peer in the *leaf* set of its root[2] (refer to [RD01a] for the *leaf* set definition) when the root is full. However, PAST does not separate object location from object key. It maintains an invariant that limits object location within the root and the leaf set of the root.

Another method applies the *power of two choices* paradigm [BCM03] in which a set of multiple hash functions is used to map an object to a set of keys. This method balances storage load by storing each object at the least loaded peer in the set of peers responsible for the object's keys. The above methods relax the storage policy by breaking the tie of an object to one peer. However, they restrict object storage to a set of peers, which introduces a considerable network overhead in a dynamic environment.

Tapestry [ZHS+04] does not restrict where an object is stored (i.e., its storage location). Indeed, the publication of an object places pointers to the object on the peers along the routing

[2]In PAST, an object is replicated into a number κ of copies assigned, respectively, to κ peers with ids that are the closest to the object's id. The root here denotes the peer to which a copy is assigned.

path from the storing peer to the object's root. However, Tapestry does not support the independence of peer id with respect to key responsibility. With its high peer degree and complex maintenance procedure, this feature prevents the integration of efficient index management load balancing based on the dynamic transfer of key responsibility.

Expressways [ZSZ02], an extension of CAN [RFH+01], constructs the P2P network as a virtual zone span hierarchy where leaves are basic CAN zones. Based on this hierarchy, routing has logarithmic performance. This structure supports index management load balancing in which peers with higher capacities tend to shoulder the task of higher expressway levels where routing traffic has more probability to pass. This method has some disadvantages : (1) peer degree is high due to the multi-level routing table; (2) balancing occurs only after aggregating the load and capacity information of all peers in the whole system; and (3) the balancing goal is to keep equal the load/capacity ratio of the individual peers, which yields continuous restructuring even when it is not required.

7.1.3 Paper organization

The remainder of this paper is organized as follows. Section 7.2 describes BALLS and its load balancing methods in details. Section 7.3 presents the experimental evaluation of the system. Section 7.4 validates the advantages of BALLS. Section 7.5 provides a discussion about the system and the experimental results. The last section concludes the paper.

7.2 P2P system and load balancing algorithms

This section first describes the BALLS topology and its support for routing and maintenance. It is followed by the presentation of the two load balancing mechanisms.

7.2.1 The P2P topology

The construction of BALLS was inspired by de Bruijn graphs. A de Bruijn graph defines a set of nodes V and a set of arcs A as follows :

- $V = [0, k^m - 1]$ where $m \in \mathbb{N}$, $k \in \mathbb{N}$, and $k > 1$. Each de Bruijn node is labelled by an m-(k-ary) digit identifier,
- $A = \{(u, v) \,|\, u \in V, v \in V, v = (uk + l) \mod k^m, l \in [0, k-1]\}$. Because of this rule, the degree (including both in and out arcs) of each de Bruijn node does not exceed $2k$.

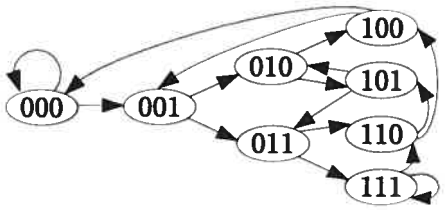

FIG. 7.1 – Binary de Bruijn graph with 8 nodes

Figure 7.1 shows an example of a binary de Bruijn graph. The de Bruijn graphs support efficient routing : The $route(u, v)$ algorithm described below performs routing from node u to node v. It requires at most m hops between any two nodes. This expresses a logarithmic bound of the routing cost in relation with the number of nodes.

route(u, v)

if $(u \neq v)$ {

$s = maxCRLSubstring(u, v)$;

$route((uk + v_{|s|}) \mod k^m, v)$;

}

Definition 1 *The longest common right-left substring of two nodes u and v, denoted*

maxCRLSubstring(u, v), *is a string* $s \in \{0, \cdots, k-1\}^*$ *such that* $u = u_0 \cdots u_{m-|s|-1}s$, $v = sv_{|s|} \cdots v_{m-1}$, *and* $u_{m-|s|-1}s \neq sv_{|s|}$.

BALLS partitions a binary undirected de Bruijn graph $G = (V, A)$ of 2^m nodes (m is the predefined node id length). The node id space $[0, 2^m - 1]$ is identical to the key space. In the following, *de Bruijn node* and *key* are used interchangeably. Each peer p holds (is responsible for) a non-empty key interval, denoted[3] $[p.b, p.e]$. For convenience, all expressions on keys are implicitly modulo 2^m, e.g., $x + y$ indicates $(x + y) \bmod 2^m$. BALLS identifies each peer p by its network address, denoted $p.a$. The separation between peer id and keys enables the change of $[p.e, p.b]$ without affecting $p.a$.

BALLS maintains an invariant : every two peers p and q are connected, denoted *connect*(p, q), if there is at least one de Bruijn arc connecting a key in $[p.b, p.e]$ and a key in $[q.b, q.e]$. In addition, p and q are connected if their key intervals are adjacent in the circular key space. We refer to this last connection type as a ring connection. The neighbourhood made up by a ring connection is called a ring neighbour. Ring connections will be needed in key interval exchange operations (e.g., index management load balancing, departure).

Definition 2 *The de Bruijn neighbourhood set of a key interval* I, *denoted* *dbneighbour*(I), *is the set of every key connected to any key in* I *by a de Bruijn arc, except the keys in* I *itself.*

It is easy to show that *dbneighbour*$([b, e]) = ([2b, 2e + 1] \cup [\lfloor b/2 \rfloor, \lfloor e/2 \rfloor] \cup [\lfloor (b + 2^m)/2 \rfloor, \lfloor (e + 2^m)/2 \rfloor]) \setminus [b, e]$. Given two peers p and q with $I = [p.b, p.e]$ and $J = [q.b, q.e]$, we have the following formula.

[3] $[b, e]$ denotes the interval of integers from b to e (inclusive). If $b \leq e$, $[b, e] = \{x \in \mathbb{Z} \mid b \leq x \leq e\}$, otherwise, $[b, e] = [b, 2^m - 1] \cup [0, e]$.

$$connect(p,q) = \begin{cases} \text{true} & \text{if } (p.e = (q.b-1) \vee p.b = (q.e+1) \\ & \quad \vee (dbneighbour(I) \cap J \neq \emptyset) \\ & \quad \vee (dbneighbour(J) \cap I \neq \emptyset)) \\ \text{false} & \text{otherwise} \end{cases}$$

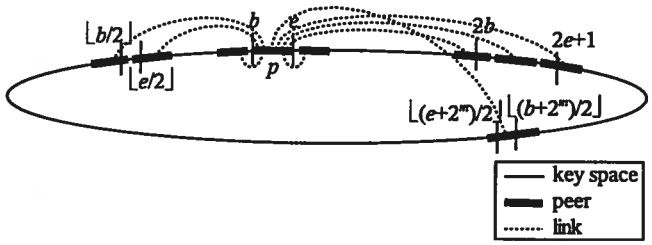

FIG. 7.2 – Example of connections from peer p in BALLS

Figure 7.2 illustrates the connections of a peer p whose key interval is $[b, e]$. The connection includes the links to the ring neighbours and to the peers holding a key interval that overlaps $dbneighbour([b, e])$.

Each peer p maintains a *neighbour list* that contains a triple $(q.a, q.b, q.e)$ for every neighbour q, i.e., the peers connected with p. Since the peer connection is undirected, p exists also in the neighbour list of q when q exists in the neighbour list of p. This facilitates key responsibility notification among the peers in comparison with other P2P systems that use directed peer connections (e.g., [KK03, MNR02]).

Loguinov et al. [LKRG03], and Naor and Weider [NW03a] use a P2P system similar to ours. They concentrate on the smoothness increase goal based on different arrival and departure algorithms. These approaches assume homogeneity of peers. In BALLS, we assume that peers are heterogeneous, that is, they have different network and storage capacities. Therefore, load balancing must be done for both types of resources.

BALLS's routing directs a message towards the peer holding a given key. We use a greedy algorithm that decreases step-by-step the distance from the current peer to the desti-

nation key. Before presenting the routing algorithm, we present some useful definitions.

In a binary 2^m-node de Bruijn graph, each node x has 4 arcs respectively to nodes : $2x$, $2x+1$, $\lfloor x/2 \rfloor$, and $\lfloor (x+2^m)/2 \rfloor$. Let the arcs to $2x$ and $2x+1$ be the fore-arcs and the arcs to $\lfloor x/2 \rfloor$ and $\lfloor (x+2^m)/2 \rfloor$ be the back-arcs. Given two nodes x and y, the length of the de Bruijn routing path from x to y that follows only fore-arcs is called the fore-distance from x to y, denoted $foredistance(x,y)$. Similarly, the length of the de Bruijn routing path that follows only back-arcs is called back-distance, denoted $backdistance(x,y)$.

Definition 3 *The distance*[4] *between two keys x and y, denoted $distance(x,y)$, is the minimum among $foredistance(x,y)$ and $backdistance(x,y)$.*

Definition 4 *The distance between a key interval I and a key x, denoted $distance(I,x)$, is equal to $distance(v,x)$ where $v \in I$ and $\nexists v' \in I \mid distance(v',x) < distance(v,x)$.*

Claim 1 *Given a node x, the set of every node y such that $foredistance(x,y) = i$ (with $i \in [0,m]$), denoted $F_i(x)$, is $[x2^i, x2^i + 2^i - 1]$.*

Proof: If $i = 0$, it is clear that $F_0(x) = \{x\}$.

If $i > 0$, suppose that $F_{i-1}(x) = [x2^{i-1}, x2^{i-1} + 2^{i-1} - 1]$ is correct. Following the fore-arcs of all nodes in $F_{i-1}(x)$, we have

$$\begin{aligned} F_i(x) &= \bigcup_{y \in F_{i-1}(x)} F_1(y) \\ &= [x2^{i-1}2, (x2^{i-1} + 2^{i-1} - 1)2 + 1] \\ &= [x2^i, x2^i + 2^i - 1] \end{aligned}$$

□

[4]Note that a routing path between two nodes in an undirected de Bruijn graph is not always the shortest path. The distance notation here does not imply the length of the shortest path. It is merely used for decision making in the greedy P2P routing algorithm.

Claim 2 *Given a node x, the set of every node y such that backdistance$(x, y) = i$ (with $i \in [0, m]$), denoted $B_i(x)$, is $\{y_0, y_1, \cdots, y_{2^i-1}\}$ where $y_j = \lfloor x/2^i \rfloor + j2^{m-i}$.*

Proof: If $i = 0$, it is clear that $B_0(x) = \{x\}$.

If $i > 0$, suppose that $B_{i-1}(x) = \{y_0, y_1, \cdots, y_{2^{i-1}-1}\}$ where $y_j = \lfloor x/2^{(i-1)} \rfloor + j2^{m-(i-1)}$ is correct. Following the back-arcs of all y_j, we have

$$B_i(x) = \bigcup_{j \in [0, 2^{i-1}-1]} B_1(y_j)$$

where

$$\begin{aligned} B_1(y_j) &= \{\lfloor y_j/2 \rfloor, \lfloor (y_j + 2^m)/2 \rfloor\} \\ &= \{\lfloor (\lfloor x/2^{(i-1)} \rfloor + j2^{m-(i-1)})/2 \rfloor, \lfloor (\lfloor x/2^{(i-1)} \rfloor + j2^{m-(i-1)} + 2^m)/2 \rfloor\} \\ &= \{\lfloor x/2^i \rfloor + j2^{m-i}, \lfloor x/2^i \rfloor + (j + 2^{i-1})2^{m-i}\} \end{aligned}$$

For all $j \in [0, 2^{i-1} - 1]$, the pair $(j, j + 2^{i-1})$ gives all integers in $[0, 2^i - 1]$.

□

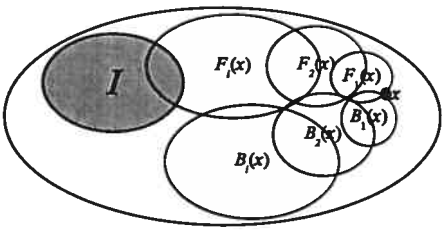

FIG. 7.3 – A set illustration of the *distance*(I, x) algorithm

The *distance*(I, x) algorithm shown below verifies the intersections $F_i(x) \cap I$ and $B_i(x) \cap I$ for i from 0 to m. If any $F_i(x) \cap I$ or $B_i(x) \cap I$ is not empty, it returns i. Figure 7.3 illustrates an approximate progression of the algorithm. The first non-empty intersection $F_i(x) \cap I$ gives distance i. The algorithm is efficient since it involves at most $m + 1$

iterations.

distance(I, x)

for $(i = 0; i \leq m; i{+}{+})$ {

if $([x * 2^i, x * 2^i + 2^i - 1] \cap I \neq \emptyset)$ return i;

$y_0 = \lfloor x/2^i \rfloor$; $s = \lfloor (I.e - y_0)/2^{m-i} \rfloor$;

if $((y_0 + s * 2^{m-i}) \in I)$ return i;

}

Routing : the routing algorithm routes a message from the current peer p to the peer holding key x.

1. if $x \in [p.b, p.e]$, the current peer is the destination. Otherwise, continue with step 2 ;
2. calculate the set $\Gamma = dbneighbour([p.b, p.e])$. Find $t \in \Gamma$ such that $distance(t, x) = distance(\Gamma, x)$. Select neighbour q such that $t \in [q.b, q.e]$. Continue routing from q.

The set Γ can contain several disjoint key intervals. The notation $distance(\Gamma, x)$ specifies the smallest distance from the key intervals in Γ to x. Key t is determined by randomly choosing a value in $F_i(x) \cap \Gamma$ or $B_i(x) \cap \Gamma$ when distance i is found. The routing algorithm decreases $distance(t, x)$ by at least 1 at each hop. Thus, the routing cost, i.e., the number of routing hops, is bounded by m. Experiments in Section 7.3.1 yield even better performances.

BALLS uses straightforward peer arrival and departure algorithms. Since the system focuses on load balancing but not on smoothness, these algorithms can be simple and yet efficient. When a peer p joins the network, it finds the root of a random key via a known peer. As soon as the root (say r) is found, r splits $[r.b, r.e]$ into two, and transfers one half to p. Since the random key is evenly selected, peers with larger key intervals have higher probability to split. Peer p informs its new neighbours (whose addresses are also obtained from r) about its new key interval and sends an acceptance message to r. Peer r then updates its key interval, informs its neighbours about the new key interval, and adjusts its neighbour

list. Suppose that the average peer degree is d, this arrival algorithm needs on average $2d + 2$ messages to maintain the P2P connection structure. Experiments in Section 7.3.1 show that d is a constant and confirm the relationship between the arrival cost and d.

When a peer q departs, it selects its ring neighbour holding the shortest key interval (say o). Then q transfers its key interval to o. If o accepts, o updates its key interval as $[o.b, o.e] \cup [q.b, q.e]$, informs the neighbours about the new key interval, and replies with an acceptance message to q. Peer q then informs its neighbours about the departure and leaves after receiving their confirmation. This departure algorithm uses on average $3d + c + 1$ messages where d is the average peer degree and c is a constant denoting the number of messages needed to select the ring neighbour o. Experiments in Section 7.3.1 confirm this cost calculation.

In order to ensure integrity of the topology, when a peer is accepting the arrival or departure of another peer, it refuses any other concurrent arrival and departure requests addressed to it. There is also an integrity problem induced from concurrent decentralized updates. Examine the following scenario :

- at time t_i, peer p_1 transfers part of its key interval to peer p_2, which involves sending p_2 a link to peer p_3 : $(p_3.a, p_3.b, p_3.e)$;
- at time t_{i+1}, p_3 updates its key interval and informs p_1 of the new $[p_3.b, p_3.e]$ (at this time, p_3 does not know about p_2) ;
- at time t_{i+2}, p_2 informs p_3 of its new key interval.

After time t_{i+2}, p_2 and p_3 connect but p_2 keeps incorrect information about p_3 : $[p_3.b, p_3.e]$. To solve this problem, node p sends its current values of $[q.b, q.e]$ to node q along with the key interval notification. If $[q.b, q.e]$ as known by p is different from the actual key interval, q sends a key interval notification back to p to correct it. This key interval notification protocol launches just enough messages for key interval updates. It does not introduce any additional costs to the maintenance.

7.2.2 Index management load balancing

Index management load refers to the network bandwidth consumed by each peer for routing messages. Due to the heterogeneous nature of the system, peers contribute different capacities, called the index management capacity, to support this load. The determination of the capacity available at each peer is out of the scope of this paper.

BALLS uses a decentralized method to distribute the index management load over the peers taking into account their capacity and current load state. A peer is overloaded when the load rises above the available capacity. The goal of load balancing is to minimize the global overload of the system. This goal reduces rebalancing costs since rebalancing is not required as long as the capacity supports the load. The balancing strategy is based on rearranging keys inside each pair of peers such that their combined overload is minimized. Because we separate peer ids and keys, a peer p may transfer some of the keys under its responsibility, thus some index management load, to a ring neighbour (i.e., a peer holding $p.b-1$ or $p.e+1$). Before going into the load balancing algorithm, we first describe how index management load is computed.

Index management load computation

The proposed index management load balancing approach requires the ability to determine the load on different subsets of a key interval. A simple solution to this requirement is to keep track of the routing traffic passing through each key in the interval. This solution becomes inefficient or even impossible when the key space size (2^m) is much higher than the number of peers.

Recall that a peer can only transfer keys to its ring neighbours to ensure that every peer holds a continuous key interval. Therefore, we only need to determine the routing traffic

passing through key subsets at the two ends of each peer's key interval.

Consequently, we propose the following approach to record traffic and compute index management load. We divide the key interval of each peer p into k levels where $k = \lfloor \log_2 s \rfloor$ and s is the size of $[p.b, p.e]$ (i.e., $s = p.e - p.b + 1$). Each level i, $i \in [0, k)$, is further broken down into 3 zones (see Fig. 7.4) : $z_{i,0} = [p.b, p.b + l_i - 1]$, $z_{i,1} = [p.e - l_i + 1, p.e]$, and $z_{i,2} = [p.b, p.e] \setminus (z_{i,0} \cup z_{i,1})$ where $l_i = \lfloor s/2^{i+1} \rfloor$. In the special case where $p.b = p.e$, only one level exists with $z_{0,0} = \{p.b\}$ and $z_{0,1} = z_{0,2} = \emptyset$.

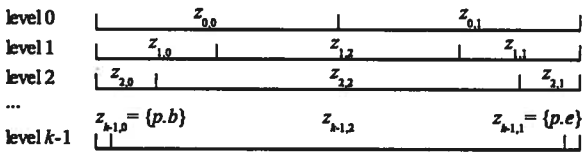

FIG. 7.4 – Dividing the key interval of peer p into zones

Each peer p manages a table $G_p[k][3]$ to register the routing traffic through the zones. According to the routing algorithm (Sect. 7.2.1), a routing message λ lands on a key t on each peer p in the path. For every level $i \in [0, k)$, if $t \in z_{i,j}$ then $G_p[i, j] = G_p[i, j] + |\lambda|$ where $|\lambda|$ denotes the size of λ. The routing traffic through peer p is $Tr_p = \sum_{j \in [0,2]} G_p[i, j]$, for any i.

The size of table G_p is always small because $k < m$. It ensures the efficiency of the routing traffic registration. Furthermore, this method allows us to determine the routing traffic through different portions at the two ends of $[p.b, p.e]$ with sizes ranging from 1 (e.g., $z_{k-1,0}$ or $z_{k-1,1}$) to $s - 1$ (e.g., $z_{k-1,0} \cup z_{k-1,2}$ or $z_{k-1,1} \cup z_{k-1,2}$).

In order to monitor index management load, which can dynamically vary over time, we must periodically reset table G_p. Denoting the period as δt, the starting time of the current period as t_0, and the current time as t_c, the current index management load is $T_p = Tr_p/(t_c - t_0)$. We sometimes need to calculate T_p when $t_c - t_0$ is too small which may result in an incorrect load. In this case, we use the formula $T_p = (Tr'_p + Tr_p)/(t_c - t'_0)$ where Tr'_p and

t'_0 are, respectively, the routing traffic and the starting time of the previous period.

We denote the index management capacity of peer p as C_p. The overload is formally given by $O_p = (T_p - C_p + |T_p - C_p|)/2$. Every peer p permanently executes the following load monitoring procedure :

1. if p has changed $p.b$ or $p.e$, go to step 4. Otherwise, go to step 2 ;
2. if $t_c - t_0 < \delta t$, go to step 1. Otherwise, go to step 3 ;
3. if $O_p > 0$, execute the index management load balancing algorithm on p. Go to step 4 ;
4. reset table $G_p[k][3]$ with $k = \lfloor \log_2(p.e - p.b + 1) \rfloor$. Set $t_0 = t_c$. Go to step 1.

Index management load balancing algorithm

The index management load balancing algorithm on a peer p starts when $O_p > 0$. It consists in transferring some part of $[p.b, p.e]$ to a ring neighbour q so as to minimize the combined overload of p and q. We denote the ring neighbours of p as $n_0(p)$ (that holds $p.b-1$) and $n_1(p)$ (that holds $p.e+1$). The zones that can be moved to $n_j(p)$ are $z_{i,j}$ and $z_{i,j} \cup z_{i,2}$ (for $i \in [0, k)$ and $j \in [0, 1]$). The interval selected to be moved must : (1) maximize the reduction of the combined overload $O_p + O_{n_j(p)}$, and (2) be as small as possible. These criteria ensure that the global overload is minimized while entailing the least rebalancing cost.

Selecting the optimal zone to move requires knowledge on the load and capacity of p and $n_j(p)$. Maintaining such information among neighbouring peers is very expensive. If p asks $n_j(p)$ about this information before transferring, it slows down the procedure. Our solution allows p to propose a set of candidate zones to $n_j(p)$. On its side, $n_j(p)$ selects the best zone based on its local load status and the information received. This solution needs only one query-answer exchange between the peers. We let $w_{h,j}$ ($h \in [0, 2k)$ and $j \in [0, 1]$) denote the candidate zones for transfer. They are determined as follows (Fig. 7.5) :

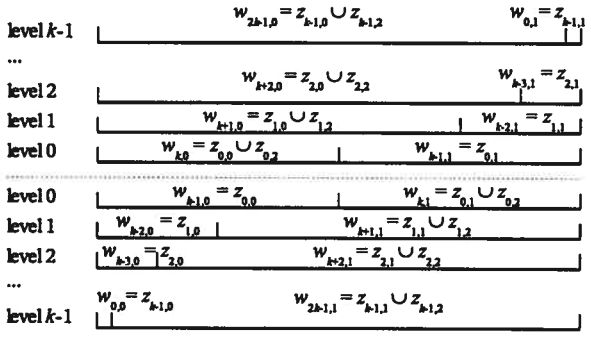

FIG. 7.5 – Candidate zones to transfer ($w_{h,j}$)

$$w_{h,j} = \begin{cases} z_{k-h-1,j} & \text{if } 0 \leq h < k \\ z_{h-k,j} \cup z_{h-k,2} & \text{if } k \leq h < 2k \end{cases}$$

The index management load on $w_{h,j}$, denoted $T(w_{h,j})$, is given by :

$$T(w_{h,j}) = \begin{cases} \dfrac{G_p[k-h-1,j]}{t_c - t_0} & \text{if } 0 \leq h < k \\ \dfrac{G_p[h-k,j] + G_p[h-k,2]}{t_c - t_0} & \text{if } k \leq h < 2k \end{cases}$$

The index management load balancing algorithm on a peer p (launched when $O_p > 0$) :

1. select the smallest $h \in [0, 2k)$ such that $\exists j \in [0, 1]$ and $T_p - T(w_{h,j}) \leq C_p$. Execute the key interval transfer algorithm (presented below) for $w_{h,j}$ from p to $n_j(p)$. If the transfer succeeds, stop the load balancing algorithm. Otherwise, continue with step 2 ;
2. set $l = (j+1) \bmod 2$. Select the smallest $h \in [0, 2k)$ such that $T_p - T(w_{h,l}) \leq C_p$. Execute the key interval transfer algorithm for $w_{h,l}$ from p to $n_l(p)$. This step stops the load balancing algorithm even if the transfer does not succeed since both ring neighbours of p have been tried.

The key interval transfer algorithm for $w_{h,j}$ from a peer p to a ring neighbour $n_j(p)$ allows $n_j(p)$ to receive one of the zones $w_{0,j}, w_{1,j}, \cdots, w_{h,j}$ from p that will minimize the

combined overload $O_p + O_{n_j(p)}$. The algorithm is straightforward. It first selects the zone that decreases the overload of p the most while keeping $n_j(p)$ underloaded (step 2a). It may be the case that no such zone is found. If the selection stops here, the transfer of load from p to $n_j(p)$ is blocked because $n_j(p)$ (being underloaded) does not intend to shed any part of its key interval. In this case, the algorithm selects the smallest zone that can decrease the combined overload to continue the load balancing (step 2b). The transfer involves the following steps :

1. p sends to $n_j(p)$ a key interval transfer message containing O_p, the list $(w_{0,j}, w_{1,j}, \cdots, w_{h,j})$, and the list $(T(w_{0,j}), T(w_{1,j}), \cdots, T(w_{h,j}))$;
2. if $n_j(p)$ is busy with another operation[5] or $O_{n_j(p)} \geq 0$, it refuses the transfer. Otherwise,
   (a) $n_j(p)$ searches for the greatest $g \in [0, h]$ that gives $T_{n_j(p)} + T(w_{g,j}) \leq C_{n_j(p)}$;
   (b) if no such g is found, $n_j(p)$ searches for the smallest $g \in [0, h]$ satisfying

$$|T(w_{g,j}) - O_p| + T(w_{g,j}) - O_p + 2(T_{n_j(p)} - C_{n_j(p)}) < 0 \tag{7.1}$$

      i. if no such g exists, $n_j(p)$ rejects the transfer;
      ii. if g is found, $n_j(p)$ sets the chosen candidate zone index as g;
3. if a candidate zone index g is chosen (by step 2a or 2(b)ii), $n_j(p)$ adds $w_{g,j}$ to its key interval, adjusts the connections to its neighbours, and replies to p an acceptance message specifying g;
4. on receiving the acceptance message from $n_j(p)$, p removes $w_{g,j}$ from its key interval and releases the connections to the peers that are no longer its neighbours. The transfer then succeeds;
5. if the proposal of p is rejected by $n_j(p)$, the transfer fails.

[5] $n_j(p)$ may be participating in the arrival, departure, or index management load balancing with another peer.

Conjecture 1 *The key interval transfer algorithm for an interval $w_{h,j}$ from a peer p to a ring neighbour $n_j(p)$, if it succeeds, will minimize the combined overload of p and $n_j(p)$.*

Proof : The key interval transfer succeeds only if a candidate zone index g is chosen in step 2a or 2(b)ii. Recall that the routing algorithm (Sect. 7.2.1) limits the choice of the next step key t in the de Bruijn neighbourhood set of the current peer. It follows that moving $w_{g,j}$ will transfer a load approximately[6] equal to $T(w_{g,j})$. Before the move, the overloads of the peers are, respectively, $O_p = (T_p - C_p + |T_p - C_p|)/2 = T_p - C_p$ and $O_{n_j(p)} = 0$. After the move, the overloads become, respectively :

$$\begin{aligned} O'_p &= (O_p - T(w_{g,j}) + |O_p - T(w_{g,j})|)/2 \\ O'_{n_j(p)} &= (T_{n_j(p)} + T(w_{g,j}) - C_{n_j(p)} + |T_{n_j(p)} + T(w_{g,j}) - C_{n_j(p)}|)/2 \end{aligned}$$

The condition for minimizing the combined overload is to minimize $\Delta O = O'_p + O'_{n_j(p)} - O_p - O_{n_j(p)}$.

If g is chosen by step 2a, $T_{n_j(p)} + T(w_{g,j}) \leq C_{n_j(p)}$. Then $O'_{n_j(p)} = 0$ and $\Delta O = O'_p - O_p < 0$. Because g is the largest possible, it induces the largest possible $T(w_{g,j})$ and thus the most negative ΔO.

If g is chosen by step 2(b)ii, $T_{n_j(p)} + T(w_{g,j}) > C_{n_j(p)}$ and (7.1) holds. It is easy to prove that the left hand side of (7.1) is $2\Delta O$ and the smallest possible g chosen induces the most negative ΔO.

□

The above algorithm ensures criterion (1) for the key interval transfer. On the other hand, the selection of the smallest h in steps 1 and 2 of the index management load balancing

[6]Because of the de Bruijn graph routing property, it cannot be guaranteed that an exact traffic through the zone moved is actually transferred to the new peer.

algorithm satisfies criterion (2), which aims at making the minimal update.

7.2.3 Storage location and key separation

The storage load of a peer p (denoted S_p) is the sum of the sizes of all objects the peer currently stores. Each peer p contributes a limited space to store objects, called the storage capacity (D_p). The system ensures the invariant $S_p \leq D_p$ on every peer p. In a P2P system where objects must reside on their root, maintaining this invariant yields the rejection of inserting a new object when its root is full (even if storage space is available on other peers). This degrades the system's storage capacity. Moreover, a redistribution of the storage load to fit the peers' storage capacity affects key distribution, which is managed by the index management load balancing.

BALLS solves the above inconveniences by allowing objects to be located at peers other than their root (i.e., the peer holding their key). A root keeps track of the objects under its responsibility via pointers, called *storage pointers*, that keep the address of the peers storing the objects. An object keeps track of its root through a pointer, called *root pointer*. This approach greatly improves the utilization of the globally available storage capacity. It enables new objects to be stored as long as sufficent space remains in the system. More importantly, rearranging objects when performing storage load balancing (presented in Sect. 7.2.4) does not influence index management load balancing. This approach also ensures the efficiency of the index management load balancing because the transfer of a key interval only moves the storage pointers of the objects involved, not the objects themselves. This therefore saves on data migration costs.

Another advantage of this approach is that it simplifies replication, which enhances object availability. Because of the separation of object locations and keys, storing replicas of an object on different peers is natural without the need of an additional special technique such

as multiple mapping hash functions (e.g., [BCM03]) or overlapping key responsibility (e.g., [NW03b]). BALLS allows up to κ replicas of an object to be stored. The root keeps storage pointers to the replicas as for individual objects. For convenience, we will use the term object to imply an object or one of its replica interchangeably in this paper.

Pointer management

Pointer management maintains the consistency of storage pointers and root pointers. Every peer disposes of two tables : *indices* and *storage*. Each item in table *indices* represents the index of an object managed by the peer. Such an item consists of the object id (*oid*) and the list of storage pointers to the object's replicas. A storage pointer consists of the replica id (*rid* - an integer in the range $[0, \kappa)$), the address of the storing peer (*location*), and the storage counter (*counter*). The *counter* field (its use will be explained below) is initially set to 0 and incremented after each change of *location*. The *storage* table contains the objects currently stored on the peer. The header of each item in this table consists of the object id (*oid*), the replica id (*rid*), the size (*size*), the root pointer (*root* being the address of the root), and the storage counter (*counter*). BALLS uses two protocols : *storage notification* and *root notification* to keep the *indices* and *storage* tables consistent.

The storage notification protocol informs the root of an object about the object's location. When a peer p accepts an object o, $o.counter$ is incremented. Peer p then sends a storage notification message σ to the actual root of o. The message contains $p.a$ and (*oid*, *rid*, *counter*) of o. We anticipate the following problems :

- due to the heterogeneity and decentralization properties of the P2P system, there may be more than one storage notification message (of an object) arriving at the root in an order other than the order of the generation time. The *counter* field lets the root know whether the notification received is newer than the corresponding storage pointer it

holds. If this is the case, the root updates the storage pointer;

- the root of o may change when p launches the notification message. Message σ therefore attaches a $root$ field specifying the root pointer as known by p. If a peer that receives σ is not the actual root of o, or if the sending peer p finds that the peer at address $\sigma.root$ no longer exists (because it has left the overlay network), σ is sent to o's key using the routing algorithm. Forwarding σ in this case does not influence index management load balancing since the load calculation only counts routing messages created by lookups from users but not by notification messages. In other words, it determines the load according to the requests but not to object migration. If the actual root of o (say r) receives σ, it compares $\sigma.root$ with $r.a$. If $\sigma.root \neq r.a$ (i.e., p keeps an incorrect root pointer), r sends back a root notification (explained below) to p for the correction;
- in the case where o moves from peer p to peer q, it may still be accessed on p after the move but before the root receives the storage notification. To solve this problem, p keeps a storage pointer to q. As soon as the root updates o's storage pointer, the corresponding pointer on p is deleted.

The root notification protocol informs an object about its root. When a key interval moves from a peer to another peer (say q), q sends root notification messages to the peers storing the objects of which the key was moved. A root notification message ρ for an object o contains $q.a$, (oid, rid, $counter$) of o, and a $location$ field specifying the address of the peer storing o as known by q. When a peer s that actually stores o receives ρ, it updates the corresponding root pointer and verifies that q keeps a correct storage pointer (using the information received). If this is not the case, s sends a storage notification message to q to perform the correction.

The two notification protocols introduced seem to complicate system maintenance. However, in comparison to traditional P2P systems that tie objects to their roots, the transfer of

a key interval in BALLS remains much less costly. It only requires sending storage pointers and notification messages whose sizes, in practice, are much smaller than the object sizes.

Object insertion

The object insertion algorithm adds up to κ replicas of a new object to different peers while ensuring $S_p \leq D_p$ on every peer p. An insertion request containing the object id (oid) and the size ($size$) is routed to the root of oid. If an index entry with the same oid as the object to insert already exists, the request is rejected. Otherwise, a replica diffusion process is launched to store the object on different peers, starting at the root.

This process sends a replica diffusion message λ_r containing oid, $size$, and $ridlist$ – the list of the remaining rids to be assigned. This message traverses multiple peers in order to distribute the replicas. It limits the number of peers visited using a ttl (time-to-live) field, which is decremented after each hop. At each peer q on the way, if $S_q + size \leq D_q$ and q does not store any replica of the same object, q accepts one replica with a rid extracted from $\lambda_r.ridlist$. If $\lambda_r.ridlist$ is not empty and $\lambda_r.ttl > 0$, λ_r is forwarded to a randomly selected not-visited peer neighbouring q. The message keeps a list of the visited peers to perform this verification. The insertion stops when κ replicas have been stored. If $\lambda_r.ttl = 0$, there are two cases : (1) if no replica was stored, the insertion fails, and (2) if the number of stored replicas falls between 1 and $\kappa - 1$, the root starts a new replica diffusion message for the remaining rids. During the diffusion process, if the key of the inserted object moves to another peer, the new root continues the process perserving the current state.

We do not discuss the object deletion here since it never increases the storage load on any peer.

7.2.4 Storage load balancing

In addition to storage space, storing objects consumes other computational resources, e.g., network bandwidth for object access and migration. These resources must be considered in the balancing algorithm.

For the system to work properly, we need another boundary for the storage load called the desired storage capacity $\overline{D}_p$ on every peer p, with $\overline{D}_p \leq D_p$. The storage load S_p is always limited by D_p but can temporarily exceed $\overline{D}_p$. Given $A_p = \overline{D}_p - S_p$ denoting the available space on peer p, we define the storage overload as $W_p = (|A_p| - A_p)/2$. Peer p is overloaded when $W_p > 0$. The goal of storage load balancing is to minimize the storage overload of the whole system ($\sum_{\forall p} W_p$) while keeping $S_p \leq D_p$ on every peer.

The storage load balancing algorithm consists in transferring storage load within pairs of peers so as to minimize their combined overload. Such transfers lead to the reduction of the global overload. Suppose that we perform a storage load transfer within a pair of peers p and q where p is overloaded ($A_p < 0$). The transfer decreases the combined overload $W_p + W_q$ only if $A_q > 0$. The storage load balancing should minimize the global overload while saving on the transfer cost (i.e., the data volume migrated). Therefore, the elementary storage load transfer within each pair of peers has the following objectives : (1) minimizing their combined overload, and (2) minimizing transfer cost.

In general, we cannot achieve both objectives at the same time. Therefore, priority must be given to one or the other : (1) before (2) or (2) before (1). These two different orders are referred to as storage load transfer strategies. The overload-oriented transfer strategy minimizes overload first, while the cost-oriented transfer strategy considers transfer cost first.

Another element to consider in storage balancing is whether transfer is unidirectional (i.e., only one peer transfers object) or bidirectional (i.e., both peers may transfer objects).

We refer to these as the 1-direction transfer and 2-direction transfer, respectively.

The 1-direction storage load transfer

An 1-direction storage load transfer is a storage load S_{pq} transfered by overloaded peer p to a peer q (with $A_q > 0$).

Definition 5 *The optimal 1-direction storage load transfer is an 1-direction storage load transfer that upholds :*

- *(1) the combined overload of p and q is minimized,*
- *(2) S_{pq} is the smallest possible.*

Therefore, the optimal 1-direction storage load transfer is the least costly transfer that yields the fastest decrease of the combined overload $W_p + W_q$.

Definition 6 *The bounded optimal 1-direction storage load transfer is an 1-direction storage load transfer that upholds :*

- *(1) the combined overload of p and q is minimized,*
- *(2) S_{pq} is smaller or equal to reduction of the combined overload.*

Hence, the bounded optimal 1-direction storage load transfer is the most effective transfer (i.e., the one minimizing $W_p + W_q$), in which the migration cost does not exceed the benefit of overload reduction.

Theorem 1 *Given two peers p, q, with $A_p < 0$ and $A_q > 0$,*

- *the optimal 1-direction storage load transfer occurs when S_{pq} is the closest to $\min(-A_p, A_q)$ such that $S_{pq} < -A_p + A_q$;*
- *the bounded optimal 1-direction storage load transfer occurs when S_{pq} is the greatest satisfying $S_{pq} \leq \min(-A_p, A_q)$.*

The proof is presented in Appendix 7.A.

The 2-direction storage load transfer

The 2-direction storage load transfer is a pair of storage loads (S_{pq}, S_{qp}) such that S_{pq} is transfered from peer p to peer q, and S_{qp} is transfered from q to p, where $A_p < 0$ and $A_q > 0$. Obviously, $W_p + W_q$ decreases only if $0 \leq S_{qp} < S_{pq}$.

Definition 7 *The optimal 2-direction storage load transfer is the 2-direction storage load transfer that upholds condition (1) first, then condition (2) :*

- *(1) the combined overload of p and q is minimized,*
- *(2) S_{qp} is the smallest possible.*

As in the 1-direction transfer mode, the conditions of this definition guarantee the greatest reduction of $W_p + W_q$ while requiring the smallest S_{qp} to save on transfer cost. We do not consider the "bounded" constraint here because a 2-direction transfer usually incurs a migration cost higher than the overload reduction.

Theorem 2 *Given two peers p, q, with $A_p < 0$ and $A_q > 0$, and S_{pq}, the optimal 2-direction storage load transfer is obtained as follows :*

- *if $S_{pq} \leq A_q$ or $A_q < S_{pq} \leq -A_p$, $S_{qp} = 0$;*
- *if $S_{pq} > \max(-A_p, A_q)$, S_{qp} is the closest to $\min(A_p, -A_q) + S_{pq}$ such that $0 \leq S_{qp} < S_{pq}$ and $S_{qp} > A_p - A_q + S_{pq}$.*

See Appendix 7.B for the proof of this theorem.

It follows from these theorems that the most effective ΔW that both 1-direction and 2-direction transfers can achieve is $\max(A_p, -A_q)$ (also refer to the appendices). In other words, these transfer styles have the same bound.

We now return to the overload-oriented and cost-oriented transfer strategies. An overload-oriented transfer tries to minimize the combined overload before considering the transfer cost involved. On the other hand, a cost-oriented transfer restricts the transfer cost not to exceed the gain while decreasing the combined overload. To fulfil its requirement, the cost-oriented transfer strategy applies the bounded optimal 1-direction transfer. The overload-oriented transfer strategy is more complicated. It first searches for the optimal 1-direction transfer. If no S_{pq} relevant to such a transfer is found, it searches for the optimal 2-direction transfer.

In a decentralized environment, an object move consists in the proposal of an object set R by the sending peer (say p) and the acceptance of a subset $R' \subseteq R$ by the receiving peer (say q). To avoid sending an object to different peers simultaneously, we define two object states : *moving* (i.e., the object is in a proposal) and *normal* (i.e., the object is not in any move). Therefore, when peer p proposes set R to q, these objects are marked as *moving* on p. After q accepts objects in R', p deletes R' and restores the *normal* state of the remaining objects in $R \setminus R'$. Other simultaneous moves only select objects in the *normal* state. We denote the set of the objects stored on p as R_p, the set of its objects in the *normal* state as $\overline{R}_p$, the storage load of $\overline{R}_p$ as $\overline{S}_p$, and $\overline{A}_p = \overline{D}_p - \overline{S}_p$.

The storage load balancing algorithm : each peer p periodically verifies its overload state. When $\overline{A}_p < 0$, p starts a balancing session :

1. p broadcasts to its neighbourhood an available space interrogation message ϕ with a *ttl* (time-to-live) field defining the diffusion depth. If it has not received ϕ before, each receiving peer q processes ϕ and decrements $\phi.ttl$. After decrementing, if $\phi.ttl > 0$, q forwards ϕ to its neighbours except p and the peer from which ϕ arrives. The processing of ϕ on q involves replying $A_q = \overline{D}_q - S_q$ to p if $A_q > 0$;
2. for each reply A_q, if $\overline{A}_p < 0$ still holds, p proposes to q an object set $R_{pq} \subseteq \overline{R}_p$ that

satisfies one of the following conditions (denoting the storage load of a set R as $S(R)$) :

$$\begin{cases} S(R_{pq}) \leq A_q \\ S(R_{pq}) \geq -\overline{A}_p \\ \nexists r \in R_{pq} \mid S(R_{pq} \setminus \{r\}) \geq -\overline{A}_p \end{cases} \tag{7.2}$$

$$\begin{cases} S(R_{pq}) \leq A_q \\ S(R_{pq}) < -\overline{A}_p \\ \nexists r \in \overline{R}_p \setminus R_{pq} \mid S(R_{pq} \cup \{r\}) \leq A_q \end{cases} \tag{7.3}$$

$$\begin{cases} S(R_{pq}) > A_q \\ |R_{pq}| = 1 \\ \nexists r \in \overline{R}_p \mid S(\{r\}) < S(R_{pq}) \end{cases} \tag{7.4}$$

(7.2) selects the smallest $S(R_{pq})$ that cancels overload on p without overloading q. If (7.2) cannot be satisfied, (7.3) selects the greatest $S(R_{pq})$ that reduces W_p without overloading q. If both (7.2) and (7.3) cannot be satisfied (i.e., every object in $\overline{R}_p$ is greater than A_q), (7.4) assigns the smallest object of $\overline{R}_p$ to R_{pq} ;

3. on receiving the R_{pq} proposal, q behaves differently according to the storage load transfer strategy (cost-oriented or overload-oriented) selected a priori.

In the cost-oriented storage load transfer, q selects $R'_{pq} \subseteq R_{pq}$ as follows, where $pivot = \min(-\overline{A}_p, A_q)$:

$$\begin{cases} 0 < S(R'_{pq}) \leq pivot \\ \nexists r \in R_{pq} \setminus R'_{pq} \mid S(R'_{pq} \cup \{r\}) \leq pivot \\ S(R'_{pq}) + S_q \leq D_q \\ \nexists (r, r') \mid r \in R'_{pq} \wedge r' \in R_q \wedge r.oid = r'.oid \end{cases} \tag{7.5}$$

Condition (7.5) selects R'_{pq} that will make the bounded optimal 1-direction transfer as indicated by Theorem 1. R'_{pq} also ensures that $S_q \leq D_q$ and does not contain any replica of an object already stored on q (which causes a replica conflict). If no such R'_{pq} is found, q refuses the proposal. Otherwise, it accepts R'_{pq}. The transfer is completed.

In the overload-oriented storage load transfer, q first selects an object set R'_{pq} as follows, where $pivot = \min(-\overline{A}_p, A_q)$:

find $R1 \subseteq R_{pq}$ satisfying (7.6)

$$\begin{cases} 0 < S(R1) < pivot \\ \nexists r \in R_{pq} \setminus R1 \mid S(R1 \cup \{r\}) < pivot \\ S(R1) + S_q \leq D_q \\ \nexists (r, r') \mid r \in R1 \wedge r' \in R_q \wedge r.oid = r'.oid \end{cases} \tag{7.6}$$

find $R2 \subseteq R_{pq}$ satisfying (7.7)

$$\begin{cases} pivot \leq S(R2) < -\overline{A}_p + A_q \\ \nexists r \in R2 \mid S(R2 \setminus \{r\}) \geq pivot \\ S(R2) + S_q \leq D_q \\ \nexists (r, r') \mid r \in R2 \wedge r' \in R_q \wedge r.oid = r'.oid \end{cases} \tag{7.7}$$

identify R'_{pq} as follows, where $\Delta W(S_{pq})$ is the variation of the combined overload as defined by (7.10) :

$$R'_{pq} = \begin{cases} R1 & \text{if } \exists R1 \wedge \nexists R2 \\ R1 & \text{if } \exists R1 \wedge \exists R2 \wedge \Delta W(S(R1)) \leq \Delta W(S(R2)) \\ R2 & \text{if } \exists R2 \wedge \nexists R1 \\ R2 & \text{if } \exists R1 \wedge \exists R2 \wedge \Delta W(S(R1)) > \Delta W(S(R2)) \end{cases}$$

The above conditions try to select R'_{pq} to make the optimal 1-direction transfer as stated in Theorem 1. They also ensure that R'_{pq} does not contain any replica of an object already stored on q and guarantees $S_q \leq D_q$. If such an R'_{pq} is identified, q accepts R'_{pq} and the transfer is completed. Otherwise (i.e., $\nexists R1$ and $\nexists R2$), the transfer continues in the 2-direction transfer mode.

In the 2-direction transfer, q selects an acceptable object set $R'_{pq} \subseteq R_{pq}$ such that

$$\begin{cases} S(R'_{pq}) + S_q \leq D_q \\ \nexists r \in R_{pq} \setminus R'_{pq} \,|\, S(R'_{pq} \cup \{r\}) + S_q \leq D_q \\ \nexists (r, r') \,|\, r \in R'_{pq} \wedge r' \in R_q \wedge r.oid = r'.oid \end{cases}$$

This selection gives the greatest R'_{pq} that guarantees $S_q \leq D_q$ and does not contain any replica of an object already on q. If no such R'_{pq} is found, q refuses the proposal. In case q finds an R'_{pq}, it selects an object set to send back R_{qp} according to the following rule, where $pivot = \min(\overline{A}_p, -A_q) + S(R'_{pq})$:

find $R1 \subseteq \overline{R}_q$ satisfying (7.8)

$$\begin{cases} 0 \leq S(R1) < pivot \\ S(R1) > \overline{A}_p - A_q + S(R'_{pq}) \\ \nexists r \in \overline{R}_q \setminus R1 \,|\, S(R1 \cup \{r\}) < pivot \end{cases} \tag{7.8}$$

find $R2 \subseteq \overline{R}_q$ satisfying (7.9)

$$\begin{cases} S(R2) \geq pivot \\ S(R2) < S(R'_{pq}) \\ \nexists r \in R2 \,|\, S(R2 \setminus \{r\}) \geq pivot \end{cases} \tag{7.9}$$

identify R_{qp} as follows, where $\Delta W(S_{qp})$ is the variation of the combined overload as

defined by (7.12) :

$$R_{qp} = \begin{cases} R1 & \text{if } \exists R1 \wedge \nexists R2 \\ R1 & \text{if } \exists R1 \wedge \exists R2 \wedge \Delta W(S(R1)) \leq \Delta W(S(R2)) \\ R2 & \text{if } \exists R2 \wedge \nexists R1 \\ R2 & \text{if } \exists R1 \wedge \exists R2 \wedge \Delta W(S(R1)) > \Delta W(S(R2)) \end{cases}$$

The selection aims to identify an object set R_{qp} that results in the optimal 2-direction storage load transfer, as required in Theorem 2. If $\nexists R1$ and $\nexists R2$, R_{qp} is not identified and q refuses the proposal. Otherwise, it accepts R'_{pq} and proposes R_{qp} to p.

Upon receiving the R_{qp} proposal, p selects $R'_{qp} \subseteq R_{qp}$ such that

$$\begin{cases} S(R'_{qp}) + S_p \leq D_p \\ \nexists r \in R_{qp} \setminus R'_{qp} \,|\, S(R'_{qp} \cup \{r\}) + S_p \leq D_p \\ \nexists (r, r') \,|\, r \in R'_{qp} \wedge r' \in R_p \wedge r.oid = r'.oid \end{cases}$$

This maximizes R'_{qp} while guarantying that $S_p \leq D_p$, and makes no replica conflict. Then p accepts R'_{qp} and the transfer is completed.

The storage load balancing algorithm uses $\overline{A}_p = \overline{D}_p - \overline{S}_p$ instead of $A_p = \overline{D}_p - S_p$ (as in Theorems 1 and 2) to enable parallel object transfers from the overloaded peer p to different underloaded peers. The inherent parallelism accelerates the minimization of the global overload.

Our implementation uses a combination of several greedy algorithms for the selection of R_{pq}, R'_{pq}, R_{qp}, and R'_{qp}. Although the greedy algorithms do not always give the optimum, they ensure good computational efficiency.

7.3 Experimental evaluation

In order to evaluate the performance of the proposed P2P system and load balancing methods, we have performed various simulations. The simulator was developed in Java. We executed simulations on different computers with Pentium 4 CPU, (512MB - 2GB) RAM, and OS Windows XP or Red Hat Fedora Core 3. The reader can refer to [LBK06b] for details about the simulator. This section presents the results obtained. The simulations were designed to evaluate the P2P topology, the index management load balancing, and the storage load balancing. We also tested the influence of the index management load balancing on the storage load balancing, and vice versa. In all experiments presented, the P2P network partitions a de Bruijn graph of 2^{32} nodes ($m = 32$). It thus manages the key space $[0, 2^{32} - 1]$.

Each characteristic was evaluated with multiple simulation runs. This required us to perform statistical analyses of the results obtained. The analyses are of two types : (1) verification of the confidence interval of the results, and (2) comparison of two or more groups of results. We usually present the results in the form of a graph of averages of the measured values. The first analysis calculates the confidence interval of each point in the graph. The confidence interval of a variable x is the interval $[x_l, x_u]$ such that $P(\overline{x} \notin [x_l, x_u]) \leq \alpha$ with a given α and with $\overline{x}$ denoting the mean of x. Narrower confidence intervals show more closeness of the measured means to the true means. In this paper, we use $\alpha = 1\%$.

Comparisons between groups of results are used to determine if the groups differ. A group of results is produced by an experiment condition and presented by a graph (of average values). Through the comparisons, we observe the effect of the chosen conditions on the experiment results. Suppose that we compare two graphs X and Y. The comparison consists in calculating the confidence interval (with a given α) of $(X_i - Y_i)$ for all values i on the horizontal axis. If the confidence interval of $(X_i - Y_i)$ does not include 0, X_i and Y_i are statistically different. We name the rate of pairs (X_i, Y_i) having confidence interval disjoint

from 0 as the difference rate. Higher difference rates show more evidence of differences between X and Y. With lower difference rates (e.g., under 10%), we cannot conclude that X and Y are different. When performing a comparison between more than two graphs, we perform a pair-wise comparison.

7.3.1 P2P topology evaluation

The evaluation of the P2P topology consists in observing the following properties : average and distribution of peer degrees, average routing costs, average arrival costs, and average departure costs. The graphs in this subsection plot the average or combined results of 30 simulations. A simulation starts from a system of one peer. Peers arrive and depart, respectively, with probabilities related to the current number of peers. The arrival probability is higher than the departure probability. This makes the system size grow as a Markov chain. We measured the properties until the system reached 2100 peers.

During the simulation, index management load balancing is active. Peer arrivals/departures and index management load balancing continuously modify the key intervals of the peers, which causes high dynamicity.

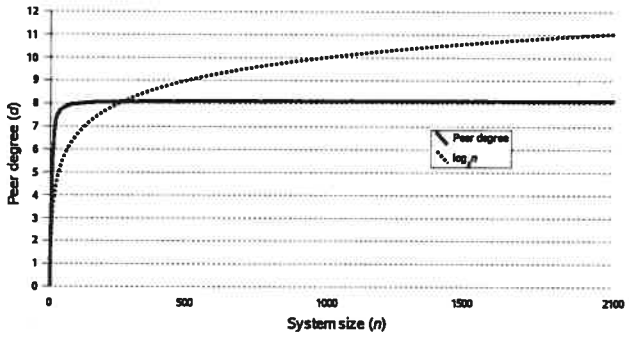

FIG. 7.6 – Average peer degree d vs. system size n

Figure 7.6 shows the average peer degree in function of system size (n). Regardless of the value of n, the average peer degree stays near 8. We obtained the same results, regardless

of the de Bruijn node id length m. We computed the confidence interval of the average peer degree for all system sizes from 1 to 2100. The limits obtained for each system size are so close to the measured average peer degree that we cannot distinguish them. The distance of the upper and lower boundaries of these intervals to the measured average peer degree gives the following statistics : max = 0.13, mean = 0.01, and median = 0.01. That is, the confidence interval for any system size is on average [8.05, 8.07]. The intervals are very close in comparison to the measured average peer degree 8.06.

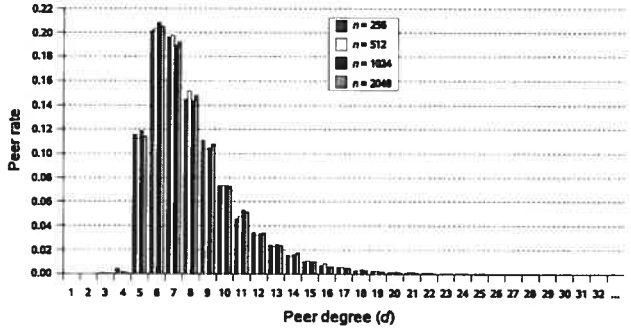

FIG. 7.7 – Peer degree distribution for different system sizes

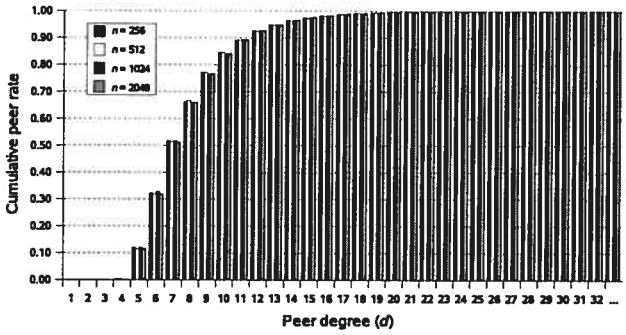

FIG. 7.8 – Cumulative peer degree distribution for different system sizes

Figure 7.7 shows the peer degree distribution calculated when n reaches 2^8, 2^9, 2^{10}, and 2^{11}. Figure 7.8 plots the cumulative rate of the peers having a degree lower than or equal to each value from 1 to 32. Although the network size increases, the distribution shape (in both graphs) almost does not change. The highest rates belong to the peers with degree from 5 to

11. It approaches 0 when the peer degree exceeds 20. Theses observations demonstrate the constant peer degree feature of the P2P topology.

It should be noted that low peer degree may lead to a fragile P2P system. We can resolve this problem by accepting some redundancy, e.g., letting each peer maintain links to multiple peers in a larger neighbourhood. The trade-off needs to be made between low peer degree and resilience to failure. Note, however, that this problem is beyond the scope of the present paper. In the following, we therefore only consider the case without redundancy.

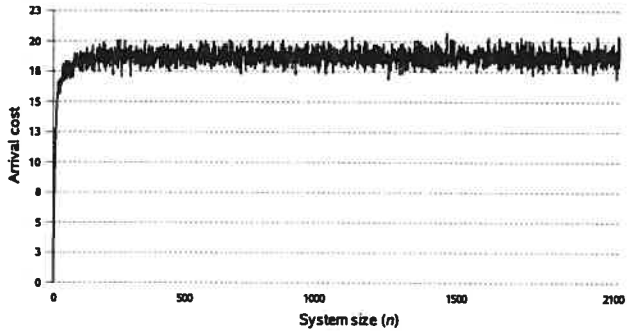

FIG. 7.9 – Arrival cost vs. system size n

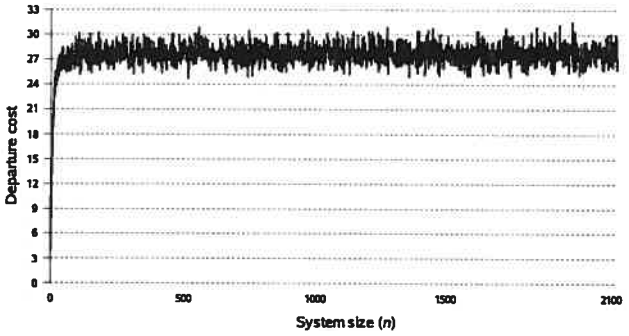

FIG. 7.10 – Departure cost vs. system size n

Figures 7.9 and 7.10 present the costs of peer arrival and departure, respectively. They show that although the system size increases, the arrival costs and the departure costs are on average 18.61 and 27.58, respectively. According to the arrival and departure algorithms described in Section 7.2.1, the average arrival and departure costs are $2d + 2$ and $3d + c + 1$,

respectively (where c is the number of messages needed to find the ring neighbour that can receive the departing peer's key interval). With $d = 8.06$ (as observed in Fig. 7.6) and $c = 2.4$, the above arrival cost and departure cost formulae give 18.12 and 27.58, respectively, which are consistent with the experimental results.

As for the peer degree evaluation, we calculated the confidence intervals of the arrival cost and of the departure cost for all system sizes from 1 to 2100. The distance between the bounds found and the measured average arrival costs has max = 2.36, mean = 1.40, and median = 1.38. Similarly, the distance between the bounds and the measured average departure costs has max = 6.81, mean = 2.74, and median = 2.66. These numbers show that the confidence intervals of the arrival costs and of the departure costs are close.

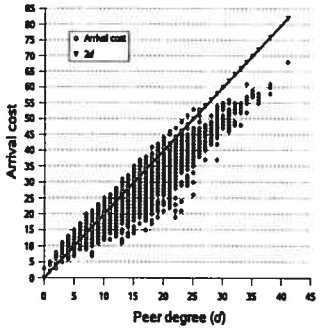

FIG. 7.11 – Arrival cost vs. peer degree d

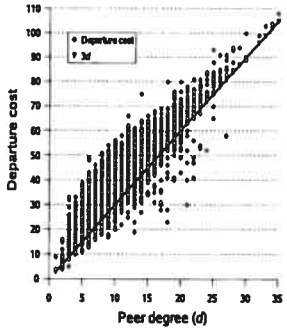

FIG. 7.12 – Departure cost vs. peer degree d

Figures 7.11 and 7.12 explore the relationships between the arrival/departure costs and the peer degree. Figure 7.11 plots the arrival costs corresponding to the degree (d) of the splitting peer. It shows that the arrival costs fall around $2d$. Similarly, Figure 7.12 plots the departure costs corresponding to the degree (d) of the departing peer. In this case, the departure costs cling to $3d$. The above observations suggest linear relationships between the arrival/departure costs and the peer degree. This explains the constant arrival/departure costs feature, which follows from the constant peer degree feature.

Figure 7.13 compares the average routing cost measured with $\log_2 n$. Although de Bruijn

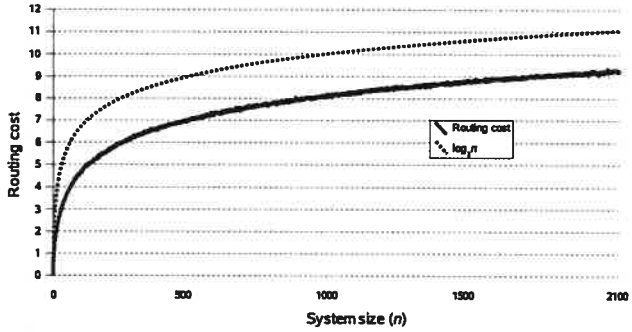

FIG. 7.13 – Routing cost vs. system size n

graph diameter m (being 32 here) is the upper bound of the routing cost, the graph shows that the average routing cost is under $\log_2 n$. With on average over 4100 routing requests finished at each system size from 1 to 2100 peers, we calculated the distance from the measured average routing costs to one limit of the confidence intervals. The observed distance having max = 0.13, mean = 0.09, and median = 0.09 shows very close confidence intervals. These results confirm the routing cost efficiency of the P2P topology.

7.3.2 Index management load balancing evaluation

The evaluation of index management load balancing explores how the global overload is influenced by the balancing algorithm. The metric for evaluating effectiveness is the index management overload ratio $\Omega = (\sum_{\forall p} O_p)/(\sum_{\forall p} T_p)$. To demonstrate the effectiveness of the load balancing, the experiments follow a 3-phase scenario. This scenario consists in running the simulation without load balancing for the first 30 simulation cycles (phase 1), then running with load balancing for the next 70 simulation cycles (phase 2), and finally running without load balancing for additional 30 simulation cycles (phase 3). When load balancing is active, the overloaded state is verified every simulation cycle.

As suggested by numerous research (e.g., [GFJ+03, LRS02, SGD+02, ZSZ03]) we ap-

ply the Zipf distribution[7] for the peers' capacity and the routing skewness. In particular, the Zipf exponent for the peers' index management capacity distribution is $\varepsilon = -1.2$. The routing source probability and the routing target probability[8] distributions are under the Zipf law with exponent $\varepsilon = -1.9$. Note that we have realized experiments with different Zipf exponents for the above distributions and also with a uniform distribution. However, they all gave results similar to those described below.

We consider three factors that can affect the load balancing result : the index management utilization ratio, the dynamicity of the routing target, and the dynamicity of the peers. The index management utilization ratio (U) is the ratio of the total index management load over the total capacity : $U = (\sum_{\forall p} T_p)/(\sum_{\forall p} C_p)$. The dynamicity of routing target (τ) denotes the change of routing target popularity in the system. It is the percentage of times a key changes its routing target probability at each simulation cycle (e.g., $\tau = 20\%$ means twenty changes within 100 cycles). The peer dynamicity (π) specifies the probability that a peer splits its keys with a joining peer and the probability that a peer departs in one simulation cycle. In these experiments, the arrival and departure probabilities are equal to maintain a relatively stable system size. Since we choose a certain U for each experiment to evaluate the Ω produced, the evolution of the system size is not required. Therefore, if a simulation cycle starts with n peers, it will have approximately $n\pi$ arrivals and $n\pi$ departures. The following experiments ran from a starting system size of 2048 peers. The graphs in this section show the average results of 20 experiments in each case.

Figure 7.14 plots the results of experiments in three cases with the same $\tau = 0\%$ and $\pi = 0\%$ but different U levels, which are, respectively, $[25\%, 30\%]$, $[55\%, 65\%]$, and $[100\%, 110\%]$. In these experiments, phase 1 (without load balancing) raises Ω to the maxi-

[7]The Zipf distribution defines that the i^{th} popular value is proportional to i^{ε} for some so called Zipf exponent ε.

[8]The probabilities for selection of, respectively, the originating peer and the destination key in lauching a simulated routing request.

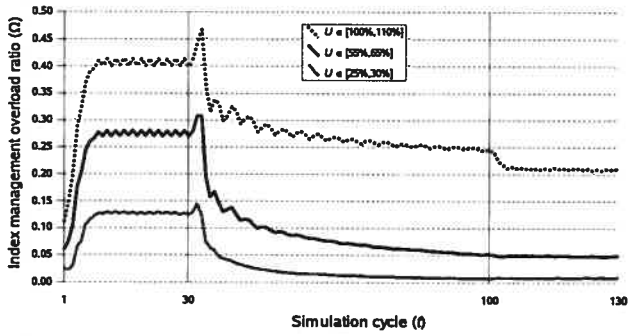

FIG. 7.14 – Index management overload ratio Ω vs. simulation cycle t for different index management utilization ratios U ($\tau = 0\%$, $\pi = 0\%$)

mum (e.g., 41.3% when $U \in [100\%, 110\%]$ or 12.9% when $U \in [25\%, 30\%]$). Phase 2 rapidly decreases Ω as a result of load balancing. In phase 3, when the load balancing is turned off, Ω remains stable at a low level (e.g., 20.6% when $U \in [100\%, 110\%]$ or 0.9% when $U \in [25\%, 30\%]$). These results demonstrate the effectiveness of the balancing algorithm. Once the load balance is established by phase 2, the global overload does not increase (in phase 3) even if the load balancing has stopped.

It is natural that higher values of U induce higher Ω. However, we compared the experiments under different U ratios to confirm that the load balancing method takes effect in a large range of capacity utilization. Even if the global load/capacity ratio falls within $[100\%, 110\%]$, the balancing method can reduce the global overload by half.

In the above graphs, there is a peak at the beginning of phase 2 and a fall at the beginning of phase 3. This follows from the difference in operation between phase 2 and phases 1 and 3. The load balancing in phase 2 requires every overloaded peer p to keep table G_p's value as long as key interval changes occur. However, phases 1 and 3 reset G_p every cycle. This makes Ω at the beginning and the end of phase 2 higher than that at, respectively, the end of phase 1 and the beginning of phase 3.

In order to observe the effect of routing target dynamicity on load balancing, we ran

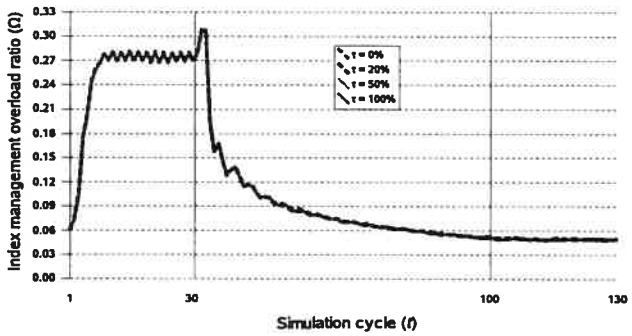

FIG. 7.15 – Index management overload ratio Ω vs. simulation cycle t for different routing target dynamicity factors τ ($U \in [55\%, 65\%]$, $\pi = 0\%$)

experiments at the same U and π but with different τ values. Figure 7.15 depicts the experimental results in four cases where $U \in [55\%, 65\%]\%$, $\pi = 0\%$, and τ, respectively, 0%, 20%, 50%, and 100%. In spite of the routing target dynamicity, which varies from 0% to 100%, the efficiency of load balancing is almost the same across the experiments. When a pairwise comparison of the results of Figure 7.15 is performed, the difference rates obtained are all 0%, which implies that we cannot recognize any difference between the graphs. This expresses a weak effect of routing target dynamicity on load balancing. This can be explained by the good randomization characteristics of the de Bruijn graph when transferring routing requests over de Bruijn nodes.

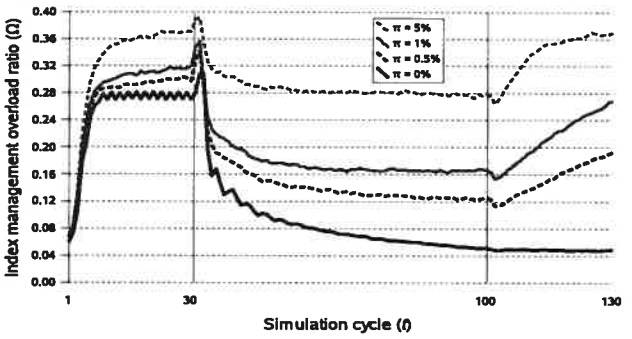

FIG. 7.16 – Index management overload ratio Ω vs. simulation cycle t for different peer dynamicity factors π ($U \in [55\%, 65\%]$, $\tau = 0\%$)

Figure 7.16 compares the results of experiments in four cases with $U \in [55\%, 65\%]$,

$\tau = 0\%$, and π values of 0%, 0.5%, 1%, and 5%, respectively. We need to explain why 5% is chosen as the highest π for experiments. The duration of a simulation cycle is the maximal time needed for the communication between any pair of peers. Suppose that the system runs in a slow environment, which requires up to 60 seconds for a pairwise communication. Thus, with $\pi = 5\%$ per cycle, 40% of peers arrive and 40% of peers depart every 8 minutes. In other words, each peer makes 144 arrivals and departures per day. A study on peer availability in the P2P file-sharing system Overnet [BSV03] observed that the rates of long-term peer arrivals and departures do not exceed 40% of peers per day. On the other hand, each peer joins and leaves on average 6.4 times per day. When comparing with these results, it is clear that $\pi = 5\%$ indicates a very high dynamicity of peers.

The load balancing method takes effect in all four cases. However, higher π generate higher Ω. When a new peer p joins the system, it needs some time to establish a connection with the neighbourhood. During this time, the neighbours that have not yet received its notification, direct routing messages to other peers instead. Peer p does not bear enough routing traffic as it should and thus we observe a little increase of the global overload at the end of its arrival.

We also see that in systems with higher peer dynamicity, Ω decreases more slowly in phase 2 and increases more rapidly in phase 3. This phenomenon results from the insertion and deletion of peers, which break the load balance state previously established. The active load balancing procedure in phase 2 keeps reducing Ω. In phase 3, Ω does not increase only if $\pi = 0\%$. Otherwise, Ω rises due to the lack of rebalancing.

7.3.3 Storage load balancing evaluation

The principal metric for evaluating the storage load balancing method is the storage overload ratio $\Psi = (\sum_{\forall p} W_p)/(\sum_{\forall p} S_p)$. The smaller Ψ is, the more storage load balance

it achieves. The evaluation involves multiple experiments following different scenarios. All experiments start with 2048 peers. We limited the peers' storage capacity and desired storage capacity to the range [100MB, 3.2GB]. Again, we selected the distribution of the desired storage capacities of the peers under the Zipf law with the exponent $\varepsilon = -1.2$. As suggested by different research (e.g., [Dow01b, DB99]), we apply the log-normal distribution[9] to the object sizes, which fall within the range [1MB, 100MB]. The chosen parameters $\mu = 2$ and $\sigma = 0.84$ yield an object size mean of 10.5MB and a median of 7.4MB. We also have performed experiments with different distribution parameters (both for the desired storage capacity and object size) and even with the uniform distribution. They all gave performance results similar to the experiments presented herein.

We observed the performance of the storage load balancing method subject to three factors : the storage utilization ratio, the peer dynamicity, and the storage load transfer strategy applied (cost-oriented or overload-oriented). The storage utilization ratio Z is the ratio of the total storage load over the total desired storage capacity, $Z = (\sum_{\forall p} S_p)/(\sum_{\forall p} \overline{D}_p)$. Peer dynamicity ($\pi$) has the same meaning as for index management load balancing evaluation (Sect. 7.3.2). By comparing the impact of the storage load transfer strategies, we can observe their strength and weakness relative to each other. Different combinations of factors produce numerous cases. The graphs in this section plot the average results for at least 20 experiments in each case.

The experiments presented in Figure 7.17 continuously insert objects in the system to increase the utilization ratio Z. The graph plots the overload ratio Ψ corresponding to the increase of Z (from 1% to 150%). In order to confirm the effectiveness of load balancing, we compared the results under two modes : active load balancing and inactive load balancing. We also compared the impact of three peer dynamicity levels : $\pi = 0\%$, $\pi = 1\%$, and $\pi = 5\%$.

[9]The log-normal distribution of a variable x defines its density probability as $P(x) = e^{-(\ln x - \mu)^2/(2\sigma^2)}(x\sigma\sqrt{2\pi})^{-1}$ with given μ and σ parameters, which are respectively mean and standard deviation of $\log x$.

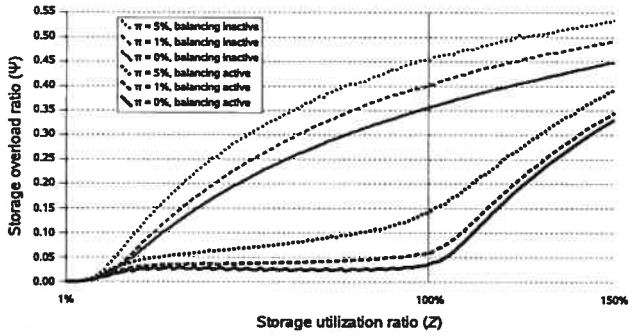

FIG. 7.17 – Storage overload ratio Ψ vs. storage utilization ratio Z for different peer dynamicity factors π (with cost-oriented balancing or without balancing)

These experiments apply the cost-oriented storage load transfer strategy. The graph shows that without load balancing, Ψ increases rapidly along with the increase of Z. However, with load balancing, the increase of Ψ is drastically slowed down as long as $Z \leq 100\%$. It rises very fast only when Z exceeds 100% (i.e., the system lacks available space to rearrange objects). We see that even with a high dynamicity ($\pi = 5\%$), the storage load balancing method is effective. Of course, higher dynamicity causes higher overload and reduces the storage load balancing effectiveness.

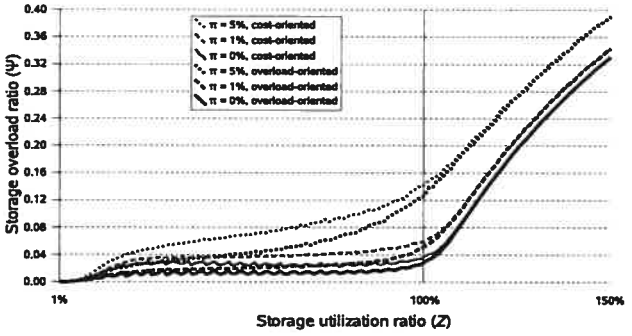

FIG. 7.18 – Storage overload ratio Ψ vs. storage utilization ratio Z for different peer dynamicity factors π (with cost-oriented balancing or overload-oriented balancing)

Figure 7.18 compares the effectiveness of the two storage load transfer strategies : cost-oriented and overload-oriented. The experiments were run with load balancing and with peer dynamicities π of 0%, 1%, and 5%, respectively. In all experiments, when π and $Z \leq 110\%$

are the same, the balancing with overload-oriented transfers produces lower Ψ than cost-oriented transfers. Overload-oriented transfers achieve higher effectiveness than cost-oriented transfers since they have more possibilities to reduce the combined overload. For $Z > 110\%$, the effectiveness of the two transfer strategies becomes the same since the possibility to rearrange objects is very small in this case.

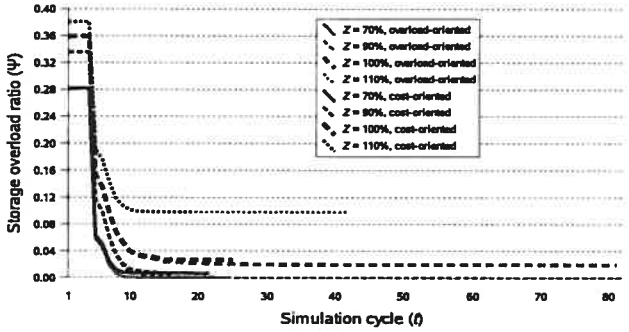

FIG. 7.19 – Storage overload ratio Ψ vs. simulation cycle t for different storage utilization ratios Z (with cost-oriented balancing or overload-oriented balancing ; $\pi = 0\%$)

In order to evaluate the convergence of the storage load balancing method, we executed experiments at a specific utilization ratio Z. The experiments ran with load balancing (using cost-oriented or overload-oriented strategy), $\pi = 0\%$, and without object insertion. This ensures that neither peer dynamicity nor storage load changes will affect the storage load balancing while verifying its convergence. The experiments stop when Ψ reaches 0 or its minimal value. Figure 7.19 shows the results of experiments with utilization ratios 70%, 90%, 100%, and 110%, respectively. In all experiments, the overload ratio Ψ decreases rapidly after a small time interval (being equal to the duration of a period of the overload verification on each peer). These results confirm that the load balancing method converges. We refer to the minimum value of Ψ as the stable storage overload ratio (denoted Ψ_f). Higher values of Z induce higher initial storage overload ratio and higher stable storage overload ratio. This is expected since a high Z corresponds to small available space and consequently to small possibilities of global overload reduction.

In all experiments using the same Z, the load balancing with overload-oriented transfers yields smaller Ψ_f than that for cost-oriented transfers. However, the time to achieve the stable state (called the stabilization time, denoted t_f) of the overload-oriented transfers is longer than that of the cost-oriented transfers. We find the details of these differences in Figures 7.20 and 7.21, which respectively compare the stable overload ratio and the stabilization time of the two transfer strategies. These figures additionally present the observation for $Z = 95\%$ and $Z = 105\%$. The differences follows from the fact that an overload-oriented storage load transfer among two peers has more chances to take place than a cost-oriented storage load transfer. Therefore, the load balancing that applies overload-oriented transfers involves more transfers. This results in a longer stabilization time but a better stable overload ratio.

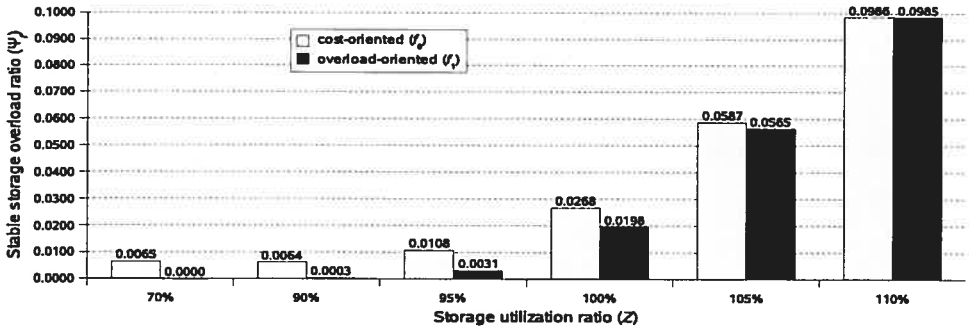

FIG. 7.20 – Stable storage overload ratio Ψ_f vs. storage utilisation ratio Z for cost-oriented and overload-oriented transfer strategies

The graph in Figure 7.21 shows that experiments at $Z = 100\%$ have the longest stabilization time. It is reduced when $Z < 100\%$ or $Z > 100\%$. When $Z > 100\%$, the lack of available space diminishes the chance for storage load transfers to take place. Therefore, the stable state is near the initial state. However, when $Z < 100\%$, the profusion of available space accelerates storage load transfers in reducing the global overload. It thus shortens the stabilization.

Let Ψ_{f_0} and Ψ_{f_1} be Ψ_f produced by the cost-oriented and overload-oriented transfer strategies, respectively. We statistically compared Ψ_{f_0} and Ψ_{f_1} under the given values of Z.

Z	Confidence interval of $\Delta\Psi_f = \Psi_{f_0} - \Psi_{f_1}$	Z	Confidence interval of $\Delta\Psi_f = \Psi_{f_0} - \Psi_{f_1}$
70%	$[0.0064, 0.0067]$	100%	$[0.0065, 0.0075]$
90%	$[0.0059, 0.0063]$	105%	$[0.0016, 0.0029]$
95%	$[0.0072, 0.0082]$	110%	$[-0.0003, 0.0005]$

TAB. 7.I – Comparison of stable storage overload ratio Ψ_f for cost-oriented (f_0) and overload-oriented (f_1) transfer strategies ($\alpha = 1\%$)

Table 7.I shows the confidence interval of $\Delta\Psi_f = \Psi_{f_0} - \Psi_{f_1}$. Except for the case $Z = 110\%$, the confidence interval of $\Delta\Psi_f$ does not include 0, which indicates a statistical difference between Ψ_{f_0} and Ψ_{f_1} and thus a considerable effect of the two transfer strategies on Ψ_f. At $Z = 110\%$, the confidence interval of $\Delta\Psi_f$ includes 0. This indicates a negligible difference of Ψ_f produced by the two transfer strategies. As explained before, in this case, the lack of available space prevents the possibility to arrange objects.

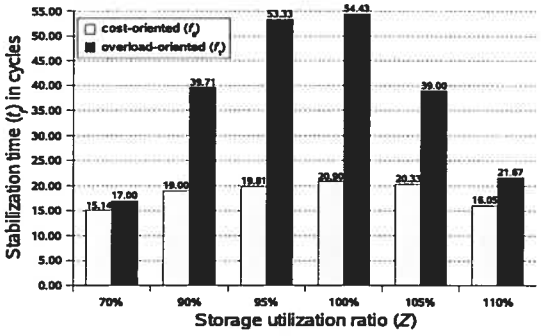

FIG. 7.21 – Stabilization time t_f vs. storage utilisation ratio Z for cost-oriented and overload-oriented transfer strategies

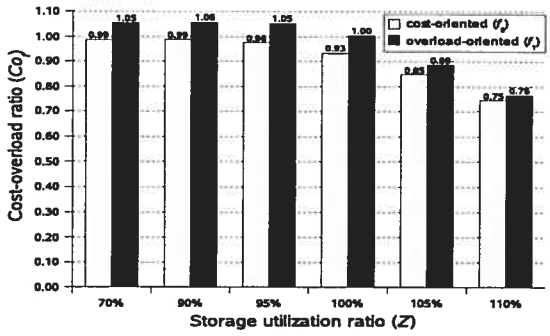

FIG. 7.22 – Cost-overload ratio Co_f vs. storage utilisation ratio Z for cost-oriented and overload-oriented transfer strategies

Similarly, we denoted the stabilization time obtained by the cost-oriented and overload-oriented transfer trategies as t_{f_0} and t_{f_1}, respectively. A statistical comparison of t_{f_0} and t_{f_1} allows us to evaluate the effect of the transfer strategies on the stabilization time. Table 7.II shows the confidence intervals of $\Delta t_f = t_{f_0} - t_{f_1}$.

Except for the case $Z = 70\%$, t_{f_0} and t_{f_1} are statistically different. This implies that the strategy has a considerable effect on the stabilization time. At $Z = 70\%$, although the

Z	Confidence interval of $\Delta t_f = t_{f_0} - t_{f_1}$	Z	Confidence interval of $\Delta t_f = t_{f_0} - t_{f_1}$
70%	$[-3.83, 0.12]$	100%	$[-38.63, -28.42]$
90%	$[-30.92, -10.51]$	105%	$[-24.01, -13.32]$
95%	$[-40.92, -26.13]$	110%	$[-8.90, -2.34]$

TAB. 7.II – Comparison of stabilization time t_f for cost-oriented (f_0) and overload-oriented (f_1) transfer strategies ($\alpha = 1\%$)

mean of t_{f_0} is a bit lower than the mean of t_{f_1}, the confidence interval of their difference covers 0. The difference between t_{f_0} and t_{f_1} in this case is not clear. The explanation of this phenomenon comes from the profusion of available space that diminishes the number of 2-direction storage load transfers in the overload-oriented transfer mode and thereby induces a weak effect of the chosen mode on the stabilization time.

Our evaluation also examined the cost of balancing, that is, the volume of data for migration. We use the experiments in the above convergence-evaluation scenario for this evaluation. The metric is the cost-overload ratio Co_f being the ratio of the cumulative storage load sent (until the stable state) over the initial global overload. Figure 7.22 shows the average cost-overload ratio for the experiments. At the same Z, an experiment with cost-oriented transfers always has the cost-overload ratio under 1 and lower than the cost-overload ratio of an experiment with overload-oriented transfers. The reason is that a cost-oriented transfer reduces the combined overload the most while preventing the cost to exceed the gain. On the other hand, an overload-oriented transfer minimizes the combined overload before minimizing the migration cost. The graph also shows that the higher Z is, the less is the cost-overload ratio. This results from the fact that smaller available space yields smaller storage load movements. Therefore, migration costs for stabilization are smaller.

We also statistically compared the cost-overload ratio across the two transfer strategies. Let Co_{f_0} and Co_{f_1} denote the cost-overload ratio under the cost-oriented and the overload-oriented transfer modes, respectively. Table 7.III lists the confidence interval of $\Delta Co_f =$

$Co_{f_0} - Co_{f_1}$ at different utilization ratios Z. No interval includes 0. This shows a significant effect of the transfer trategies on the cost-overload ratio.

Z	Confidence interval of $\Delta Co_f = Co_{f_0} - Co_{f_1}$	Z	Confidence interval of $\Delta Co_f = Co_{f_0} - Co_{f_1}$
70%	$[-0.0695, -0.0660]$	100%	$[-0.0740, -0.0694]$
90%	$[-0.0690, -0.0652]$	105%	$[-0.0417, -0.0354]$
95%	$[-0.0791, -0.0731]$	110%	$[-0.0208, -0.0172]$

TAB. 7.III – Comparison of cost-overload ratio Co_f for cost-oriented (f_0) and overload-oriented (f_1) transfer strategies ($\alpha = 1\%$)

7.3.4 Integrated load balancing

To this point, index management load balancing and storage load balancing methods have been evaluated separately. One can question whether these load balancing methods preserve their performance when operating concurrently. In other words, are index management load balancing and storage load balancing affecting each other ? We designed experiments to assess this influence.

The experiments ran on a system of 2048 peers with peer dynamicity $\pi = 0\%$ and routing target dynamicity $\tau = 0\%$. The distribution of the desired storage capacity, the object size, the index management capacity, the probability of routing source, and the probability of routing target are the same as the experiments presented in Sections 7.3.2 and 7.3.3. The metrics for the load balancing performance are again the index management overload ratio Ω and the storage overload ratio Ψ. These metrics are verified in the following cases :

- (00) inactive index management load balancing, inactive storage load balancing,
- (01) inactive index management load balancing, active storage load balancing,
- (10) active index management load balancing, inactive storage load balancing,
- (11) active index management load balancing, active storage load balancing.

In those cases where the storage load balancing is active, we employ the cost-oriented

transfer strategy. To confirm the performance of the index management load balancing in a large range of utilization ratio, we realized experiments at two levels : $U \in [55\%, 65\%]$ and $U \in [100\%, 110\%]$. The graphs in this section plot the average results for at least 20 experiments within each case.

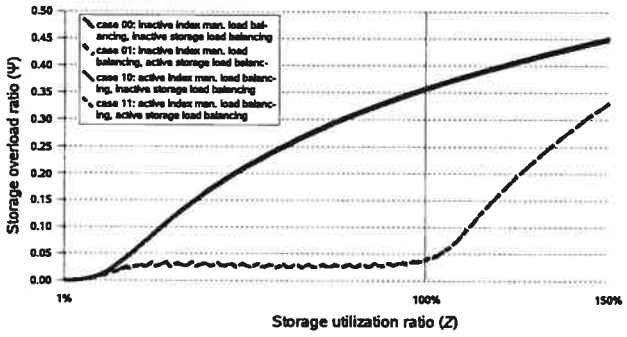

FIG. 7.23 – Storage overload ratio Ψ vs. storage utilization ratio Z for different balancing cases ($U \in [55\%, 65\%]$, $\pi = 0\%$, $\tau = 0\%$)

Figure 7.23 compares the increase of the storage overload ratio Ψ (along with the increase of the storage utilization ratio Z from 1% to 150%) in the four cases. The index management utilization ratio U is set to the range $[55\%, 65\%]$. The increase of Ψ in the two cases 00 and 10 is similar. In the two other cases, the presence of the storage load balancing prevents the increase of Ψ and gives the same results regardless whether the index management load balancing is active or not.

We performed statistical comparisons (as described at the beginning of Sect. 7.3) between the graphs of cases 00 and 10, and between those of cases 01 and 11. The difference rate obtained from the former comparison is 78.00% and from the latter it is 24.67%. These rates show a statistical difference in each compared graph pair. However, the difference is very small. In each comparison between two graphs X and Y, the average difference $|X_i - Y_i|$ obtained never exceeds 1% of the graphs' scale (for all values i on the horizontal axis). The scale of a graph here denotes the graph's maximal value.

The above differences result from the measurement method. Index management load

balancing consists in transferring key intervals among peers. A transfer not only moves the key interval but also the replica diffusion processes of the objets involved in the key interval moved (see Sect 7.2.3). To ensure the integrity of the decentralized operation, the replica diffusion processes on the source peer are not stopped until they receive an acceptance message from the destination peer. Therefore, between the times of sending a proposal and receiving an acceptance, replica diffusion processes (for one object) may function concurrently on both peers. This situation accelerates object insertion. Because the storage utilization ratio considered is a real number, we cannot measure the overload corresponding to exactly one value point on the horizontal axis. However, we determined it by calculating the average of the measured results for each interval of length equal to 1% on the storage utilization ratio axis. Therefore, the acceleration of object insertion can affect the results measured with some probability. The small difference given by the statistical analyses reflects the effect of the object insertion acceleration on the measurement. We cannot, however, conclude that there is an impact of the index management load balancing on the effectiveness of the storage load balancing.

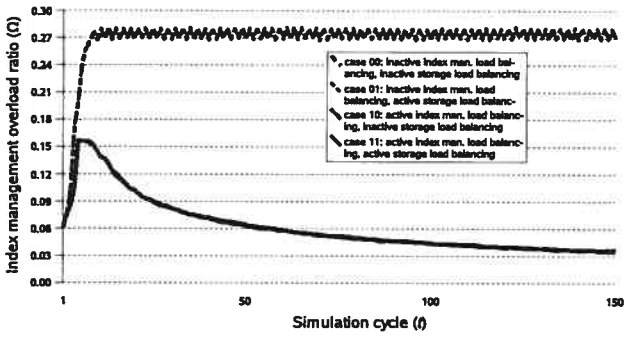

FIG. 7.24 – Index management overload ratio Ω vs. simulation cycle t for different balancing cases ($U \in [55\%, 65\%]$, $\pi = 0\%$, $\tau = 0\%$)

In Figure 7.24, we compare the index management overload ratio Ω (for 150 simulation cycles) in the four cases. The utilization ratio U is chosen in the range $[55\%, 65\%]$. The variation of Ω is the same among the cases 00 and 01, and among the cases 10 and 11. We

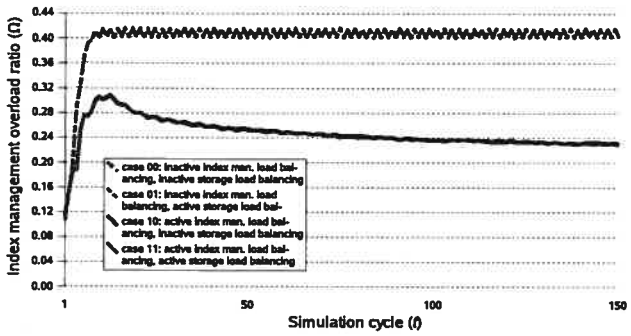

FIG. 7.25 – Index management overload ratio Ω vs. simulation cycle t for different balancing cases ($U \in [100\%, 110\%]$, $\pi = 0\%$, $\tau = 0\%$)

see that the presence of the storage load balancing does not degrade the result of the index management load balancing. Figure 7.25 shows similar experiments but at utilization ratio $U \in [100\%, 110\%]$. We observe similar results except that the Ω values obtained are higher. Statistical comparisons between the graphs of cases 00 and 01, and between the graphs of cases 10 and 11 (in both Figures 7.24 and 7.25) were also performed. The comparisons of graphs in Figure 7.24 give difference rates as low as 0.67% and 0.00%, respectively. The corresponding rates for the graphs in Figure 7.25 are 0.00% and 0.00%. The results demonstrate that the storage load balancing does not influence the effectiveness of the index management load balancing.

7.4 Validation

The performance of the proposed P2P topology and the load balancing methods have been evaluated. In this section, we validate the above results by demonstrating the advantages of the technical solutions chosen. We consider two points : the separation between the object location and the key, and index management load balancing based on key transfer. The demonstration compares our approach with some related work. We have implemented simulators inspired by these related models to run experiments for comparison purposes. These

simulators only implement the elements and functions required to perform comparisons with our approach. They therefore maximize simplicity to preserve efficiency and to guaranty conformance to the methods' specifications.

7.4.1 Separation vs. attachment of the storage location and the key

To validate the separation between the storage location and the key, we investigate how the dependency of the object location on the key influences the integrated index management load and storage load balancing. An example of the attachement of the object location to the key is the application of virtual servers in the storage load balancing. According to this approach, each physical peer maintains multiple virtual servers, which operate as individual peers in a normal system. The peers can achieve load balancing by arranging the virtual servers' residence. Because a virtual server represents a virtual peer, it is responsible for a set of keys. In the application of virtual servers for storage load balancing, the objects' location must be tied to their key.

In this discussion, we do not compare BALLS with the Virtual Servers system [RLS+03] in the general context as it was introduced. We simply evaluate the ability to apply the notion of virtual servers for simultaneously supporting the storage load balancing and the index management load balancing. Thus, we compare with the application of virtual servers for integrated load balancing. We developed a P2P simulator that creates virtual servers on top of physical peers. In simulations, it lets each virtual server manage one key, the smallest size of a virtual server. The simulator implements an index management load balancing similar to ours. The support of this index management load balancing adds a constraint to the virtual servers : each physical peer manages only numerically adjacent virtual servers, i.e., virtual servers on a peer compose a continuous key interval. Therefore, the transfer of virtual servers is limited among ring neighbours.

The storage load balancing is based on transferring virtual servers (the virtual servers carry all objects that belong to them). The simulator implements only the one-to-one transfer. That is, when a peer discovers the overload, it transfers an appropriate set of virtual servers to a ring neighbour for achieving the balancing goal. The use of this straightforward operation ensures the simulator's simplicity. The calculation of the global overloads, loads, and capacities simply aggregate the overloads, loads, and capacities, respectively, of all the peers, which are measured in a similar way as BALLS, thereby allowing to compare the simulation results.

The experiments ran in four cases (like the experiments presented in Sect. 7.3.4) :

- (00) inactive index management load balancing, inactive storage load balancing,
- (01) inactive index management load balancing, active storage load balancing,
- (10) active index management load balancing, inactive storage load balancing,
- (11) active index management load balancing, active storage load balancing.

The system comprises 2048 physical peers with dynamicity $\pi = 0\%$ and routing target dynamicity $\tau = 0\%$. The parameter settings for the peer's capacities, object size, and routing skewness are the same as those used for the experiments described in Section 7.3.4. The index management utilization ratio U is set in the range $[55\%, 65\%]$. The storage utilization ratio Z increases along with object insertions. To enable the comparison, we used the storage overload ratio Ψ and the index management overload ratio Ω as metrics to measure the effectiveness of the storage load balancing and the index management load balancing, respectively. The following graphs depict the average results of at least 20 experiments in each case.

Figure 7.26 compares the storage overload ratio Ψ (for the storage utilization ratio Z increasing from 1% to 150%) in the four cases. The storage load balancing still takes effect. However, the presence of the index management load balancing produces a higher Ψ. We have realized statistical comparisons between the graphs of cases 00 and 10, and between the

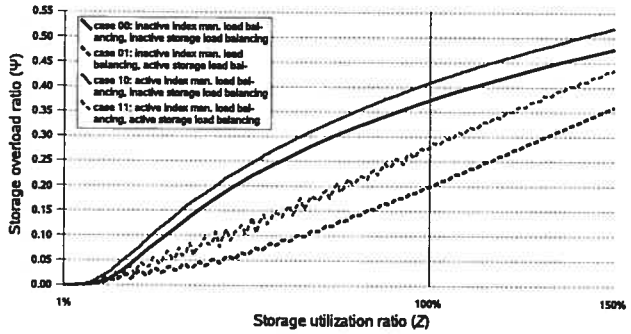

FIG. 7.26 – Storage overload ratio Ψ vs. storage utilization ratio Z for different balancing cases using the virtual servers approach ($U \in [55\%, 65\%]$, $\pi = 0\%$, $\tau = 0\%$)

graphs of cases 01 and 11, which give difference rates as high as 96.67%. These results show a considerable effect of the index management load balancing on the storage load distribution. The transfer of key intervals in the index management load balancing involves transferring the corresponding objects as well. It therefore breaks the current distribution of the storage load. The index management load balancing considerably reduces the effectiveness of the storage load balancing if it is active.

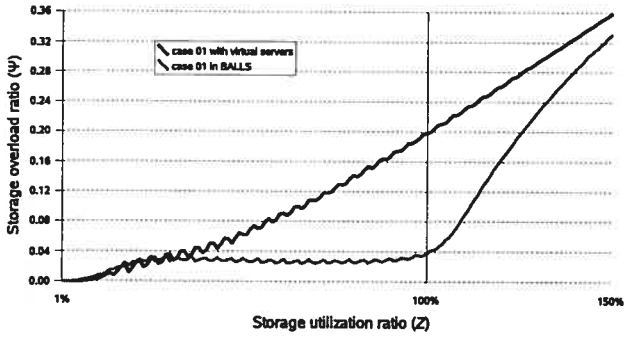

FIG. 7.27 – Storage overload ratio Ψ vs. storage utilization ratio Z for the virtual servers approach and BALLS (case 01, storage balancing only ; $U \in [55\%, 65\%]$, $\pi = 0\%$, $\tau = 0\%$)

We also compared the result of the storage load balancing (without index management load balancing) to that of our system (Fig. 7.27). Our approach produces a much better result even when using the cost-oriented strategy. This difference comes from the fact that we balance the storage load at the object level while the arragement of virtual servers does it at

the key level. The manipulation at the key level is less flexible because all objects belonging to a key must move together. However, if we arrange the storage load at the object level, we can select individual objects to move. Moreover, better performances are obtained because our peers can freely choose objects and destination peers to transfer storage load without worrying about the maintenance of the peer connection structure (as in moving keys).

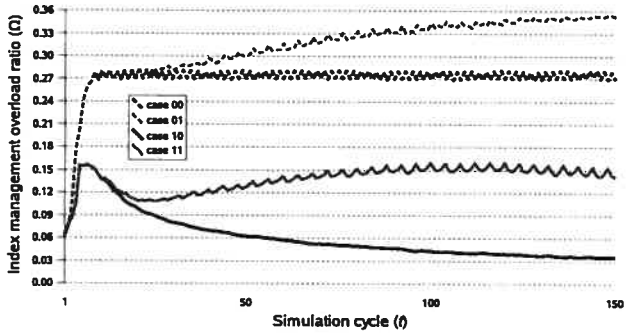

FIG. 7.28 – Index management overload ratio Ω vs. simulation cycle t for different balancing cases using the virtual server approach ($U \in [55\%, 65\%]$, $\pi = 0\%$, $\tau = 0\%$)

We also investigated the impact of the storage load balancing on the index management load balancing. Figure 7.28 compares the index management overload ratio Ω (for 150 simulation cycles) in the four cases. Again, the index management load balancing takes effect. However, it is considerably affected by the simultaneous storage load balancing. Statistical comparisons between the graphs of cases 00 and 01, and between the graphs of cases 10 and 11, reinforce this observation. We indeed obtain high difference rates at 83.33% and 89.33%, respectively. Moving virtual servers in the storage load balancing breaks the current distribution of the index management load on the peers and often produces higher Ω. If the index management load balancing operates, its effectiveness is greatly reduced.

The comparison of the above experiments with the corresponding experiments on our system (Sect. 7.3.4) exposes two major disadvantages of associating the object and key locations :

– the index management load balancing and the storage load balancing are no longer in-

dependent. They reduce the performance of each other when being executed concurrently ;

- constraining object location to the key location prevents the choice of objects and destination peers in storage load migration. This constraint greatly reduces the effectiveness of the storage load balancing even without the intervention of the index management load balancing.

7.4.2 Disadvantage of restricting object location

As discussed in Section 7.1.2, some systems that break the tie between object storage and root still maintain a restriction on object placement. This introduces considerable network overhead when they change. We validate this discussion by evaluating this type of overhead in PAST [RD01b], which is a representative example of such systems. PAST uses a replica diversion technique so as to balance the storage load. This technique allows a file (PAST uses files as storage objects) to stay on one of the peers in the leaf set of its root instead of being attached to the root. Because of this relaxation, PAST reduces file insertion failures and thus improves storage capability. The replica diversion of PAST does not really separate the file location from the file's root. PAST maintains the following invariant : a file is stored inside the leaf set of the root. Suppose that l is the leaf set size. The arrival of a peer changes l peers' leaf set boundaries. It thus pushes some files out of their roots' leaf sets. To enforce the above invariant, PAST moves these files to the corresponding leaf sets. This invariant thus introduces additional maintenance costs.

We evaluated the file migration costs caused by peer arrival in such a system through experiments. To that end, we developed a simulator that constructs a simple P2P system, in which each peer p maintains a leaf set containing the contacts of l peers having ids numerically closest to p's id. Each file is replicated and assigned to κ roots being the peers with ids

numerically closest to the m-bit prefix of the file's id (m is the length of the peer id). The location of a replica is limited to its root and the leaf set of the root. The maintenance of this invariant against the system's evolution is straightforward. It discovers the files that are out of their limit and moves them to a valid host. The measurement of the migration costs incurred by a peer arrival consists in adding the sizes of all replicas moved to maintain the above invariant. To ensure correctness and efficiency of the measurements, this simulator performs calculations in a centralized manner.

The simulation sets the peer population to 2250, the peer id length (m) to 128 bits, and the number of replicas of a file (κ) to 5. The distribution of the peers' storage capacity is the same as that in experiments described by [RD01b] : normal distribution with $\sigma = 10.8$, $\mu = 27$MB, lower bound = 2MB, and upper bound = 51MB. Because we cannot obtain the set of files used in [RD01b], we generated file sizes using a log-normal distribution with min = 1KB, max = 10MB, median = 500KB, and mean = 1MB. In the context of experiments that verify the migration costs of PAST's peer arrivals, we do not use the experimental results presented by [RD01b]. Thus, the application of its exact settings is not necessary.

The experiments let peers arrive and depart with the same probability to maintain a relatively stable number of peers and storage capacity. We consider two metrics : (1) the load move ratio that is the ratio of the storage load moved (caused by one peer arrival) over the total storage load, and (2) the file move ratio that is the ratio of the number of moved files (caused by one peer arrival) over the total number of files. The experiments measure the cumulative load move ratio Λ (i.e., the sum of all load move ratios for all arrivals) and the cumulative file move ratio Φ (i.e., the sum of all file move ratios for all arrivals) for 500 arrivals. We obtained results at the following utilization ratio (i.e., the ratio of the total load over the total capacity) Z : 10%, 30%, 50%, 70%, and 90%. Two leaf set sizes (16 and 32) have also been tried. The graphs in Figures 7.29 and 7.30 plot the average results for at least 20 experiments in each case.

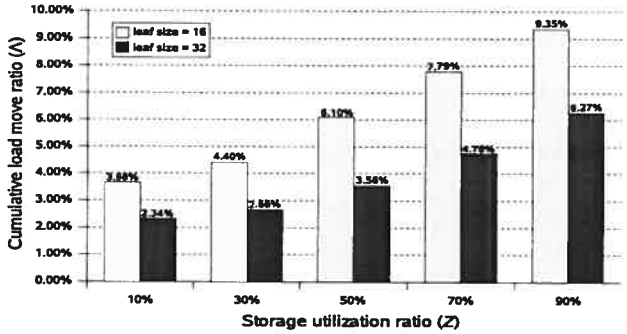

FIG. 7.29 – Cumulative load move ratio Λ vs. storage utilization ratio Z for 500 arrivals in PAST

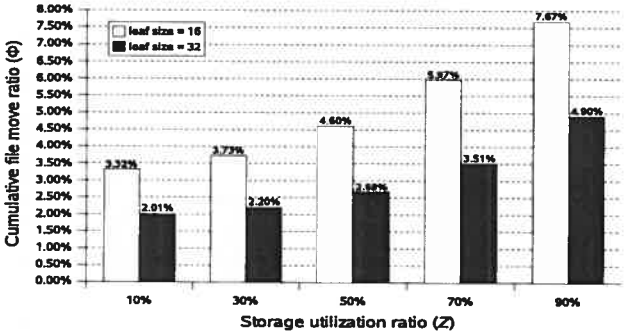

FIG. 7.30 – Cumulative file move ratio Φ vs. storage utilization ratio Z for 500 arrivals in PAST

A smaller leaf size leads to higher Λ and Φ. This is because the smaller leaf size increases the probability that a file is out of the root's leaf set when a peer arrives. We see that higher utilization ratios tend to produce higher migration costs. In all cases, the migration cost is considerable in comparing with the total load and the total number of files (Λ ranges from 2.34% to 9.35% and Φ ranges from 2.01% to 7.67%).

In order to reduce migration costs, PAST must use an additional caching mechanism. Note that in BALLS, peer arrival never incurs object migration costs as it completely separates object location and object key. In fact, it only requires a number of root notification messages equivalent to the number of objects whose keys fall within the key interval moved during peer arrival. In storage systems where object sizes are large enough, the communication costs incurred by the root notification messages are very small compared to the migration

costs.

In a highly loaded network, adding a peer (with low storage load) usually attracts objects from other peers to reduce their overload. We do not measure the cost of this tranfer as the migration cost for peer arrival since it does not serve any structure invariant maintenance. It is measured, however, in the migration cost for load balancing.

Note that the difference in the load balancing goal between PAST and BALLS does not allow us to compare them on their effectiveness. While PAST aims at maximizing the use of free space to decrease file insertion failures, BALLS minimizes the global overload based on the desired storage capacity of the peers.

7.4.3 Validation of the index management load balancing method

Since each peer in our system maintains an interval of de Bruijn nodes (i.e., keys) and the routing algorithm follows routing paths in the de Bruijn graph, transferring key intervals among peers allows to adjust the distribution of the index management load towards a balanced state. The use of the de Bruijn graph ensures a low peer degree (near 8) regardless of system size. The construction of the network makes the connection between peers symmetric. It means that if a peer p has a link to a peer q, then peer q is linked to peer p. These features enable low-cost maintenance procedures and thus efficient index management load balancing.

Expressways [ZSZ02] also considers index management load on the peers. The load balancing method of Expressways is based on the reorganization of links between the peers. As discussed in Section 7.1.2, this method yields a high peer degree. We evaluate this feature by experiments.

Expressways extends the idea of CAN [RFH+01] by managing a hierarchy of zone spans

in which the leaves are basic CAN zones and each parent (internal) zone covers all its child zones. Because of the hierarchical management, the links of internal zones (also called expressway zones) tend to incur more routing traffic than those of their child zones. The Expressways load balancing method promotes peers with higher capacities to higher expressway levels so as to equalize the peers' load/capacity ratio.

The above load balancing goal is different from that of BALLS, which minimizes the global overload when the overload occurs. Again, this difference does not allow us to use the balancing effectiveness as the comparison criterion. However, we can compare BALLS and Expressways on the costs of the load balancing methods. In particular, these costs are function of the peer degree in each system.

The index management load balancing method of Expressways can only be applied in a P2P structure that manages a hierarchy of zone spans. In a basic CAN system, the peer degree is constant. In Expressways, multiple zone levels require multiple link levels. If a system has n peers, it requires a hierarchy of $O(\log n)$ zone levels and thus a peer degree in $O(\log n)$. We evaluated this assessment through experiments.

The simulator implements a simple prototype of the Expressways system, which is based on hierarchical management of zone spans. Each peer maintains the links to its neighbouring zone spans at all levels of the hierarchy. The peer arrival and departure consist of, respectively, splitting and merging zone spans and updating the links of peers to maintain the Expressways topology. A straightforward measurement of peer degree that counts the number of links at all levels on each peer enables us to obtain comparable results.

Our Expressways simulation sets the number of dimensions at 2, the coverage at 4, and the largest zone span at the entire Cartesian space. An experiment starts with 1 peer with an arrival probability higher than the departure probability. It stops when the system reaches 2100 peers.

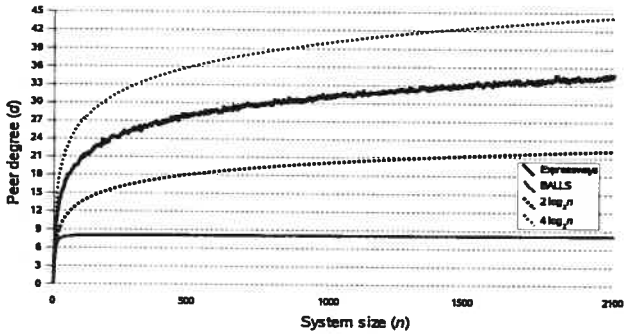

FIG. 7.31 – Peer degree d vs. system size n in Expressways

Figure 7.31 plots the average result for 40 experiments. It compares the average peer degree of Expresssways with that of BALLS. The experiments show that the measured average peer degree of Expressways falls within $2\log_2 n$ and $4\log_2 n$ (where n is the current system size). Confidence intervals were calculated for the average peer degree at all system sizes from 1 to 2100. The distance between the boundaries of these intervals and the measured average peer degree has max = 0.55, mean = 0.38, and median = 0.38. These numbers show very close confidence intervals. We see that the peer degree of Expressways is considerably higher than the constant peer degree (near 8) measured in BALLS. Thus the connection structure of Expressways tends to induce a higher complexity for maintenance.

In the current implementation of our system, we send a root notification message for every object with the same key. However, a simple improvement could greatly reduce this cost. We can notify all objects stored in the same peer about their new root using a single message, hence reducing the number of messages required.

7.5 Discussion

We have proposed and evaluated the performance of the BALLS topology and integrated load balancing methods. We have also shown the advantages of these methods by comparing

them with some related methods. From the results obtained, we have the following comments.

Recall that our work aims at two goals : (1) simultaneously supporting index management load balancing and storage load balancing, and (2) providing low maintenance costs. We now review the fulfilment of these requirements.

As described, simultaneous index management load balancing and storage load balancing is achieved by separating object location and key location. It allows the storage load balancing algorithm to arrange objects regardless of the key distribution. Therefore, the storage load balancing and the index management load balancing do not affect each other. The evaluation of integrated load balancing in BALLS and in the application of virtual servers (Sect. 7.3.4 and 7.4.1) confirmed this conclusion. The experiments using BALLS, where the key and object locations are independent, showed that the effectiveness of the index management load balancing is not affected by the storage load balancing and vice versa. However, if the system ties the object location to the key location (directly or indirectly), index management and storage load balancing are no longer independent, reducing the effectiveness of each other.

We also have seen that storage load balancing in a system having independent object and key locations is more effective than in a system without this independence. If the objects are tied to their key, each storage load move involves moving at least one key and all objects belonging to the key. Moreover, to maintain the P2P connection structure, only a certain set of keys on a peer can be selected for moving and only a certain set of neighbouring peers can receive the transferred keys. For example, in experiments with virtual servers in Section 7.4.1, a peer p can only move key intervals at two ends of $[p.b, p.e]$ to its ring neighbours. However, if we separate the object and key locations, the peers are free to choose objects and destination peers for the move. The peers have more possibilities to make storage load moves and thus induce a higher effectiveness of storage load balancing.

Let us now consider the cost of maintenance, which is usually defined as the bandwidth spent to maintain the P2P structure consistency against the network dynamicity. In BALLS, the dynamicity that affects the P2P structure is made of the peer arrivals/departures and the index management load balancing, which involve a key interval transfer.

In a traditional system where the object location is tied to the key, the transfer of a key always requires to move the objects belonging to it. The key interval transfer cost thus includes the object migration cost. In BALLS, a key interval transfer never needs to move objects. Instead, it only moves the associated storage pointers and launches the necessary root notification messages. In general, an object whose key falls within the transferred interval needs one pointer to move and one notification message to be sent. If we assume that the size of a pointer and the size of a notification message are much smaller than the size of an object, BALLS greatly saves on the transfer cost.

Although small, notification messages induce communication costs for object management in BALLS. These costs depend on the distribution of objects over the key space. The evaluation of these costs is beyond the scope of this paper.

We noticed in Section 7.4.2 that PAST relaxes the storage constraint by using redirection pointers and the replica diversion technique. Yet, its peer arrival produces a considerable migration cost. This follows from the fact that the file location and the key are not really separated.

When considering peer departure, the separation of object location and key location also have advantages. Since it allows object replication, the departure of a peer needs only to copy the objects that are not replicated elsewhere. Replicas of the other objects can be re-established later. This saves on the time of the departure procedure. Replication is also useful in storage load balancing. It distributes the access of an object on multiple peers so as to minimize bandwidth for "hot" content.

The maintenance of object availability is a common problem of P2P systems. Indeed, we should ensure that an object will still exist even when the peer storing it leaves the network. The difficulty in availability maintenance increases if more peers depart than peers arrive, i.e., the global storage capacity decreases. It also depends on the selection of storage capacity of the peers. This paper does not address this problem.

BALLS also achieves low maintenance costs by virtue of its low peer degree. A low peer degree simplifies the maintenance against the change of key responsibility. Experiments in Section 7.3.1 showed that BALLS's average peer degree is near 8, regardless of system size. This peer degree is considerably small in comparison to logarithmic peer degree systems such as Chord [SMK+01] and Tapestry [ZHS+04].

We have compared the peer degree of BALLS with another P2P system that also supports index management load balancing, Expressways, in Section 7.4.3. The experiments showed that the average peer degree of Expressways falls within $2\log_2 n$ and $4\log_2 n$ (at an n-peer system size). This degree is obviously higher than the constant peer degree of BALLS.

Finally, unlike other load balancing methods that aim at globally equalizing the load or the load/capacity ratio of the peers, our methods minimize the global overload (for both index management load and storage load). In practice, as long as the capacity of a peer covers the load, the peer can work properly and does not need any operation to reduce its load. Therefore, a load balancing action in this situation is not necessary. A peer provokes the load balancing operation only when its load exceeds the capacity and generates a positive overload.

The minimization of the global overload rearranges load from overloaded peers to other peers as long as sufficient capacity is available. It thus pays the least cost to reach the balanced state. So, relying on the minimization of the global overload allows us to save on the cost of rebalancing.

7.6 Conclusion

This paper has described BALLS, a structured P2P overlay that allows for simultaneous index management load balancing and storage load balancing. This system achieves integrated load balancing due to the separation among object location and object key location. This separation also saves on maintenance costs against peer dynamicity.

Experiments have evaluated and confirmed the effectiveness of the proposed index management load balancing and storage load balancing methods. We also compared the storage load balancing results that are obtained from two storage load transfer strategies : cost-oriented and overload-oriented. The former ensures that the transfer cost is limited by the overload reduction but reduces the effectiveness.

Our evaluation confirmed that the proposed index management load balancing and storage load balancing methods can simultaneously operate without reducing the effect of each other. Experiments of the load balancing methods on a P2P system that associates the objects in their key location showed that this association breaks the independence of the load balancing methods and degrades their effectiveness. We also compared BALLS with other structured P2P systems on the cost of maintenance. The comparison confirmed that BALLS reduces maintenance costs.

An evaluation of BALLS in a real network is planned. The fault-tolerance improvement and the evaluation of communication costs to maintain object consistency also need to be considered. These will allow us to produce and deploy an applicable P2P system.

7.A Proof of Theorem 1

Proof : Suppose that a peer p ($A_p < 0$) makes an 1-direction storage load transfer to a peer q ($A_q > 0$). Their overload before and after the transfer are, respectively :

$$\begin{aligned} W_p &= (|A_p| - A_p)/2 = -A_p \\ W_q &= (|A_q| - A_q)/2 = 0 \\ W'_p &= (|A_p + S_{pq}| - A_p - S_{pq})/2 \\ W'_q &= (|A_q - S_{pq}| - A_q + S_{pq})/2 \end{aligned}$$

The change to the combined overload of p and q is :

$$\begin{aligned} \Delta W &= W'_p + W'_q - W_p - W_q \\ &= (|A_p + S_{pq}| + |A_q - S_{pq}| + A_p - A_q)/2 \end{aligned} \tag{7.10}$$

We determine S_{pq} for the following cases :

1. if $S_{pq} \leq -A_p$,

   (a) if $S_{pq} \leq A_q$, (7.10)$\Rightarrow \Delta W = -S_{pq}$.

   Because $0 < S_{pq} \leq \min(-A_p, A_q)$, we have $0 > \Delta W \geq \max(A_p, -A_q)$.

   (b) if $S_{pq} > A_q$, (7.10)$\Rightarrow \Delta W = -A_q$.

   Because $A_q < S_{pq} \leq -A_p$, we have $-A_q > A_p$ and $\Delta W = \max(A_p, -A_q)$.

2. if $S_{pq} > -A_p$,

   (a) if $S_{pq} \leq A_q$, (7.10)$\Rightarrow \Delta W = A_p$.

   Because $A_q \geq S_{pq} > -A_p$, we have $-A_q < A_p$ and $\Delta W = \max(A_p, -A_q)$.

(b) if $S_{pq} > A_q$, (7.10)$\Rightarrow \Delta W = A_p - A_q + S_{pq}$.

$\Delta W < 0$ only if $S_{pq} < -A_p + A_q$.

The following graph shows the dependency of ΔW on S_{pq}.

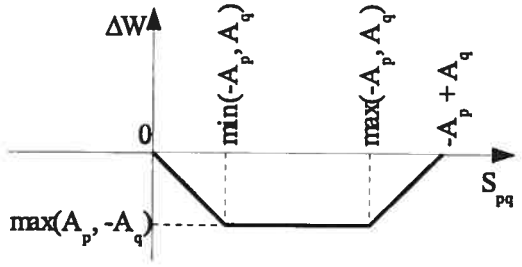

FIG. 7.32 – $\Delta W(S_{pq})$ vs. S_{pq} in the 1-direction storage load transfer mode

By definition, the optimal 1-direction storage load transfer requires :

$$\left.\begin{array}{l}\text{first, } \Delta W \text{ is the most negative} \\ \text{second, } S_{pq} \text{ is the smallest possible}\end{array}\right\} \qquad (7.11)$$

The variation of ΔW shows that only S_{pq} chosen the closest to $\min(-A_p, A_q)$ such that $S_{pq} < -A_p + A_q$ satisfies (7.11).

Due to its definition, the bounded optimal 1-direction storage load transfer restricts S_{pq} not higher than the reduction of $W_p + W_q$. From the ΔW graph, S_{pq} does not exceed the combined overload reduction (being $-\Delta W$) only when $S_{pq} \leq \min(-A_p, A_q)$. In order to achieve the most negative ΔW while ensuring the above restriction, S_{pq} must be the greatest but not higher than $\min(-A_p, A_q)$.

□

7.B Proof of Theorem 2

Proof : Let W_p, W_q, W'_p, and W'_q denote the overload of p and q, before and after the transfer, respectively.

$$\begin{aligned} W_p &= (|A_p| - A_p)/2 = -A_p \\ W_q &= (|A_q| - A_q)/2 = 0 \\ W'_p &= (|A_p + S_{pq} - S_{qp}| - A_p - S_{pq} + S_{qp})/2 \\ W'_q &= (|A_q - S_{pq} + S_{qp}| - A_q + S_{pq} - S_{qp})/2 \end{aligned}$$

The change to the combined overload of p and q after the transfer is :

$$\begin{aligned} \Delta W &= W'_p + W'_q - W_p - W_q \\ &= (A_p - A_q + |A_p + S_{pq} - S_{qp}| + |A_q - S_{pq} + S_{qp}|)/2 \end{aligned} \tag{7.12}$$

Given A_p, A_q, and S_{pq}, ΔW is a function of S_{qp}. For achieving the optimal 2-direction storage load transfer, S_{qp} must satisfy (7.13).

$$\left.\begin{array}{l} \text{first, } \Delta W \text{ is the most negative} \\ \text{second, } S_{qp} \text{ is the smallest possible} \end{array}\right\} \tag{7.13}$$

We determine S_{qp} (respecting $0 \leq S_{qp} < S_{pq}$) for the following cases :

1. if $S_{pq} \leq A_q$,

   (a) if $S_{pq} \leq -A_p$, (7.12)$\Rightarrow \Delta W = -S_{pq} + S_{qp} < 0$. From (7.13), we choose $S_{qp} = 0$.

   (b) if $S_{pq} > -A_p$,

      i. if $S_{qp} \geq A_p + S_{pq} > 0$, (7.12)$\Rightarrow \Delta W = -S_{pq} + S_{qp} \Rightarrow A_p \leq \Delta W < 0$.

ii. if $0 \leq S_{qp} < A_p + S_{pq}$, (7.12)$\Rightarrow \Delta W = A_p < 0$.

In case 1(b)ii, both ΔW and S_{qp} are respectively less than those in case 1(b)i. Thus, from (7.13), we choose $S_{qp} = 0$, which falls within case 1(b)ii.

In case 1 ($S_{pq} \leq A_q$), $S_{qp} = 0$.

2. if $S_{pq} > A_q$,

(a) if $S_{pq} \leq -A_p$,

i. if $S_{qp} \geq -A_q + S_{pq} > 0$, (7.12)$\Rightarrow \Delta W = -S_{pq} + S_{qp} \Rightarrow -A_q \leq \Delta W < 0$.

ii. if $0 \leq S_{qp} < -A_q + S_{pq}$, (7.12)$\Rightarrow \Delta W = -A_q < 0$.

In case 2(a)ii, both ΔW and S_{qp} are respectively less than those in case 2(a)i. Thus, from (7.13), we choose $S_{qp} = 0$, which falls within case 2(a)ii.

In case 2a ($A_q < S_{pq} \leq -A_p$), $S_{qp} = 0$.

(b) if $S_{pq} > -A_p$,

i. if $S_{qp} \geq A_p + S_{pq}$,

A. if $S_{qp} \geq -A_q + S_{pq}$, (7.12)$\Rightarrow \Delta W = -S_{pq} + S_{qp} < 0$. Because $S_{qp} \geq \max(A_p, -A_q) + S_{pq}$, we have $\Delta W \geq \max(A_p, -A_q)$. If S_{qp} increases, ΔW increases as well.

B. if $S_{qp} < -A_q + S_{pq}$, (7.12)$\Rightarrow \Delta W = -A_q < 0$. This case occurs when $A_p < -A_q$. Then, $\Delta W = \max(A_p, -A_q)$.

ii. if $S_{qp} < A_p + S_{pq}$,

A. if $S_{qp} \geq -A_q + S_{pq}$, (7.12)$\Rightarrow \Delta W = A_p < 0$. This case occurs when $A_p > -A_q$. Then, $\Delta W = \max(A_p, -A_q)$.

B. if $S_{qp} < -A_q + S_{pq}$, (7.12)$\Rightarrow \Delta W = A_p - A_q + S_{pq} - S_{qp}$. (7.13)$\Rightarrow S_{qp} > A_p - A_q + S_{pq}$. If S_{qp} increases, ΔW decreases. In addition, because $S_{qp} < \min(A_p, -A_q) + S_{pq}$, we have $\Delta W > A_p - A_q - \min(A_p, -A_q) = \max(A_p, -A_q)$.

In case 2b, S_{qp} affects ΔW in three phases as illustrated in Figure 7.33 :

- when S_{qp} increases from $\max(0, A_p - A_q + S_{pq})$ to $\min(A_p, -A_q) + S_{pq}$, ΔW decreases from $\min(0, A_p - A_q + S_{pq})$ to $\max(A_p, -A_q)$;
- when S_{qp} falls within the range $[\min(A_p, -A_q) + S_{pq}, \max(A_p, -A_q) + S_{pq}]$, then $\Delta W = \max(A_p, -A_q)$;
- when S_{qp} increases from $\max(A_p, -A_q) + S_{pq}$ to S_{pq}, then ΔW increases from $\max(A_p, -A_q)$ to 0.

To satisfy (7.13), we choose S_{qp} the closest to $\min(A_p, -A_q) + S_{pq}$.

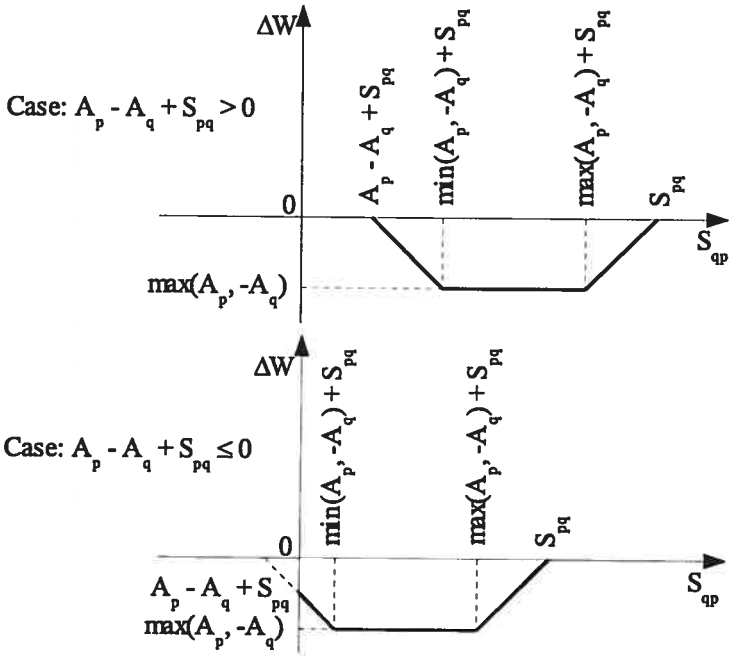

FIG. 7.33 – $\Delta W(S_{qp})$ vs. S_{qp} where $S_{pq} > \max(-A_p, A_q)$

In case 2b ($S_{pq} > \max(-A_p, A_q)$), S_{qp} is the closest to $\min(A_p, -A_q) + S_{pq}$ and respects the condition : $0 \leq S_{qp} < S_{pq}$ and $S_{qp} > A_p - A_q + S_{pq}$.

□

Chapitre 8

Discussion

Les chapitres précédents ont présenté les résultats de notre travail. En conclusion, nous présentons les contributions du travail et identifions les problèmes ouverts.

8.1 Contributions

À notre connaissance, aucun système P2P structuré existant ne supporte l'équilibrage intégré de charge de stockage et de charge de gestion d'index. La contribution de notre travail est de proposer une solution combinée de ces deux problèmes. La solution comporte des mécanismes d'équilibrage de charge de stockage et de charge de gestion d'index intégrés dans un système P2P structuré (BALLS). De plus, notre solution vise deux autres objectifs : (1) la prise en compte de tous les facteurs d'impact sur l'équilibrage de charge (distribution des clés sur les pairs, distribution des objets sur les clés, popularité et taille des objets, capacité des pairs et dynamisme des pairs et des objets), et (2) la réduction des coûts de maintenance. Ces considérations sont souvent négligées par les travaux connexes. La discussion ci-dessous montre que notre solution répond à ces exigences. Nous commençons par une discussion sur

la performance des mécanismes d'équilibrage de charge.

8.1.1 Performance des mécanismes d'équilibrage de charge

Dans notre approche, l'équilibrage de charge (de stockage et de gestion d'index) a pour but de minimiser la surcharge totale dans le système. Il se base sur le transfert dynamique des objets et des clés, respectivement. Chaque transfert élémentaire minimise la surcharge combinée des pairs impliqués en économisant de surcroît le coût de trafic. Les preuves théoriques (voir pages 116, 163 et 165) montrent que les transferts effectués satisfont cette exigence avec une haute probabilité. L'agrégation des tels transferts élémentaires mène à la minimisation de la surcharge totale.

Les simulations ont confirmé la performance des mécanismes d'équilibrage. Elles l'ont évaluée sur deux propriétés : efficacité et convergence. L'évaluation de l'efficacité de l'équilibrage de charge de stockage (sect. 7.3.3) compare le ratio de surcharge Ψ produit par deux scénarios : équilibrage activé et équilibrage désactivé. Sans équilibrage, Ψ augmente très vite en parallèle avec l'augmentation du ratio d'utilisation de stockage ($Z = (\sum_{\forall p} S_p)/(\sum_{\forall p} \overline{D}_p)$). Cependant, l'équilibrage activé ralentit beaucoup l'augmentation de Ψ (pour $Z \leq 100\%$). La croissance ne devient rapide que si $Z > 100\%$, donc lorsqu'il manque de l'espace disponible. L'équilibrage prend effet dans un large intervalle de niveaux de dynamisme des pairs. Naturellement, plus le réseau est dynamique, plus l'efficacité de l'équilibrage est réduite.

L'évaluation de la convergence de l'équilibrage de charge de stockage garde Z fixe à un certain niveau (tout au long d'une exécution) et vérifie la réduction de Ψ. Les simulations réduisent (sans augmenter) Ψ jusqu'à une valeur minimale Ψ_f. L'existence d'un tel ratio de surcharge stable Ψ_f dans toutes les simulations démontre la convergence attendue.

Nous avons aussi identifié deux stratégies pour le transfert de charge dans l'équilibrage

de charge de stockage : *transfert orienté coût* et *transfert orienté surcharge*. La première stratégie vise à minimiser la surcharge combinée des pairs mais assure que le coût de migration des objets ne dépasse pas la réduction de la surcharge. La deuxième stratégie minimise la surcharge combinée à tout prix. Les simulations montrent que l'application de la deuxième stratégie donne toujours un meilleur ratio de surcharge que l'application de l'autre stratégie. Cependant, dans la vérification de la convergence, elle demande un plus long temps de stabilisation et un coût de migration plus élevé.

L'évaluation de l'équilibrage de charge de gestion d'index (sect. 7.3.2) explore l'efficacité et la convergence dans une même simulation. Celle-ci comporte trois phases : phase 1 (30 cycles de simulation) sans équilibrage, phase 2 (70 cycles de simulation) équilibrage activé et phase 3 (derniers 30 cycles de simulation) sans équilibrage. La comparaison entre les résultats des phases 1 et 2 révèle l'efficacité de l'équilibrage. D'autre part, la comparaison entre les résultats des phases 2 et 3 montre la convergence. Nous examinons aussi l'impact des différentes conditions sur la performance de l'équilibrage. Un grand intervalle de paramètres a été essayé : ratio d'utilisation de gestion d'index ($U = (\sum_{\forall p} T_p)/(\sum_{\forall p} C_p)$), dynamisme des pairs ($\pi$) et dynamisme de la popularité de clé (τ).

Dans toutes les simulations, le ratio de surcharge Ω diminue rapidement en phase 2 par rapport à la phase 1. Cela prouve l'efficacité de l'approche. Évidemment, l'efficacité diminue si U ou π croît. Pour vérifier la convergence, nous supposons qu'il n'y a pas de dynamisme des pairs ($\pi = 0$). Dans ce cas, la phase 3 garde Ω stable, inférieur à celui de la phase 2. Ce résultat exprime une bonne distribution des clés (faite lors de la phase 2), qui n'augmente pas la surcharge malgré l'arrêt de l'équilibrage. Ceci montre la convergence attendu. Si $\pi > 0$, le changement de la distribution de capacité des pairs bouleverse l'équilibre établi au cours de la phase 2. Il en résulte une augmentation de la surcharge.

Les simulations présentent un point intéressant à remarquer. Le dynamisme de la po-

pularité de clé (τ) a un impact très faible sur la performance de l'équilibrage de charge de gestion d'index. Ce phénomène est une conséquence de l'application du graphe de De Bruijn dans le routage de BALLS. À chaque étape, le routage choisit une clé se rapprochant le plus de la cible. Comme il y a souvent plusieurs clés ayant une même distance à la cible, la sélection de la prochaine clé dans cet ensemble est aléatoire. Grâce à une large répartition des clés ayant une même distance à la cible sur tout l'espace de clés (voir les assertions 1 et 2, sect. 5.3.1), la distribution des chemins de routage est quasi indépendante de la distribution des cibles. Cette propriété explique le peu d'impact observé.

8.1.2 Équilibrage intégré

Nous discutons maintenant de la performance du fonctionnement concurrent des deux mécanismes d'équilibrage. L'équilibrage de charge de stockage et de charge de gestion d'index se base sur deux types de transfert. L'équilibrage de charge de stockage transfère des objets (des pairs surchargés à des pairs non-surchargés). D'autre part, l'équilibrage de charge de gestion d'index transfère des clés. Pour rendre ces transferts possibles, BALLS sépare (1) l'emplacement d'un objet et celui de sa clé et (2) les clés et l'id des pairs. La première séparation permet l'indépendance entre les clés et les objets. La deuxième séparation garantit l'indépendance entre la distribution des clés et l'id (fixe) des pairs. On peut donc intégrer l'équilibrage de charge de gestion d'index basé sur le transfert dynamique des clés. En conséquence, les deux mécanismes d'équilibrage de charge prennent effet sans affecter l'efficacité de l'autre. Nos simulations ont confirmé cette assertion.

L'évaluation du fonctionnement simultané des deux mécanismes d'équilibrage de charge (sect. 7.3.4) exécute quatre cas de simulation :

- sans équilibrage de charge (cas 00),
- seulement avec l'équilibrage de charge de stockage (cas 01),

- seulement avec l'équilibrage de charge de gestion d'index (cas 10) et
- simultanément avec les deux mécanismes d'équilibrage (cas 11).

La comparaison entre les graphes de ratio de surcharge obtenus pour les quatre cas nous permet d'observer l'impact réciproque des mécanismes d'équilibrage.

Les résultats des simulations montrent qu'on ne peut pas percevoir des différences entre les graphes du ratio de surcharge de stockage Ψ des deux cas 01 et 11. Nous trouvons une relation similaire entre les graphes de Ψ produits par les cas 00 et 10. Ceci signifie que l'efficacité de l'équilibrage de charge de stockage est indépendante de l'activation de l'équilibrage de charge de gestion d'index. D'autre part, les graphes du ratio de surcharge de gestion d'index Ω sont indistincts entre les cas 10 et 11, et entre les cas 00 et 01. Cette comparaison indique que l'activation de l'équilibrage de charge de stockage n'influence pas l'efficacité de l'équilibrage de charge de gestion d'index.

Une évaluation similaire effectuée dans un système dont le stockage des objets est attaché à leur clé (sect. 7.4.1) donne une autre relation entre les mécanimes d'équilibrage de charge. Les simulations s'exécutent dans les mêmes quatre cas 00, 01, 10 et 11. Cependant, les graphes de Ψ et Ω ne sont plus indépendants. L'activation d'un aspect d'équilibrage de charge réduit l'efficacité de l'autre aspect, et vice-versa.

L'évaluation montre que l'indépendance entre l'emplacement des objets et les clés assure l'indépendance des mécanismes d'équilibrage. Cependant, lorsque cet emplacement est lié aux clés, l'efficacité des équilibrages est réduite s'ils opèrent concurremment. Cela prouve la supériorité de BALLS dans la résolution du problème de l'équilibrage de charge intégré.

8.1.3 Prise en compte des facteurs d'impact

Nos méthodes d'équilibrage de charge prennent en compte les facteurs d'impact suivants : la distribution des clés sur les pairs, la distribution des objets sur les clés, la taille et

la popularité des objets, la capacité des pairs, le dynamisme des objets et le dynamisme des pairs. Nous calculons la charge de stockage d'un pair en additionnant la taille des objets sur le pair. D'autre part, la charge de gestion d'index d'un pair est déterminée par le volume des messages de routage y arrivant dans une unité de temps. Ces calculs déterminent la charge courante sur un pair à chaque moment. Ceci garantit que notre approche considère la distribution des clés sur les pairs, la distribution des objets sur les clés, la taille, la popularité et le dynamisme des objets.

Les opérations d'équilibrage de charge que nous proposons s'intéressent à la surcharge plutôt qu'à la charge. La surcharge est calculée lors de chaque décision de transfert. De cette façon, les mécanismes d'équilibrage tiennent compte de la capacité des pairs et de leur dynamisme. La prise en compte de ces facteurs est aussi confirmée par les simulations. L'évaluation des mécanismes d'équilibrage (sect. 7.3.2 et 7.3.3) nous montre leur efficacité même dans un environnement avec un grand dynamisme des pairs et des objets.

8.1.4 Économie du coût de maintenance

Le coût de maintenance bas résulte d'abord du degré bas des pairs. L'application du graphe de De Bruijn produit un degré constant (sect. 7.3.1). Les résultats des simulations montrent aussi une relation linéaire des coûts d'arrivée et de départ avec le degré des pairs. Leur moyenne est constante et indépendante de la taille du système. Les systèmes ayant un degré logarithmique des pairs (p.ex., Tapestry, Chord, Pastry) exigent souvent un coût de maintenance $O(\log^2 n)$.

Expressways utilise une autre approche d'équilibrage de charge de gestion d'index basée sur la réorganisation des liens entre les pairs. Cette approche exige un grand nombre de liens. Selon nos résultats de simulation (sect. 7.4.3), le nombre de liens maintenus par un pair dans Expressways est logarithmique en fonction de la taille du système. Il est considérablement

plus grand que le degré constant de BALLS, notamment quand le système évolue.

L'application des graphes de De Bruijn dote BALLS d'une haute extensibilité. Dans les systèmes dont le degré dépend de la taille (p.ex., $O(\log n)$), plus le système évolue, plus complexes sont le statut des pairs et le coût de maintenance. Ces attributs restent constants dans BALLS même si la taille s'approche de 2^{128} (la taille de l'espace d'adresses IPv6). Les échanges de clés et d'objets pour l'équilibrage de charge sont réalisés entre les voisins directs. Leur complexité est théoriquement indépendante de la taille du système.

Un autre facteur permettant d'économiser sur le coût de maintenance est la séparation entre les objets et leur clé. Dans les systèmes traditionnels qui rattachent le lieu de stockage des objets à celui des clés, un déplacement de responsabilité de clés entraîne le transfert des objets correspondants. L'arrivée et le départ des pairs impliquent toujours un déplacement de responsabilité de clés. Dans ces systèmes, le coût de maintenance comprend un coût de migration des objets. On a vu que même lorsqu'un système relaxe cette contrainte (p.ex., PAST), un coût de migration existe lors de la maintenance. Les simulations présentées à la section 7.4.2 ont étudié le volume des objets déplacés par 500 arrivées dans PAST. Le coût de migration observé est toujours considérable pour plusieurs valeurs du ratio d'utilisation de stockage Z (10%, 30%, 50%, 70% et 90%).

Au contraire, BALLS n'a pas à déplacer les objets lors de l'arrivée d'un pair. Au lieu d'un coût de migration d'objet, BALLS subit un coût de transmission des indices des objets et un coût de notification du changement de racine. Dans des applications avec des objets de grande taille (p.ex., services de stockage de ressources multimédia), ces coûts additionnels ne sont pas considérables par rapport à la migration des objets eux-mêmes. La séparation complète de l'emplacement des objets de celui de leur clé présente le même avantage pour l'équilibrage de charge de gestion d'index, qui déplace dynamiquement des clés entre pairs.

Au départ d'un pair de BALLS, on n'a qu'à déplacer les objets ayant une copie unique

pour garantir leur disponibilité. La réplication permet de négliger les autres objets car leur copie sera rétablie plus tard. Ce comportement réduit la complexité du mécanisme de départ.

Plusieurs autres méthodes choisissent d'égaliser le taux de charge/capacité de tous les pairs. Cet objectif exige la réaction du système à tout moment lorsque l'égalité de ce taux est brisée. En arguant qu'un pair est capable de maintenir sa performance s'il subit une charge couverte par sa capacité, nous choisissons de minimiser la surcharge du système. Ceci permet aux pairs de rester inactifs s'ils n'ont pas de surcharge. Une opération d'équilibrage ne s'exécute que si elle a une possibilité de diminuer la surcharge. Cela nous permet d'éviter des réorganisations inutiles.

8.1.5 Degré des pairs, coût de routage et équilibrage de charge

Datta et al. [DGA04] arguent qu'un système P2P inspiré du graphe de De Bruijn (p.ex., CAN D2B, Koorde) ne peut pas maintenir à la fois un degré constant des pairs, un coût logarithmique de routage et une distribution de responsabilité de clés adaptée à une distribution biaisée de charge. Selon ces auteurs, un réseau appliquant le graphe de De Bruijn ayant une distribution biaisée de responsabilité de clés ne garantit pas une borne logarithmique du coût de routage. Ils expliquent que ce problème provient du manque de liens nécessaires pour le routage quand la distribution des clés sur les pairs est inégale. Rappelons qu'un pair p dans CAN D2B n'a pas besoin de se connecter à tous les pairs voisins possibles ayant l'id plus long que celui de p (sect. 2.3.3). Les voisins possibles de p sont les pairs liés à p par au moins un arc de De Bruijn. Si on suppose que chaque pair doit maintenir les liens à tous les voisins possibles (c.-à-d., la connexion des pairs couvre tous les arcs du graphe de De Bruijn), le coût de routage redevient logarithmique mais le degré des pairs n'est plus constant.

Revenons sur la signification de la propriété « degré constant des pairs ». Datta et al. [DGA04] l'utilisent pour signifier la borne supérieure constante du degré de tous les

pairs. D'après leur analyse, la limite constante du degré des pairs et le coût logarithmique de routage sont deux buts conflictuels des applications du graphe de De Bruijn ayant une distribution inégale des clés sur les pairs.

Lorsque nous avons analysé les coûts de maintenance (coût d'arrivée et coût de départ) du système global, nous avons considéré le degré moyen des pairs plutôt que le degré maximal. Selon les analyses théoriques (sect. 7.2.1) et empiriques (sect. 7.3.1), les coûts d'arrivée et de départ sont proportionnels au degré moyen des pairs. Nos simulations montrent que le degré moyen des pairs est constant en dépit du nombre de pairs (fig. 7.6). Elles présentent aussi un taux très faible de pairs ayant le degré supérieur à 20 (fig. 7.7). De plus, la connexion entre les pairs couvre tous les arcs de De Bruijn. Cela garantit une borne logarithmique pour le coût de routage. Clairement, ces simulations démontrent qu'il n'y a pas de conflit entre le degré moyen constant des pairs et le coût logarithmique de routage, peu importe la distribution des clés sur les pairs.

8.2 Problèmes ouverts

Nous décrivons ici certains problèmes ouverts. La première remarque concerne l'évaluation du coût des notifications lors des déplacements dynamiques des clés et des objets à travers le réseau. Rappelons que BALLS sépare les ids des pairs, les clés et les objets pour des fins d'équilibrage de charge et d'économie du coût de maintenance. Ceci permet la migration indépendante des clés et des objets. Les protocoles *notification de racine* et *notification de stockage* (sect. 7.2.3) transmettent des messages de notification pour mettre à jour les pointeurs de référence dans le système après chaque déplacement. En pratique, le coût de ces messages est beaucoup plus petit que le coût de migration des objets. L'évaluation du coût des message de notification est ainsi négligée dans le travail présent.

Une autre évaluation à faire est celle dans un réseau réel tel que PlanetLab. Le simulateur de BALLS supporte deux modes de simulation. Le mode centralisé a servi à l'évaluation décrite dans cette thèse, notamment à la section 7.3. Le mode décentralisé n'a pas encore été utilisé. Nous le réservons pour une vérification du comportement de BALLS dans un environnement réel parallèle et à temps réel. Cette vérification nous permettra de faire les rajustements nécesaires afin de fabriquer un système BALLS applicable.

Pendant le fonctionnement du réseau, des pairs peuvent potentiellement tomber en panne ou se déconnecter sans avertir. De tels événements peuvent empêcher le fonctionnement correct (p.ex., routage/recherche, gestion des objets) du système. La technique de réplication des objets incorporée dans le système ne suffit pas pour contrer ces pannes. Elle augmente la disponibilité des objets. Cependant, une perte d'un pair sans avertissement peut trouer le répertoire des indices des objets. De plus, le départ d'un pair peut détériorer la disponibilité des objets s'il manque de capacité de stockage. Des techniques additionnelles qui minimisent l'impact négatif des pannes et maintiennent la persistance des données ainsi que l'état valide du système doivent être développées.

Notre solution équilibre la charge en se basant sur des capacités définies a priori. Le mécanisme de spécification de ces capacité n'est pas défini. Normalement, des participants autonomes au système veulent l'exploiter le plus mais y contribuer le moins. Les valeurs qu'ils choisissent ne reflètent pas toujours les vraies capacités. Un choix égoïste des capacités par les usagers peut réduire les ressources disponibles dans le système. De tels choix ne compromettent cependant pas le fonctionnement valide du système, car les capacités utilisées sur chaque pair sont basées sur les capacités déclarées. Toutefois, il est clair qu'un mécanisme de détermination des capacités réelles assurerait une meilleure performance du système P2P.

Quoique les problèmes ci-dessus ne soient pas l'objet du travail actuel, il paraît essentiel de les résoudre pour rendre notre système applicable dans un environnement pratique.

Bibliographie

[Abe01] Karl Aberer. P-Grid : A self-organizing access structure for P2P information systems. In 6^{th} *International Conference on Cooperative Information Systems (CoopI'01), LNCS 2172, Springer Verlag*, pages 179–194, Trento, Italy, September 2001.

[ACMD+03] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-Grid : A self-organizing structured P2P system. *Special Section on Peer to Peer Data Management, SIGMOD Record*, 32(3) :29–33, September 2003.

[ACMR02] Sameer Ajmani, Dwaine E. Clarke, Chuang-Hue Moh, and Steven Richman. Conchord : Cooperative SDSI certificate storage and name resolution. In 1^{st} *International Workshop on Peer-to-Peer Systems (IPTPS'02), LNCS 2429, Springer Verlag*, pages 141–154, Cambridge, Massachusetts, USA, March 2002.

[ADH03] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth. The quest for balancing peer load in structured peer-to-peer systems. Technical Report IC/2003/32, École Polytechnique Fédérale de Lausanne, 2003.

[adHV99] Friedhelm Meyer auf der Heide and Berthold Vöcking. Shortest-path routing in arbitrary networks. *Journal of Algorithms*, 31(1) :105–131, April 1999.

[And01] David Anderson. Seti@home. In *Peer-to-Peer Harnessing the Power of Disruptive Technologies*. O'Reilly, ISBN 0-596-00110-X, 2001.

[AS03] Baruch Awerbuch and Christian Scheideler. Peer-to-peer systems for prefix search. In 22^{nd} *Annual Symposium on Principles of Distributed Computing (ACM PODC'03)*, pages 123–132, Boston, Massachusetts, USA, July 2003.

[BCM03] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple load balancing for distributed hash tables. In 2^{nd} *International Workshop on Peer-to-peer Systems (IPTPS'03), LNCS 2735, Springer Berlin / Heidelberg*, pages 80–87, Berkeley, California, USA, February 2003.

[BDE97] Jean-Claude Bermond, Robin W. Dawes, and Fahir Ö. Ergincan. De bruijn and kautz bus networks. *Networks*, 30(3) :205–218, 1997.

[BE96] Jean-Claude Bermond and Fahir Ö. Ergincan. Bus interconnection networks. *Discrete Applied Mathematics*, 68(1-2) :1–15, June 1996.

[bit06] BitTorrent. http ://www.bittorrent.com, 2006.

[BKadH05] Marcin Bienkowski, Miroslaw Korzeniowski, and Friedhelm Meyer auf der Heide. Dynamic load balancing in distributed hash tables. In 4^{th} *International Workshop on Peer-to-peer Systems (IPTPS'05), LNCS 3640, Springer Berlin / Heidelberg*, pages 217–225, New York, USA, February 2005.

[BKK+03] Hari Balakrishnan, M. Frans Kaahoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems. *Communication of the ACM*, 46(2) :43–48, February 2003.

[BSV03] Ranjita Bhagwan, Stefan Savage, and Geoffrey Voelker. Understanding availability. In 2^{nd} *International Workshop on Peer-to-peer Systems (IPTPS'03), LNCS 2735, Springer Berlin / Heidelberg*, pages 256–267, Berkeley, California, USA, February 2003.

[CCRS87] Fan R.K. Chung, Edward G. Coffman, Martin I. Reiman, and Burton Simon. The forwarding index of communication networks. *IEEE Transactions on Information Theory*, IT-33(2) :224–232, March 1987.

[CDK+03] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream : High-bandwidth content distribution in a cooperative environment. In 2^{nd} *International Workshop on Peer-to-peer Systems (IPTPS'03), LNCS 2735, Springer Berlin / Heidelberg*, pages 292–303, Berkeley, California, USA, February 2003.

[CDKR02] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. SCRIBE : A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 20(8) :1489–1499, October 2002.

[CDKR03] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scalable application-level anycast for highly dynamic groups. In 5^{th} *COST264 International Workshop on Networked Group Communications (NGC'03), LNCS 2816, Springer*, pages 47–57, Munich, Germany, September 2003.

[CHM+02] Ian Clarke, Theodore W. Hong, Scott G. Miller, Oskar Sandberg, and Brandon Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, 6(1) :40–49, 2002.

[CMM02] Russ Cox, Athicha Muthitacharoen, and Robert T. Morris. Serving DNS using a peer-to-peer lookup service. In 1^{st} *International Workshop on Peer-to-Peer Systems (IPTPS'02), LNCS 2429, Springer Verlag*, pages 155–165, Cambridge, Massachusetts, USA, March 2002.

[CMS95] Richard Cole, Bruce M. Maggs, and Ramesh K. Sitaraman. Routing on butterfly networks with random faults. In 36^{th} *Symposium on Foundations of*

Computer Science (FOCS'95), pages 558–570, October 1995.

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet : A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies : Workshop on Design Issues in Anonymity and Unobservability, LNCS 2009, Springer Berlin / Heidelberg*, page 46, 2001.

[dB46] N. G. de Bruijn. A combinatorial problem. In *Koninklijke Nederlands Akademie van Wetenschappen*, volume 49, 1946.

[DB99] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *ACM International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS'99)*, pages 59–70, Atlanta, Georgia, USA, May 1999.

[DGA04] Anwitaman Datta, Sarunas Girdzijauskas, and Karl Aberer. On de Bruijn routing in distributed hash tables : There and back again. In 4^{th} *IEEE International Conference on Peer-to-Peer Computing (P2P'04)*, pages 159–166, Zurich, Switzerland, August 2004.

[DKK+01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In 18^{th} *ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 202–215, Chateau Lake Louise, Banff, Canada, October 2001.

[Dow01a] Allen B. Downey. The structural cause of file size distributions. In *ACM International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS'01)*, pages 328–329, Cambridge, Massachusetts, USA, 2001.

[Dow01b] Allen B. Downey. The structural cause of file size distributions. In 9^{th} *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'01)*, page 361, 2001.

[DR01] Peter Druschel and Antony Rowstron. PAST : A large-scale, persistent peer-to-peer storage utility. In 8^{th} *Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 75–80, Elmau/Oberbayern, Germany, May 2001.

[DT03] Oren Dobzinski and Anat Talmy. Viceroy - on the implementation of a peer to peer network. Technical Report 2003-77, Leibnitz Center of the School of Computer Science and Engineering, the Hebrew University of Jerusalem, September 2003.

[DZD+03] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common API for structured peer-to-peer overlays. In 2^{nd} *International Workshop on Peer-to-peer Systems (IPTPS'03), LNCS 2735, Springer Berlin / Heidelberg*, pages 33–44, Berkeley, California, USA, February 2003.

[edo03] eDonkey2000 - Overnet. http ://www.edonkey2000.com, 2003.

[FG03] Pierre Fraigniaud and Philippe Gauron. Brief announcement : An overview of the content-addressable network D2B. In 22^{nd} *Annual Symposium on Principles of Distributed Computing (ACM PODC'03)*, page 151, Boston, Massachusetts, USA, July 2003.

[fre05] FreePastry. http ://freepastry.rice.edu/freepastry/, 2005.

[fre06] The Freenet project. http ://freenet.sourceforge.net, viewed 2006.

[GDJ06] Saikat Guha, Neil Daswani, and Ravi Jain. An experimental study of the Skype peer-to-peer VoIP system. In 5^{th} *International Workshop on Peer-to-peer Systems (IPTPS'06)*, Santa Barbara, California, USA, February 2006.

[GDS+03] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In 19^{th} *ACM Symposium on Operating Systems Principles (ACM SOSP'03)*, pages 314–329, Bolton Landing, New York, USA, October 2003.

[GFJ+03] Zihui Ge, Daniel R. Figueiredo, Sharad Jaiswal, Jim Kurose, and Don Towsley. Modeling peer-peer file sharing systems. In 22^{nd} *Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE INFOCOM'03)*, San Francisco, California, USA, April 2003.

[gnu01] Gnutella.com home page. http ://www.gnutella.com, 2001.

[HAR+03] Qi He, Mostafa Ammar, George Riley, Himanshu Raj, and Richard Fujimoto. Mapping peer behavior to packet-level details : A framework for packet-level simulation of peer-to-peer systems. In 11^{th} *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (IEEE MASCOTS'03)*, pages 71–78, October 2003.

[HKRZ02] Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed object location in a dynamic network. In 14^{th} *ACM Symposium on Parallel Algorithms and Architectures (ACM SPAA'02)*, pages 41–52, Winnipeg, Manitoba, Canada, August 2002.

[HR02] Steven Hand and Timothy Roscoe. Mnemosyne : Peer-to-peer steganographic storage. In 1^{st} *International Workshop on Peer-to-Peer Systems (IPTPS'02), LNCS 2429, Springer Verlag*, pages 130–140, Cambridge, Massachusetts, USA, March 2002.

[IRD02] Sitaram Iyer, Antony Rowstron, , and Peter Druschel. Squirrel : A decentralized peer-to-peer web cache. In 21^{st} *Annual ACM Symposium on Principles of*

Distributed Computing (ACM PODC'02), pages 213–222, Monterey, California, USA, July 2002.

[Jos02] Sam Joseph. NeuroGrid : Semantically routing queries in peer-to-peer networks. In *Web Engineering and Peer-to-Peer Computing (NETWORKING'02), LNCS 2376, Springer Berlin / Heidelberg*, pages 202–214, Pisa, Italy, May 2002.

[Jos03] Sam Joseph. An extendible open source P2P simulator. *P2PJournal*, November 2003.

[kaz05] KaZaA. http ://www.kazaa.com, 2005.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore : An architecture for global-scale persistent storage. In 9^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*, pages 190–201, Cambridge, Massachusetts, USA, November 2000.

[KK03] M. Frans Kaashoek and David R. Karger. Koorde : A simple degree-optimal distributed hash table. In 2^{nd} *International Workshop on Peer-to-peer Systems (IPTPS'03), LNCS 2735, Springer Berlin / Heidelberg*, pages 98–107, Berkeley, California, USA, February 2003.

[KR04a] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In 3^{rd} *International Workshop on Peer-to-peer Systems (IPTPS'04)*, pages 131–140, San Diego, California, USA, February 2004.

[KR04b] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In 16^{th} *ACM Symposium on Parallelism in*

Algorithms and Architectures (ACM SPAA'04), pages 36–43, Barcelona, Spain, June 2004.

[LBK06a] Viet Dung Le, Gilbert Babin, and Peter Kropf. BALLS : a structured peer-to-peer system with integrated load balancing. *Special issue of Annals of Telecommunications on the theme New technologies in distributed systems (NOTERE)*, 2006.

[LBK06b] Viet Dung Le, Gilbert Babin, and Peter Kropf. BALLS simulator : Evaluator of a structured peer-to-peer system with integrated load balancing. In Patrick Bellot, Vu Duong, and Marc Bui, editors, 4^{th} *IEEE International Conference on Computer Sciences : Research, Innovation and Vision for the Future (RIVF'06), Addendum contributions*, pages 59–64, Ho Chi Minh City, Vietnam, February 2006.

[LBK06c] Viet Dung Le, Gilbert Babin, and Peter Kropf. A structured peer-to-peer system with integrated index and storage load balancing. In *Innovative Internet Community Systems (I2CS'05), LNCS 3908, Springer Berlin / Heidelberg*, pages 41–52, 2006.

[LKR05] Jian Liang, Rakesh Kumar, and Keith W. Ross. The KaZaA overlay : A measurement study. *Computer Networks*, 49(6), October 2005.

[LKRG03] Dmitri Loguinov, Anuj Kumar, Vivek Rai, and Sai Ganesh. Graph-theoretic analysis of structured peer-to-peer systems : Routing distance and fault resilience. In *ACM SIGCOMM'03*, pages 395–406, Karlsruhe, Germany, August 2003.

[Loo03] Alred W. Loo. The future of peer-to-peer computing. *Communication of the ACM*, 46(9) :57–61, September 2003.

[LRS02] Qin Lv, Sylvia Ratnasamy, and Scott Shenker. Can heterogeneity make Gnutella scalable? In 1^{st} *International Workshop on Peer-to-Peer Systems (IPTPS'02), LNCS 2429, Springer Verlag*, pages 94–103, Cambridge, Massachusetts, USA, March 2002.

[Man04] Gurmeet Singh Manku. Balanced binary trees for id management and load balance in distributed hash tables. In 23^{rd} *Annual ACM symposium on Principles of distributed computing (ACM PODC'04)*, pages 197–205, St. John's, Newfoundland, Canada, July 2004.

[MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy : A scalable and dynamic emulation of the butterfly. In 21^{st} *Annual ACM Symposium on Principles of Distributed Computing (ACM PODC'02)*, pages 183–192, Monterey, California, USA, July 2002.

[MRS01] Michael Mitzenmacher, Andréa W. Richa, and Ramesh Sitaraman. *The Power of Two Random Choices : A Survey of Techniques and Results*, volume 1 of *Handbook of Randomized Computing*, pages 255–305. Kluwer Press, July 2001.

[MSZ03] Shashidhar Merugu, Sridhar Srinivasan, and Ellen Zegura. p-sim : A simulator for peer-to-peer networks. In 11^{th} *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (IEEE MASCOTS'03)*, pages 213–218, October 2003.

[nap06] Napster. http ://www.napster.com, viewed 2006.

[NW03a] Moni Naor and Udi Weider. Novel architectures for P2P application : the continuous-discrete approach. In 15^{th} *Annual ACM Symposium on Parallel Algorithms and Architectures (ACM SPAA'03)*, pages 50–59, San Diego, California, USA, June 2003.

[NW03b] Moni Naor and Udi Wieder. A simple fault tolerant distributed hash table. In 2^{nd} *International Workshop on Peer-to-peer Systems (IPTPS'03), LNCS 2735, Springer Berlin / Heidelberg*, pages 88 – 97, Berkeley, California, USA, February 2003.

[pee05] Peersim peer-to-peer simulator. http ://peersim.sourceforge.net/, 2005.

[pee06] PeerCast.org. http ://www.peercast.org, viewed 2006.

[Rat02] Sylvia Paul Ratnasamy. *A Scalable Content-Addressable Network.* PhD thesis, University of California at Berkeley, 2002.

[RD01a] Antony Rowstron and Peter Druschel. Pastry : Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In 18^{th} *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.

[RD01b] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In 18^{th} *ACM Symposium on Operating Systems Principles (ACM SOSP'01)*, pages 188–201, Banff, Alberta, Canada, October 2001.

[REG+03] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond : The OceanStore prototype. In 2^{nd} *USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, California, USA, 2003. USENIX.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *ACM SIGCOMM'01*, pages 160–172, San Diego, California, USA, August 2001.

[RFI02] Matei Ripeanu, Ian Foster, and Adriana Iamnitchi. Mapping the Gnutella network : Properties of large-scale peer-to-peer systems and implications for sys-

tem design. *IEEE Internet Computing Journal*, 6(1), 2002.

[RLS+03] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in structured P2P systems. In 2^{nd} *International Workshop on Peer-to-peer Systems (IPTPS'03), LNCS 2735, Springer Berlin / Heidelberg*, pages 68–79, Berkeley, California, USA, February 2003.

[set06] SETI@Home. http ://setiathome.berkeley.edu, viewed 2006.

[SGD+02] Stefan Saroiu, Krishna P. Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy. An analysis of internet content delivery systems. In 5^{th} *Symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 315–327, Boston, Massachusetts, USA, December 2002.

[Shi01] Clay Shirky. Listening to Napster. In *Peer-to-Peer Harnessing the Power of Disruptive Technologies*. O'Reilly, ISBN 0-596-00110-X, 2001.

[sky06] Skype. http ://www.skype.com, 2006.

[SMK+01] Ion Stoica, Robert Moris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for Internet applications. In *ACM SIGCOMM'01*, pages 149–160, San Diego, California, USA, August 2001.

[sta06] Stardust@Home. http ://stardustathome.ssl.berkeley.edu, 2006.

[THS05] Egemen Tanin, Aaron Harwood, and Hanan Samet. A distributed quadtree index for peer-to-peer settings. In 21^{st} *International Conference on Data Engineering (IEEE ICDE'05)*, April 2005.

[udc04] United devices cancer research projet. http ://www.grid.org/projects/cancer, 2004.

[VBKJ02] Jean Vaucher, Gilbert Babin, Peter Kropf, and Thierry Jouve. Experimenting with gnutella communities. In *Distributed Communities on the Web (DCW'02), LNCS 2468, Springer Berlin*, pages 85–99, Sydney, Australia, April 2002.

[WZLL04] Xiaoming Wang, Yueping Zhang, Xiafeng Li, and Dmitri Loguinov. On zone-balancing of peer-to-peer networks : Analysis of random node join. In *ACM SIGMETRICS'04*, pages 211–222, New York, New York, USA, June 2004.

[YJF05] Wai Gen Yee, Dongmei Jia, and Ophir Frieder. Finding rare data objects in P2P file-sharing systems. In 5^{th} *IEEE International Conference on Peer-to-Peer Computing (IEEE P2P'05)*, pages 181–190, September 2005.

[ZHS+04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry : A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1) :41–53, January 2004.

[ZKJ01] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry : An infrastructure for fault-tolerance wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California Berkeley, April 2001.

[ZSZ02] Zheng Zhang, Shu-Ming Shi, and Jing Zhu. Self-balanced P2P expressways : When marxism meets confucian. Technical Report MSR-TR-2002-72, Microsoft Research, 2002.

[ZSZ03] Zheng Zhang, Shu-Ming Shi, and Jing Zhu. SOMO : self-organized metadata overlay for resource management in P2P DHT. In 2^{nd} *International Workshop on Peer-to-peer Systems (IPTPS'03), LNCS 2735, Springer Berlin / Heidelberg*, pages 170–182, Berkeley, California, USA, February 2003.

[ZZJ+01] Shelley Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John Kubiatowicz. Bayeux : An architecture for wide-area, fault-tolerant data disse-

mination. In 11^{th} *International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'01)*, pages 11–20, Port Jefferson, New York, USA, June 2001.

[ZZZ+03] Feng Zhou, Li Zhuang, Ben Y. Zhao, Ling Huang, Anthony D. Joseph, and John Kubiatowicz. Approximate object location and spam filtering on peer-to-peer systems. In *ACM/IFIP/USENIX International Middleware Conference (Middleware'03)*, pages 1–20, Rio de Janeiro, Brazil, June 2003.