P2P Infrastructure for Content Distribution

Manal El Dick

École Centrale de Nantes **Université de Nantes** **École des Mines de Nantes**

ÉCOLE DOCTORALE STIM

« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE MATHÉMATIQUES »

Année 2010

N° attribué par la bibliothèque

P2P Infrastructure for Content Distribution

THÈSE DE DOCTORAT

Discipline : Informatique

Spécialité : Bases de Données

Présentée
et soutenue publiquement par

Manal EL DICK

*Le 21 Janvier 2009 à l'*UFR *Sciences & Techniques, Université de Nantes,*
devant le jury ci-dessous

Président	:	Pr. Bernd Amann	Université Pierre et Marie Curie
Rapporteurs	:	Ricardo Jimenez-Peris, Professeur	Université de Madrid
		Jean-Marc Pierson, Professeur	Université Paul Sabatier
Examinateurs	:	Reza Akbarinia, Chargé de recherche	INRIA Nantes
		Anne-Marie Kermarrec, Directrice de recherche	INRIA Rennes
		Esther Pacitti, Professeur	Université de Montpellier

Directrice de thèse : Esther Pacitti

Laboratoire: LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE.
UMR CNRS 6241. 2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.

N° ED 503-080

P2P INFRASTRUCTURE FOR CONTENT DISTRIBUTION

Infrastructure P2P pour la Distribution de Contenu

Manal EL DICK

favet neptunus eunti

Université de Nantes

Manal EL DICK

P2P Infrastructure for Content Distribution

IV+**??**+VI p.

This document was edited with these-LINA LaTeX2e class of the "Association of Young Researchers on Computer Science ()" from the University of Nantes (available on : http://login.irin.sciences.univ-nantes.fr/). This LaTeX2e class is under the recommendations of the National Education Ministry of Undergraduate and Graduate Studies (circulaire n° 05-094 du 29 March 2005) of the University of Nantes and the Doctoral School of « Technologies de l'Information et des Matériaux(ED-STIM) », et respecte les normes de l'association française de normalisation (AFNOR) suivantes :

- AFNOR NF Z44-005 (décembre 1987)
  Documentation – Références bibliographiques – Contenu, forme et structure ;
- AFNOR NF Z44-005-2/ISO NF 690-2 (février 1998)
  Information et documentation – Références bibliographiques – Partie 2 : documents électroniques, documents complets ou parties de documents.

Print : thesisManal.tex – 02/02/2010 – 0:55.

Last class review:

Acknowledgements

My thanks go first to the members of my PhD committee for their time, reviews and encouragement. I would like to thank my advisor Esther Pacitti for she helped me to develop autonomy, patience and meticulous attention to detail. I am also very grateful to Patrick Valduriez for his perfect and invaluable management of the needs of the Atlas-GDD group. It is a pleasure to thank Bettina Kemme for our fruitful collaboration and her efficient feedbacks. Many, many thanks to my colleagues at Atlas-GDD, Philippe, Patricia, Sylvie, Mohammed, Eduardo, Jorge, Reza, Vidal and the others. I cherish our insightful discussions as much as our laughs, which enriched my PhD experience and made it more pleasant. Thank you !

I heartily thank the colleagues and friends that I met at the LINA over the years, for making it a delightful place to work. I was lucky to discover genuine friendship there. To Rabab, Matthieu, Anthony, Lorraine, Amenel, Mounir, I say : Our conversations enlightened my way of thinking. I truly hope our paths will cross again.

I owe an enormous debt of gratitude to my parents, brothers, family and friends -in Lebanon, France and the US, you know yourselves- for your amazing and unconditional support. You were here, despite the distance, whenever I needed to let off steam.

Finally, to Fadi ! Words alone cannot convey my thanks. Your confidence in me has forced me to never bend to difficulty and always defy my limits. I owe this achievement to you.

P2P Infrastructure for Content Distribution

Manal EL DICK

Abstract

The explosive growth of the Web requires new solutions for content distribution that meets the requirements of scalability, performance and robustness. At the same time, Web 2.0 has fostered participation and collaboration among users and has shed light on Peer-to-Peer (P2P) systems which involve resource sharing and decentralized collaboration. This thesis aims at building a low-cost infrastructure for content distribution based on P2P systems. However, this is extremely challenging given the dynamic and autonomous behavior of peers as well as the locality-unaware nature of P2P overlay networks. In the first stage, we focus on P2P file sharing as a first effort to build a basic infrastructure with loose requirements. We address the problem of bandwidth consumption from two angles: search inefficiency and long-distance file transfers. Our solution Locaware leverages inherent properties of P2P file sharing; it performs locality-aware index caching and supports keyword queries which are the most common in this context. In the second stage, we elaborate a P2P CDN infrastructure, which enables any popular and under-provisioned website to distribute its content with the help of its community of interested users. To efficiently route queries and serve content, Flower-CDN infrastructure intelligently combines different types of overlays with gossip protocols while exploiting peer interests and localities. PetalUp-CDN brings scalability and adaptability under massive and variable scales while the maintenance protocols provide high robustness under churn. We evaluate our solutions through extensive simulations and the results show acceptable overhead and excellent performance, in terms of hit ratio and response times.

Keywords : P2P systems, content distribution, interest-awareness, locality-awareness

Infrastructure P2P pour la Distribution de Contenu

Résumé

Le Web connaît ces dernières années un essor important qui implique la mise en place de nouvelles solutions de distribution de contenu répondant aux exigences de performance, passage à l'échelle et robustesse. De plus, le Web 2.0 a favorisé la participation et la collaboration entre les utilisateurs tout en mettant l'accent sur les systèmes P2P qui reposent sur un partage de ressources et une collaboration décentralisée. Nous avons visé, à travers cette thèse, la construction d'une infrastructure P2P pour la distribution de contenu. Toutefois, cette tâche est difficile étant donné le comportement dynamique et autonome des pairs ainsi que la nature des overlays P2P. Dans une première étape, nous nous intéressons au partage de fichiers en P2P. Nous abordons le problème de consommation de bande passante sous deux angles : l'inefficacité de la recherche et les transferts de fichiers longue distance. Notre solution Locaware consiste à mettre en cache des index de fichiers avec des informations sur leurs localités. Elle fournit également un support efficace pour les requêtes par mots clés qui sont courantes dans ce genre d'applications. Dans une deuxième étape, nous élaborons une infrastructure CDN P2P qui permet à tout site populaire et sous-provisionné de distribuer son contenu, par l'intermédiaire de sa communauté d'utilisateurs intéressés. Pour un routage efficace, l'infrastructure Flower-CDN combine intelligemment différents types d'overlays avec des protocoles épidémiques tout en exploitant les intérêts et les localités des pairs. PetalUp-CDN assure le passage à l'échelle alors que les protocoles de maintenance garantissent la robustesse face à la dynamicité des pairs. Nous évaluons nos solutions au travers de simulations intensives ; les résultats montrent des surcoûts acceptables et d'excellentes performances, en termes de taux de hit et de temps de réponse.

Mots-clés: Systèmes Pair à Pair, distribution de contenu, intérêts, localités physiques

Discipline : Informatique
Spécialité : Bases de Données

Laboratoire : LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE.
UMR CNRS 6241. 2, rue de la Houssinière, BP 92 208 – 44 322 Nantes, CEDEX 3.

Contents

Contents i

List of Figures v

Introduction 1

1 Content Distribution in P2P Systems 7
- 1.1 Introduction . . . 7
- 1.2 Insights on Content Distribution Netwoks . . . 9
  - 1.2.1 Background on Web Caching . . . 9
  - 1.2.2 Overview of CDNs . . . 10
    - 1.2.2.1 Replication and Caching in CDN . . . 12
    - 1.2.2.2 Location and Routing in CDN . . . 13
  - 1.2.3 Requirements and Open Issues of CDN . . . 14
- 1.3 P2P Systems . . . 16
  - 1.3.1 Overview of P2P Systems . . . 16
  - 1.3.2 Unstructured Overlays . . . 17
    - 1.3.2.1 Decentralization Degrees . . . 17
    - 1.3.2.2 Decentralized Routing Techniques . . . 19
    - 1.3.2.3 Behavior under Churn and Failures . . . 21
    - 1.3.2.4 Strengths and Weaknesses . . . 22
  - 1.3.3 Structured Overlays . . . 22
    - 1.3.3.1 DHT Routing . . . 23
    - 1.3.3.2 Behavior under Churn and Failures . . . 27
    - 1.3.3.3 Strengths and Weaknesses . . . 27
  - 1.3.4 Requirements of P2P Systems . . . 28
- 1.4 Recent Trends for P2P Content Distribution . . . 29
    - 1.4.0.1 Trend 1: Locality-Based Overlay Matching . . . 30
    - 1.4.0.2 Trend 2: Interest-Based Topology Matching . . . 32
    - 1.4.0.3 Trend 3: Gossip Protocols as Tools . . . 34
    - 1.4.0.4 Trend 4: P2P Overlay Combination . . . 37
    - 1.4.0.5 Challenges to Be Met . . . 39

1.4.0.6 Discussion . . . 40
1.5 P2P Content Distribution Systems . . . 41
1.5.1 Overview . . . 41
1.5.2 P2P File Sharing . . . 43
1.5.2.1 Inherent Properties . . . 43
1.5.2.2 Indexing Approaches . . . 45
1.5.2.3 Discussion . . . 50
1.5.3 P2P CDN . . . 50
1.5.3.1 Insights into Caching and Replication . . . 51
1.5.3.2 Deployed Systems . . . 52
1.5.3.3 Centralized Approaches . . . 54
1.5.3.4 Unstructured Approaches . . . 54
1.5.3.5 Structured Approaches . . . 55
1.5.3.6 Discussion . . . 56
1.6 Conclusion . . . 58

2 Locality-Aware P2P File Sharing 61
2.1 Introduction . . . 61
2.2 Problem Definition . . . 63
2.2.1 P2P File Sharing Model . . . 63
2.2.2 Index Caching Model . . . 63
2.2.3 Problem Statement . . . 63
2.3 Locaware Design and Implementation . . . 64
2.3.1 Bloom Filters as Keyword Support . . . 64
2.3.1.1 Bloom Filters . . . 65
2.3.1.2 Maintaining a Bloom Filter for the Index cache . . . 65
2.3.2 Locaware Index Caching . . . 66
2.3.2.1 Locality-Awareness . . . 66
2.3.2.2 Locality-Aware Indexes . . . 66
2.3.2.3 Controlling the Cache Size . . . 67
2.3.3 Locaware Query Searching . . . 68
2.3.4 Storage and Bandwidth Considerations . . . 69
2.3.4.1 About Bloom Filters Usage . . . 69
2.3.4.2 About Locality-Awareness . . . 70
2.4 Performance Evaluation . . . 70
2.4.1 Evaluation Methodology . . . 71
2.4.2 Experimental Setup . . . 71
2.4.2.1 Configuring the P2P Network . . . 72
2.4.2.2 Configuring the Workload . . . 72
2.4.3 Experimental Results . . . 73
2.4.3.1 Search Traffic . . . 73
2.4.3.2 Success Rate . . . 73
2.4.3.3 Locality-Awareness . . . 74

2.4.4 Lessons Learnt . . . 75
2.5 Conclusion . . . 76

3 Locality and Interest Aware P2P CDN 77
3.1 Introduction . . . 77
3.2 Flower-CDN Overview and Preliminaries . . . 79
3.3 D-ring Model . . . 80
3.3.1 Key Management . . . 81
3.3.2 Directory Tools . . . 83
3.3.3 P2P Directory Service . . . 83
3.3.3.1 Query Processing . . . 84
3.3.3.2 Joining the Petal . . . 85
3.4 Petal Model . . . 85
3.4.1 Gossip-Based Management . . . 86
3.4.1.1 Gossip Tools . . . 86
3.4.1.2 Gossip Behavior . . . 87
3.4.1.3 Push Behavior . . . 88
3.4.2 Query Processing . . . 89
3.5 Discussion of Design Choices . . . 91
3.6 Cost Analysis . . . 91
3.7 Performance Evaluation . . . 96
3.7.1 Evaluation Methodology . . . 96
3.7.2 Trade off: Impact of gossip . . . 98
3.7.3 Hit ratio . . . 100
3.7.4 Locality-awareness . . . 101
3.7.5 Discussion . . . 102
3.8 Conclusion . . . 103

4 High Scalability and Robustness in a P2P CDN 105
4.1 Introduction . . . 105
4.2 PetalUp-CDN . . . 107
4.2.1 Problem Statement . . . 107
4.2.2 D-ring Architecture in PetalUp-CDN . . . 108
4.2.3 D-ring Evolution in PetalUp-CDN . . . 110
4.2.3.1 D-ring Expansion . . . 110
4.2.3.2 D-ring Shrink . . . 112
4.2.4 Petal Management in PetalUp-CDN . . . 113
4.3 Robustness Under Churn . . . 114
4.3.1 Maintenance of Connection between D-ring and Petals . . . 114
4.3.2 Maintenance of D-ring . . . 115
4.3.2.1 Failures and Leaves . . . 115
4.3.2.2 Joins and Replacements . . . 116
4.4 Performance Evaluation . . . 117

4.4.1 Evaluation Methodology . . . 117
4.4.2 Robustness to churn . . . 119
4.4.3 Scalability . . . 120
4.4.3.1 Flower-CDN . . . 120
4.4.3.2 PetalUp-CDN . . . 121
4.4.4 Discussion . . . 122
4.5 Conclusion . . . 123

5 Deployment of Flower-CDN **125**
5.1 Introduction . . . 125
5.2 Flower-CDN Browser Extension . . . 126
5.2.1 Configuration . . . 127
5.2.2 Connection with Flower-CDN network . . . 128
5.3 Flower-CDN Implementation . . . 129
5.3.1 Global Architecture . . . 129
5.3.1.1 DHT-based Applications . . . 129
5.3.1.2 Flower-CDN . . . 129
5.3.2 Implementation Architecture . . . 131
5.3.2.1 Components . . . 131
5.3.2.2 Components at Work . . . 135
5.4 Conclusion . . . 136

Conclusion **139**

Bibliography **143**

A Résumé Étendu **155**

List of Figures

1.1 Web caching. . . . 10
1.2 Overview of a CDN. . . . 11
1.3 Akamai example. . . . 12
1.4 P2P overlay on top of the Internet. . . . 17
1.5 Types of unstructured P2P overlays. . . . 18
1.6 Blind routing techniques of unstructured overlays. . . . 21
1.7 Tree routing geometry. . . . 24
1.8 Hypercube routing geometry. . . . 25
1.9 Ring routing geometry. . . . 26
1.10 Locality-aware construction of CAN 32
1.11 Peer A gossiping to Peer B. . . . 35
1.12 How a P2P system can leverage gossiping. . . . 36
1.13 A two-layers DHT overlay [NT04]. . . . 39
1.14 P2P infrastructure for content distribution. . . . 43
1.15 Example of routing indices [CGM02]. . . . 46
1.16 Uniform index caching. . . . 48
1.17 Selective index caching. Case of DiCAS 49
1.18 CoralCDN hierarchy of key-based overlays [FFM04]. . . . 53
1.19 DHT strategies in a P2P CDN. . . . 56
1.20 Kache soft state at a peer. . . . 57

2.1 A Bloom filter sample 65
2.2 Locaware index caching. . . . 67
2.3 Search traffic evolution. . . . 73
2.4 Success rate evolution. . . . 74
2.5 Transfer distance evolution. . . . 75
2.6 Distribution of locality-aware file transfers 75

3.1 Flower-CDN architecture. . . . 80
3.2 Peer ID structure in D-ring. . . . 81
3.3 D-ring distribution of keys. . . . 82
3.4 New client on D-ring. . . . 84

3.5 Impact of petal size and probability on the number of rounds required to spread the rumor in the petal. . . . 94
3.6 Impact of petal size and probability on the number of messages required to spread the rumor in the petal. . . . 94
3.7 Impact of probability p_S on the number of views exchanged to spread the rumor. 95
3.8 Hit ratio evolution in static environment. . . . 100
3.9 Lookup latency in static environment. . . . 101
3.10 Transfer distance in static environment. . . . 102

4.1 Peer ID structure in D-ring of PetalUp-CDN. . . . 108
4.2 PetalUp-CDN architecture. . . . 109
4.3 Hit ratio evolution in dynamic environment. . . . 119
4.4 Query distribution in dynamic environment. . . . 120
4.5 PetalUp-CDN vs. Flower-CDN performance and overhead. . . . 123

5.1 Flower-CDN extension within the web browser. . . . 126
5.2 Configuration of Flower-CDN. . . . 128
5.3 DHT-based application global architetcure. . . . 130
5.4 Flower-CDN global architecure 130
5.5 Flower-CDN implementation architecture. . . . 132
5.6 Scenario 1: $processQuery(q)$ at content protocol 137
5.7 Scenario 2: $processAnswer(q)$ at content protocol 138

Introduction

Motivation

In the last decade, Web 2.0 [O'R05] has brought a paradigm shift in how people use the Web. Before this Web evolution, users were merely passive consumers of content that is provided to them by a selective set of websites. In a nutshell, Web 2.0 has offered an "architecture of participation" where individuals can participate, collaborate, share and create content. Web 2.0 applications deliver a service that gets better the more people use it, while providing their own content and remixing it with others' content. Today, there are many emerging websites that have helped to pioneer the concept of participation in Web 2.0. Popular examples include the online encyclopedia Wikipedia that enables individuals to create and edit content (articles), social networking sites like Facebook, photo and video sharing sites like YouTube and Flickr as well as wikis and blogs. Social networking is even allowing scientific groups to expand their knowledge base and share their theories which might otherwise become isolated and irrelevant [LOZB96].

With the Internet reaching a critical mass of users, Web 2.0 has encouraged the emergence of *peer-to-peer* (P2P) technology as a new communication model. The P2P model stands in direct contrast to the traditional client-server model, as it introduces symmetry in roles, where each peer is both a client and a server. Whereas a client-server network requires more investment to serve more clients, a P2P network pools the resources of each peer for the common good. In other terms, it exhibits the "network effect" as defined by economists [KS94]: the value of a network to an individual user scales with the total number of participants. In theory, as the number of peers increases, the aggregate storage space and content availability grow linearly, the user-perceived response time remains constant, whereas the search throughput remains high or even grows. Therefore, it is commonly believed that P2P networks are naturally suited for handling large-scale applications, due to their inherent self-scalability.

Since the late 1990s, P2P technology has gained popularity, mainly in the form of file sharing applications where peers exchange multimedia files. Some of the most popular P2P file sharing protocols include Napster [CG01], Freenet [CMH+02], Gnutella [Gnu05], BitTorrent [PGES05] and eDonkey2000. According to several studies [SGD+02], P2P file sharing accounts for more traffic than any other application on the Internet. Despite the emergence of sophisticated P2P network structures, file-sharing communities favor

unstructured networks for their high flexibility. File search commonly relies on blindly flooding the query over the P2P network, without any knowledge about the file location. Flooding mechanism has several attractive features such as simplicity, reliability and flexibility in expressing a query rather than strictly requiring the exact filename. However, it suffers from high bandwidth consumption because of its search blindness and its message redundancy. Many efforts have been done to tackle this problem which severely threatens the scalability of P2P file sharing networks. Along these lines, index caching has been proposed to incorporate indexing information in a simple and practical way. The main concept is to cache query responses in the form of indexes, on their way back to the query originator. Existing techniques [PH03, Sri01, WXLZ06] exhibit salient limitations because they trade either storage efficiency and/or query flexibility for search efficiency. Most importantly, they perform random file transfers between peers, totally ignoring their physical proximity and therefore increasing costs and response times unnecessarily. This critical issue has implications for user experience and Internet scalability [RFI02] and needs to be resolved to ensure the deployment of P2P file sharing.

In the course of time, P2P collaboration has extended well beyond simple file sharing. As Web 2.0 users are becoming more actively involved, P2P networks have enabled the creation of large-scale communities that cooperatively manage the content of their interest. The success of Wikipedia attests that as a mode of article production, P2P-style collaboration can succeed and even operate with an efficiency that closed systems cannot compete with. Projects like computation sharing over a P2P network in SETI@home [ACK+02] demonstrate that people are willing to share their resources to achieve common benefits. We focus on content sharing in P2P networks where large numbers of users connect to each other in a P2P fashion in order to request and provide content.

Under the Web 1.0 context, the content of web-servers is distributed to large audiences via *Content Distribution Networks* (CDN) [BPV08]. The main mechanism is to replicate popular content at strategically placed and dedicated servers. As it intercepts and serves the clients' queries, a CDN decreases the workload on the original web-servers, reduces bandwidth costs, and keeps the user-perceived latency low. Given that the Web is witnessing an explosive growth in the amount of web content and users, P2P networks seem to be the perfect match to build low-cost infrastructures for content distribution. This is because they can offer several advantages like decentralization, self-organization, fault-tolerance and scalability. In a P2P system, users serve each other queries by sharing their previously requested content, thus distributing the content without the need for powerful and dedicated servers.

However, due to the decentralized and open nature of P2P networks, making efficient use of P2P advantages is not a straightforward endeavor. Many challenges need to be overcome when building a P2P infrastructure that is as *scalable*, *robust* and high *performing* as commercial CDNs.

One major issue with any P2P system is the mismatch between the P2P network and the underlying IP-level network, which has two strong negative impacts. First, it can dramatically increase the consumption of network resources which limits the system scalability [RFI02]. Second, it can severely deteriorate the performance by increasing user-

perceived latency. For an efficient collaboration and a good quality of service, users should be able to access nearby content and communicate with peers close in locality. For this, the P2P network needs to incorporate some *locality-awareness* which refers to information about the physical location of peers and content. Previous efforts [DMP07, MPDJP08] on distributed and locality-aware algorithms have motivated us to deepen our investigation on this issue given the potential performance gains.

Another concern is that peers are not dedicated servers but autonomous and volunteer participants with their own heterogeneous interests. Peers unexpectedly fail, frequently join and leave the network by thousands [SR06]. Furthermore, they cannot be charged with heavy workloads or forced to contribute against their interests. Under these conditions, it is hard to ensure reliability since a peer departure can cause content or performance loss. Furthermore, scalability is constrained by efficient load balancing over peers. The challenge is thereby to cope with the autonomy of peers and efficiently maintain the network under their dynamicity so that it does not affect the system performance in processing queries and serving content.
In the P2P literature, several approaches like [IRD02, WNO+02, RY05, FFM04] have been proposed that build a P2P CDN. However, they usually compromise one requirement for another. In short, they are typically confronted with the trade-off between autonomy and reliability, or between quality of service and maintenance cost [DGMY02]. Some of them can achieve high reliability by reducing peer autonomy, while others can offer a good performance and quality of service for a high maintenance cost. Obviously, there is still room for improvement. Most importantly, the existing P2P CDNs lack of effective scalability as they operate on small scales.

Contributions

The goal of this thesis is to contribute to the development of novel and efficient P2P infrastructures for content distribution. In short, our work has evolved as follows. First, we have focused on P2P file sharing which can be considered as the simplest form of content distribution. This helped us make our first steps in exploring locality-awareness as a strong requirement and a significant source of gains. In addition, we have made a first attemp in dealing with the autonomous behaviour of peers and leveraging the inherent properties. Second, we have switched to more sophisticated collaborations and aimed at building a pure P2P infrastructure that can provide the scalability, reliability and performance of a commercial CDN with much lower costs.

More precisely, our contributions in this thesis are the following.

First, we survey content distribution systems which can range from file sharing to more elaborate systems that create a distributed infrastructure for organizing, indexing, searching and retrieving content. We shed light on the requirements and open issues of traditional content ditribution techniques, in particular commercial CDNs. Then, we give a comprehensive study of P2P systems from the perspective of content sharing and identify the design requirements that are crucial to make efficient use of P2P advantages.

We also present the recent trends and their challenges that can improve the performance of P2P content distribution. Finally, we discuss existing P2P CDNs and evaluate them according to the previously identified requirements. This analysis allows us to identify the requirements that our solutions should provide and the challenges that we might encounter.

Our second contribution targets file sharing in unstructured P2P networks. For this, we propose Locaware [DPV07,DP09], a new approach that tackles the existing limitations in P2P file sharing. It provides locality-aware and selective index caching in order to efficiently reduce unnecessary bandwidth consumption. Basically, a peer intercepts query responses and selectively caches several indexes per file, along with information about their physical locations. Thus, a peer can answer a query by providing several possibilities, which improves file availability and enables the selection of a file copy close to the query originator. Moreover, Locaware combines its indexing scheme with a query routing technique that provides some expressiveness and flexibility in the query formulation. In short, indexes are compactly summarized using Bloom Filters [Blo70] and then sent to the neighbors. The simulation results demonstrate that Locaware can limit wasted bandwidth and reduce network resource usage. They motivate us to elaborate more on Bloom filters and locality-awareness, in order to achieve greater performance improvement. On the one hand, the impact of locality-awareness could be more significant and its benefits intensified if exploited in query routing. On the other hand, Bloom Filters could be explored for more sophisticated search and caching techniques.

Our third contribution consists in building a P2P CDN, called Flower-CDN [DPK09a, DPK09d], that does not require dedicated or powerful servers. Flower-CDN distributes the popular content of any under-provisioned website by strictly relying on the community of users interested in its content. To achieve this, it takes into account the interests and localities of users, and accordinly organizes peers and serves queries. Flower-CDN adopts a novel and hybrid architecture that combines the strengths of the two types of P2P networks, i.e., structured and unstructured. It relies on a P2P directory service called D-ring, that is built and managed according to the interests and localities of the peers providng its services. D-ring helps new participants to quickly find peers in the same locality that are interested in the same website. Peers with respect to the same locality and website form together a cluster overlay called *petal*, to enable an efficient collaboration. Within a petal, peers use Bloom filters and gossip protocols [EGKM04] to exchange information about their contacts and content, allowing Flower-CDN to maintain accurate information despite dynamic changes. We use this two-layered infrastructure consisting of D-ring and the petals for a locality-aware query routing and serving. D-ring ensures a reliable access for new clients, whereas petals allow them to subsequently perform locality-aware searches and provide them close-by content. Thus, most of the query routing takes place within a locality-based cluster leading to short response times and local data transfer. Our simulation results show that Flower-CDN achieves significant gains of locality-awareness with limited overhead.

Our fourth contribution aims at providing our P2P CDN with high scalability and robustness under large scale and dynamic participation of peers. Thus, we propose

PetalUp-CDN [DPK09b], which dynamically adapts Flower-CDN to increasing numbers of participants in order to avoid overload situations. In short, PetalUp-CDN enables D-ring to progressively expand to manage larger petals so that all the participants share the workload rather evenly. In addition, we maintain our P2P CDN in face of high churn and failures, by relying on low-cost gossip protocols. Our maintenance protocols [DPK09c] preserve the locality and interest aware features of our achitecture and enables fast and efficient recovery. Based on our extensive empirical analysis, we show that our approach leverages larger scales to achieve higher improvements. Furthermore we conclude that Flower-CDN can maintain an excellent performance under a highly dynamic participation of peers.

Our fifth contribution address the deployment of Flower-CDN for public use. We show how to transparently integrate Flower-CDN into the user's web browser and dynamically configure it according to the interests of the user. We design the implementation architecture that covers security and privacy issues in a simple and practical manner.

Thesis Organization

The thesis is organized as follows. In Chapter 1, we provide a literature review of the state-of-the-art for content distribution and P2P systems. First we give more insight into traditional CDNs and their requirements which are needed for the design of novel and cheaper alternatives. Then we present P2P systems and identify their fundamental requirements and challenges. Finally, we introduce the existing P2P solutions for content distribution and enlighten their open issues.

Chapter 2 is dedicated to Locaware, our locality-aware solution for P2P file sharing. The first part of the chapter recall the context of P2P File Sharing and index caching in order to clearly define the problem. The second part focuses on the design and implementation of Locaware, and finally its performance evaluation through simulations.

In Chapter 3, we present Flower-CDN, our proposed P2P infrastructure that exploits localities and interests of peers for efficient content distribution. After a quick overview, we explore the D-ring model with its different features and services. Then we describe the Petal model and its gossip-based management. In addition, we discuss and argument our design choices, and analyse the costs of our solution. We conclude with our extensive simulation methodology and results.

Chapter 4 addresses the scalability and robustness of our P2P CDN. We present the highly scalable version of Flower-CDN, PetalUp-CDN, with its design and dynamic construction. Then we discuss the maintenance protocols that ensure the robustness of Flower-CDN and PetalUp-CDN under churn. Finally, we present our empirical analysis for robustness and scalability.

In Chapter 5, we give guidelines on the deployment of Flower-CDN for public use and discuss implementation details.

Finally, we conclude and highlight future directions of research.

List of Publications

International Journals

- Vidal Martins, Esther Pacitti, Manal El Dick, and Ricardo Jimenez-Peris. Scalable and topology-aware reconciliation on P2P networks. Journal of Distributed and Parallel Databases, 24(1-3):1-43, 2008.

International Conferences

- Manal El Dick, Esther Pacitti, and Bettina Kemme. A highly robust P2P-CDN under large-scale and dynamic participation. In *Proceedings of the 1st International Conference on Advances in P2P Systems (AP2PS)*, pages 180-185, 2009.

- Manal El Dick, Esther Pacitti, and Bettina Kemme. Flower-CDN: a hybrid P2P overlay for efficient query processing in CDN. In **Proceedings of the 12th ACM International Conference on Extending Database Technology (EDBT)**, pages 427-438, 2009.

- Manal El Dick, Vidal Martins, and Esther Pacitti. A topology-aware approach for distributed data reconciliation in P2P networks. In *Proceedings of the 13th International Conference on Parallel and Distributed Computing (Euro-Par)*, pages 318-327, 2007.

International Workshops

- Manal El Dick, Esther Pacitti, and Bettina Kemme. Leveraging P2P overlays for large-scale and highly robust content distribution and search. In *Proceedings of the VLDB PhD Workshop*, 2009.

- Manal El Dick and Esther Pacitti. Locaware: Index caching in unstructured P2P-file sharing systems. In *Proceedings of the 2nd ACM International Workshop on Data Management in Peer-to-peer systems (DAMAP)*, page 3-, 2009.

- Manal El Dick, Esther Pacitti, and Patrick Valduriez. Location-aware index caching and searching for P2P systems. In *Proceedings of the 5th International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, 2007.

National Conferences

- Manal El Dick, Esther Pacitti, and Bettina Kemme. Un réseau pair-à-pair de distribution de contenu exploitant les intérêts et les localités des pairs. In *Actes des 23èmes Journées Bases de Données Avancées*, (BDA) (Informal Proceedings), pages 407-388, 2009.

Chapter 1

Content Distribution in P2P Systems

Abstract. In order to define the problems we address in this thesis, the first chapter provides a literature review of the state-of-the-art for content distribution. In short, the contributions of this chapter are of threefold. First, it gives more insight into traditional Content Distribution Networks (CDN), their requirements and open issues. Second, it discusses P2P systems as a cheap and scalable alternative for CDN and extracts their design challenges. Finally, it evaluates the existing P2P systems dedicated for content distribution. Although significant progress has been made in P2P content distribution, there are still many open issues.

1.1 Introduction

The explosive growth of the Internet has triggered the conception of massive scale applications involving large numbers of users in the order of thousands or millions. According to recent statistics [ITU07], the world had 1.5 billion Internet users by the end of 2007. The client-server model is often not adequate for applications of such scale given its centralized aspect. Under this model, a content provider typically refers to a centralized web-server that exclusively serves its content (e.g., web-pages) to interested clients. Eventually, the web-server suffers congestion and bottleneck due to the increasing demands on its content [Wan99]. This substantially decreases the service quality provided by the web-server. In other terms, the web-server gets overwhelmed with traffic due to

a sudden spike in its content popularity. As a result, the website becomes temporarily unavailable or its clients experience high delays mainly due to long download times, which leaves them in frustration. That is why the World Wide Web is often pejoratively called World Wide Wait [Moh01].

In order to improve the Internet service quality, a new technology has emerged that efficiently delivers the web content to large audiences. It is called *Content Distribution Network* or *Content Delivery Network* (CDN) [BPV08]. A commercial CDN like Akamai[1] is a network of dedicated servers that are strategically spread across the Internet and that cooperate to deliver content to end-users. A content provider like Google and CNN can sign up with a CDN so that its content is deployed over the servers of the CDN. Then, the requests for the deployed content are transparently redirected to and handled by the CDN on behalf of the origin web-servers. As a result, CDNs decrease the workload on the web-servers, reduce bandwidth costs, and keep the user-perceived latency low. In short, CDNs strike a balance between the costs incurred on content providers and the QoS provided to the users [PV06]. CDNs have became a huge market for generating large revenues [iR08] since they provide content providers with the highly required *scalabiliy*, *reliability* and *performance*. However, CDN services are quite expensive, often out of reach for small enterprises or non-profit organizations.

The new web trend, Web 2.0, has brought greater collaboration among Internet users and encouraged them to actively contribute to the Web. *Peer-to-Peer* (P2P) networking is one of the fundamental underlying technologies of the new world of Web 2.0. In a P2P system, each node, called a *peer*, is client and server at the same time – using the resources of other peers, and offering other peers its own resources. As such, the P2P model is designed to achieve *self-scalability* : as more peers join the system, they contribute to the aggregate resources of the P2P network. P2P systems that deal with content sharing (e.g., sharing files or web documents) can be seen as a form of CDN, where peers share content and deliver it on each other's behalf [SGD+02]. The more popular the content (e.g., file or web-page), the more available it becomes as more peers download it and eventually provide it for others. Thus, the P2P model stands in direct contrast to traditional CDNs like Akamai when handling increasing amounts of users and demands. Whereas a CDN must invest more in its infrastructure by adding servers, new users bring their own resources into a P2P system. This implies that P2P systems are a perfect match for building cheap and scalable CDN infrastructures. However, making use of P2P self-scalability is not a straightforward endeavor because designing an efficient P2P system is very challenging.

This chapter aims at motivating our thesis contributions in the field of content distribution. For this purpose, it reviews the state-of-the-art for both traditional and P2P content distribution in order to identify the shortcomings and highlight the challenges.

Roadmap. The rest of this chapter is organized as follows. Section1.2 gives more insight into traditional CDNs and highlights their requirements which are needed for the design

[1] `http://www.akamai.com`

of novel and cheaper alternatives. Section 1.3 presents P2P systems and identifies their fundamental design requirements. Section 1.4 investigates the recent P2P trends that are useful for content distribution and identifies their challenges. Then, Section 1.5 deeply explores the state-of-art in P2P solutions for content distribution. It evaluates the existing approaches against the previously identifed requirements (for both P2P and CDN) and enlightens open issues.

1.2 Insights on Content Distribution Netwoks

Content distribution networks is an important web caching application. First, let us briefly review the different web caching techniques in order to position and understand the CDN technology. Then, we shed lights on CDNs, their requirements and their open issues.

1.2.1 Background on Web Caching

A web cache is a disk storage of predefined size that is reserved for content requested from the Internet (such as HTML pages and images)[2]. After an original request for an object has been successfully fulfilled, and that object has been stored in the cache, further requests for this object results in returning it from the cache rather than the original location. The cache content is temporary as the objects are dynamically cached and discarded according to predefined policies (further details in Section 1.2.2.1).

Web caching is widely acknowledged as providing three major advantages [CDF+98]. First, it reduces the bandwidth consumption since fewer requests and responses need to go over the network. Second, it reduces the load on the web-server which handles fewer requests. Third, it reduces the user-perceived latency since a cached request is satisfied from the web cache (which is closer to the client) instead of the origin web-server. Together, these advantages make the web less expensive and better performing.

Web caching can be implemented at various locations using *proxy servers* [Wan99, Moh01]. A *proxy server* acts as an intermediary for requests from clients to web-servers. It is commonly used to cache web-pages from other web-servers and thus intercepts requests to see if it can fulfill them itself. A proxy server can be placed in the user's local computer as part of its web browser or at various points between the user and the web-servers. Commonly, proxy caching refers to the latter schemes that involve dedicated servers out on the network while the user's local proxy cache is rather known as *browser cache*.

Depending on their placement and their usage purpose, we distinguish two kinds of proxies, *forward proxies* and *reverse proxies*. They are illustrated in Figure 1.1.

A *forward proxy* is used as a gateway between an organisation (i.e., a group of clients) and the Internet. It makes requests on behalf of the clients of the organisation. Then, it caches requested objects to serve subsequent requests coming from other clients of the organisation. Large corporations and Internet Service Providers (ISP) often set up

[2]Web caching is different from traditional caching in main memory that aims at limiting disk accesses

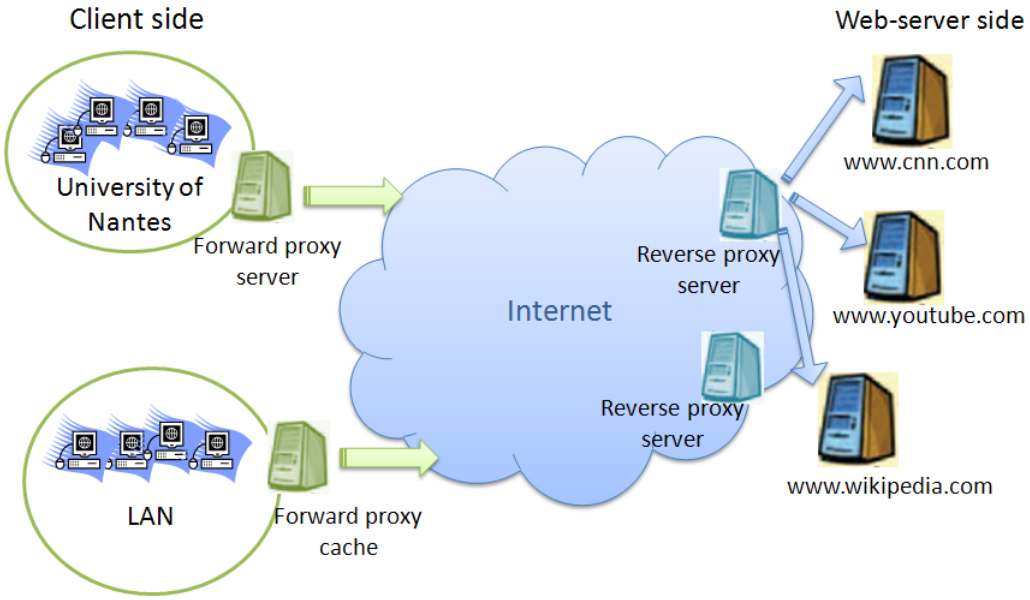


Figure 1.1: Web caching: different placements of proxy servers.

forward proxies on their firewalls to reduce their bandwidth costs by filtering out repeated requests. As illustrated in Figure 1.1, the university of Nantes has deployed a forward proxy that interacts with the Internet on behalf of the university users and handles their queries.

A *reverse proxy* is used in a network in front of web-servers. It is delegated the authority to operate on behalf of these web-servers, while working in close cooperation with them. Typically, all requests addressed to one of the web-servers are routed through the proxy server which tries to serve them via caching. Figure 1.1 shows a reverse proxy that acts on behalf of the web-servers of wikipedia.com, cnn.com and youtube.com by handling their received queries. A CDN deploys reverse proxies throughout the Internet and sells caching to websites that aim for larger audience and lower workload. The reverse proxies of a CDN are commonly known as *surrogate servers*.

1.2.2 Overview of CDNs

A CDN deploys hundreds of surrogate servers around the globe, according to complex algorithms that take into account the workload pattern and the network topology [Pen03]. Figure 1.2 gives an overview of a CDN that distributes and delivers the content of a web-server in the US.

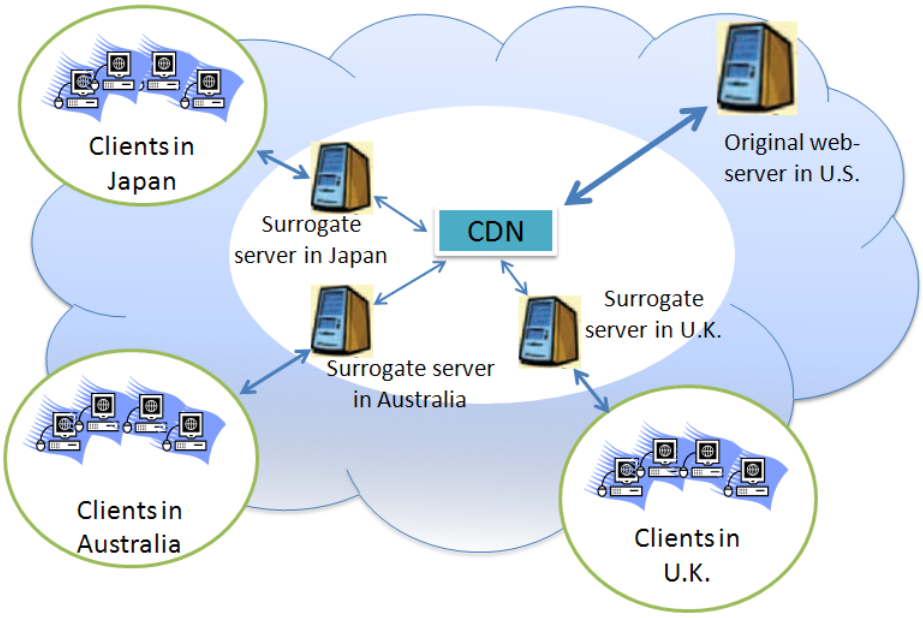


Figure 1.2: Overview of a CDN.

Examples of commercial CDNs are Akamai [3] and Digital Island [4]. They mainly focus on distributing static content (e.g., static HTML pages, images, documents, audio and video files), dynamic content (e.g., HTML or XML pages that are generated on the fly based on user specification) and streaming audio or video. Further, ongoing research aims at extending CDN technology to support video on demand (VoD) and live streaming. In this thesis, we mainly focus on static content. This type of content has a low frequency of change and can be easily cached; its freshness can be maintained via traditional caching policies [Wan99].

A CDN stores the content of different web-servers and therefore handles related queries on behalf of these web-servers. Each website selects specific or popular content and pushes it to the CDN. Clients requesting this content are then redirected to their closest surrogate server via *DNS redirection* or *URL rewriting*. The CDN manages the replication and/or caching of the content among its surrogate servers. These techniques are explained in more detail below.

The interaction between a user and a CDN takes place in a transparent manner, as if it is done with the intended origin web-server. Let us consider a typical user interaction with the well-known CDN, Akamai [Tec99], which mainly deals with objects embedded in a web-page (e.g., images, scripts, audio and video files). First, the user's browser sends a request for a web-page to the website. In response, the website returns the appropriate

[3] `http://www.akamai.com`
[4] `http://www.digitalisland.com/`

HTML page as usual, the only difference being that the URLs of the embedded objects in the page have been modified to point to the Akamai network. As a result, the browser next requests and obtains the embedded objects from an optimal surrogate server.

How is this transparent interaction achieved from a technical perspective? In the following, we investigate this issue by exploring CDN techniques for replication and caching on one hand, and location and routing on the other hand.

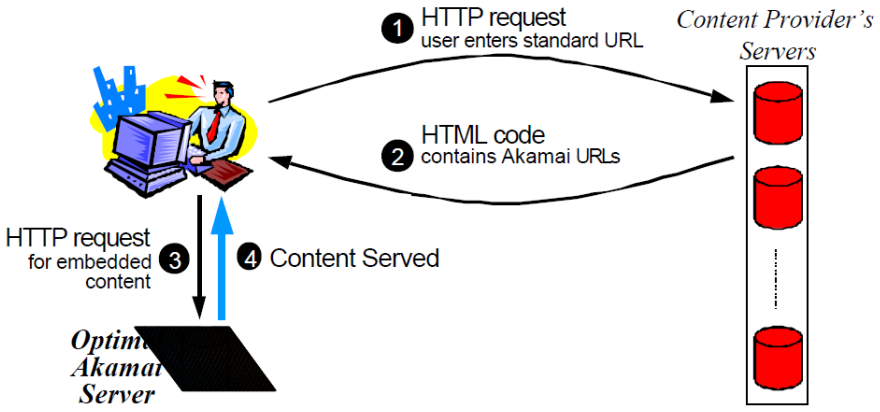


Figure 1.3: Typical user interaction with a website using Akamai services [Tec99].

1.2.2.1 Replication and Caching in CDN

According to [CMT01] and [SPV06], replication involves creating and permanently maintaining duplicate copies of content on different nodes to improve content availability. One the other hand, caching consists in temporarily storing passing by request responses (e.g., web-pages, embedded objects like images) in order to reduce the response time and network bandwidth consumption on future, equivalent requests. Note that web documents are typically accessed in read-only mode: requests read a document without changing its contents.

Replication. In a CDN, replication is typically initiated when the origin web-servers push content to any surrogate servers [PV06, BPV08]. The surrogate servers then manage the replication of the content among each other, either on-demand or beforehand.

In on-demand replication, the surrogate server that has received a query and experienced a cache miss, pulls the requested content from the origin web-server or other surrogate servers. In the latter case, it might use a centralized or distributed index to find a nearby copy of the requested content within the CDN.

Beforehand replication implies different strategies that replicate objects a priori and dynamically adjust their placement in a way that brings them closer to the clients and balances the load among surrogate servers [Pen03].

However, due to replication requirements in terms of cost and time, any replicas' placement should be static for a large amount of time.

Caching. Given that popularity of objects may fluctuate with time, some replicas may become redundant and unnecessary. This leads to unoptimized storage management at surrogate servers. That is why caching can be seen as an interesting alternative to replication, especially in cases where unpredictable numerous users have suddenly interest in the same content.

Content objects are dynamically cached and evicted from the cache according to cache replacement policies. More precisely, each cached object is assigned a cache expiration policy which defines how long it is fresh based on its own characteristics [Wan99]. Upon receiving a request for an object, the server first checks the freshness of its cached version before serving it. In case it has expired, the surrogate server checks with the origin server if the object has changed (by sending *a conditional GET (cGET)* request, e.g. *If-Modified-Since* request). Subsequently, the origin server either validates the cached copy or sends back a fresh copy. Since the cache has a limited storage size, the server might need to evict cached objects via one of the cache replacement policies that have been studied in [Wan99]. In the policy LRU, the rarely requested objects stored in the local cache are replaced with the new incoming objects. Additionally, the cache may regularly check for expired objects and evict them.

An evaluation of caching and replication as seperate approaches in CDNs is covered in [KM02], where caching outperforms but replication is still preferred for content availability and reliability of service. If replication and caching cooperate they may greatly fortify the CDN since both deal with the same problem but from a different approach. Indeed, [SPV06] has proven that potential performance improvement is possible in terms of response time and hit ratio if both techniques are used together in a CDN. CDNs may take advantage of the dynamic nature of cache replacement policies while maintaining static replicas for availability and reliability.

1.2.2.2 Location and Routing in CDN

To serve a query in a CDN, there are two main steps, server location and query routing. The first step defines how to select and locate an appropriate surrogate server holding a replica of the requested object whereas the second step consists in routing the query to the selected surrogate server. In several existing CDNs, these two steps are combined together in a single operation.

A query routing system uses a set of metrics in selecting the surrogate server that can best serve the query. The most common metrics include proximity to the client, bandwidth availability, surrogate load and availability of content. For instance, the distance between the client and a surrogate server can be measured in terms of *round-trip-time*(RTT) via the ping common tool.

Actually, each CDN uses its proprietary algorithms and mechanisms for location and routing and does not always reveal all the technology details. Here, we try to give a generic description of the mechanisms commonly used by CDNs, based on the materials

in [BPV08, Pen03]. The most common query routing technique are *DNS redirection* and *URL rewriting*.

DNS Redirection. CDNs can perform dynamic request routing using the Internet's *Domain Name System* (DNS). The DNS is a distributed directory service for the Internet whose primary role is to map *fully qualified domain names* (FQDNs) to IP addresses. For instance, hostname.example.com translates to 208.77.188.166. The DNS distributes the responsibility of assigning domain names and mapping those names to IP addresses over *authoritative name servers*: an *authoritative name server* is designated to be responsible for each particular domain, and in turn can designate other authoritative name servers for its sub-domains. This results in a hierarchical authority structure that manages the DNS. To determine an FQDN's address, a DNS client sends a request to its local DNS server which resolves it by recursively querying a set of authoritative DNS servers. When the local DNS server receives an answer to its request, it sends the result to the DNS client and caches it for future queries.

Normally, DNS mapping from an FQDNs to an IP address is static. However, CDNs use modified authoritative DNS servers to dynamically map each FQDN to multiple IP addresses of surrogate servers. The query answer may vary depending on factors such as the locality of the client and the load on the surrogate servers. Typically, the DNS server returns, for a request, several IP addresses of surrogate servers holding replicas of the requested object. The DNS client chooses a server among these. To decide, it may issue pings to the servers and choose based on resulting RTTs. It may also collect historical information from the clients based on previous access to these servers.

URL Rewriting. In this approach, the origin web-server redirects the clients to different surrogate servers by rewriting the URL links in a web-page. For example, with a web-page containing an HTML file and some embedded objects, the web-server would modify references to embedded objects so that they point to the CDN or more particularly to the best surrogate server. Thus, the base HTML page is retrieved from the origin web-server, while embedded objects are retrieved from CDN servers. To automate the rewriting process, CDNs provide special scripts that transparently parse web-page content and replace embedded URLs. URL rewriting can be proactive or reactive. In the proactive URL rewriting, the URLs for embedded objects of the main HTML page are formulated before the content is loaded in the origin server. In reactive approach involves rewriting the embedded URLs of a HTML page when the client request reaches the origin server.

1.2.3 Requirements and Open Issues of CDN

As introduced previously, a CDN has to fulfill stringent requirements which are mainly *reliability*, *performance* and *scalability* [PV06].

- **Reliability** guarantees that a client can always find and access its desirable content. For this, the network should be robust and avoid single point of failure.

- **Performance** mainly involves the response time perceived by end-users submitting queries. Slow response time is the single greatest contributor to clients abandoning web-sites [Tec04].
- **Scalability** refers to the adaptability of the network to handle more amounts of content, users and requests without significant decline in performance. For this, the network should prevent bottlenecks due to overload situations.

The reliability and performance of a CDN is highly affected by the mechanisms of content distribution as well as content location and routing. Content distribution defines how the content is distributed over the CDN and made available for clients. It mainly deals with the placement of content and involves caching and replication techniques in order to make the same content accessible from several locations. Thus, with these techniques, the content is located near to the clients yielding low response times and high content availability since many replicas are distributed. Content location and routing defines how to locate the requested content and route requests towards the appropriate and relevant servers. *Locality-awareness* refers to any topological information about the localities of peers to be able to evaluate their physical proximity. *Locality-awareness* is a top priority in routing mechanisms in order to find content close to the client in locality.

To expand and scale-up, CDNs need to invest significant time and costs in provisioning additional infrastructures [Tec04]. Otherwise, they would compromise the quality of service received by individual clients. Further, they should dynamically adapt their resource provisioning in order to address unexpected and varying workloads. This inevitably leads to more expensive services for websites. In the near future, the clients will also have to pay to receive high quality content (in some of today's websites like CNN.com, users have already started to pay a subscription to view videos). In this context, scalability will be an issue to deliver high quality content, maintaining low operational costs [BPV08].

Most recently, traditional CDNs [BPV08] have turned towards P2P technology to reduce investments in their own infrastructure, in the context of video streaming. The key idea is to dynamically couple traditional CDN distribution with P2P distribution. Basically, the CDN serves a handful of clients which in turn provide the content to other clients. Joost[5] and BitTorrent[6] are today's most representative CDN companies using P2P technology to deliver Internet television and video streaming, respectively.

To conclude this section, we observe that P2P technology is being progressively accepted and adopted as a mean of content distribution. The existing CDNs still depend –at least partly- on a dedicated infrastructure, which requires investment and centralized administration. If the CDN could rely on a cheap P2P infrastructure supported only by end-users, this would provide a cheap and scalable alternative that avoids the considerable costs. In the rest of this chapter, we further investigate the feasibility of pure P2P content distribution.

[5] `http://www.joost.com`

[6] The technology is called *BitTorrent DNA* (*Delivery Network Accelerator*). Available at `http://www.bittorrent.com/dna/`

1.3 P2P Systems

In the past few years, P2P systems have emerged as a popular way to share content. The most representative systems include Gnutella [Gnu05, JAB01, Jov00], BitTorrent[7] [PGES05] and Fastrack/Kazaa [LKR06]. The popularity of P2P systems is attested by the fact that the P2P traffic accounts for more than 70% of the overall Internet traffic according to a recent study[8].

The P2P model holds great promise for decentralized collaboration among widespread communities with common interests. This communal collaboration lies at the heart of the Web 2.0 paradigm and stems from the principle of resource sharing. By distributing storage, bandwidth and processing across all participating peers, P2P systems can achieve high scalability, that would otherwise depend on expensive and dedicated infrastructure.

Let us first give an overview of P2P systems by defining their main concepts then we can explore them in more detail.

1.3.1 Overview of P2P Systems

P2P systems operate on application-level networks referred to as *overlay networks* or more simply *overlays*. In other words, peers are connected via a logical overlay network superposed on the existing Internet infrastructure. When two peers are connected via a logical link, this implies that they know each other and can regularly share information across this link. We say that the peers are *neighbors* in the P2P network. Figure 1.4 shows a P2P overlay network where Peer A and Peer B are neighbors, independently of their Internet location. A P2P overlay network serves as an infrastructure for applications that wish to exploit P2P features. It relies on a *topology* and its associated *routing protocol*. The overlay topology defines how the peers are connected whereas the routing protocol defines how the messages are routed between peers. According to their degree of structure, P2P overlays can be classified into two main categories: *structured* and *unstructured*. Typically, they differ on the constraints imposed on how peers are organized and where shared objects are placed [QB06].

The P2P overlay has a direct impact on the performance, reliability and scalability of the system. Given that P2P networks operate in open and vulnerable environments, peers are continuously connecting and disconnecting, sometimes unexpectedly failing. The high rates of peer arrival and departure creates the effect of churn [SR06] and requires a continuous restructuring of the network core. For the purpose of reliability, the P2P overlay must be designed in a way that treats failures and churn as normal occurences. For the purpose of scalability, the P2P overlay should dynamically accommodate to growing numbers of participants. The performance of P2P systems refers to their efficiency in locating desirable content, which tightly depends on the P2P overlay, mainly on the routing protocol.

[7] `http://www.bittorrent.com/`

[8] Available at `http://www.ipoque.com/resources/internet-studies/internet-study-2008_2009`.

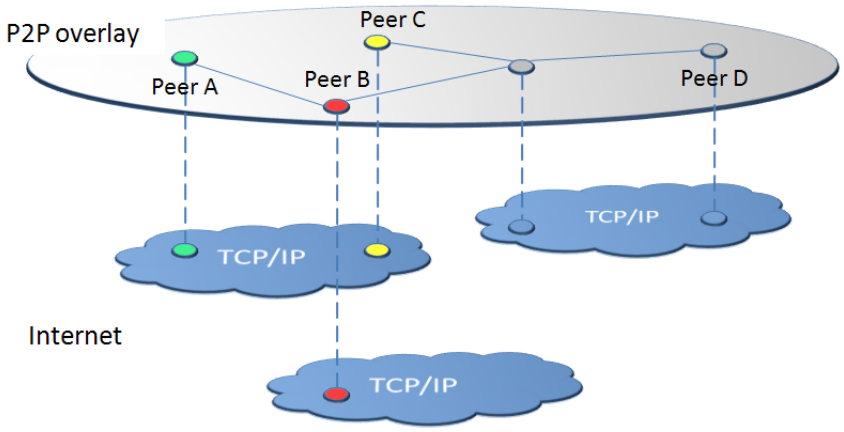


Figure 1.4: P2P overlay on top of the Internet infrastructure.

Basically, when a peer searches for a given object, it originates a query and routes it over the P2P overlay. Whenever a peer receives the query, it searches its local repository for the requested object. Eventually, the query reaches a peer that can satisfy the query and respond to the requester. The responder peer is either able to provide a copy of the requested object or has a pointer to the location of the object. Accordingly, the responder peer generates a *query response* that contains along with the object information (e.g., filename, id), the address of the provider peer. Upon receiving the query response, the query originator downloads a copy of the object from the provider peer.

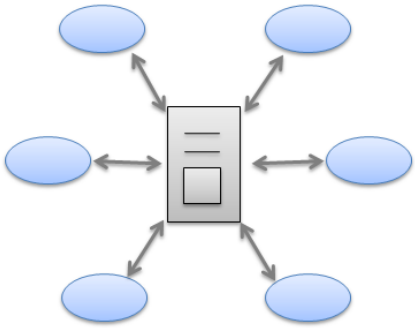
In the following, we present the two categories of P2P overlays, i.e., unstructured and structured overlays. For each category, we discuss its behavior under churn as well as its strengths and weaknesses. Then, we summarize by stating the requirements of P2P systems and accordingly compare both categories.

1.3.2 Unstructured Overlays

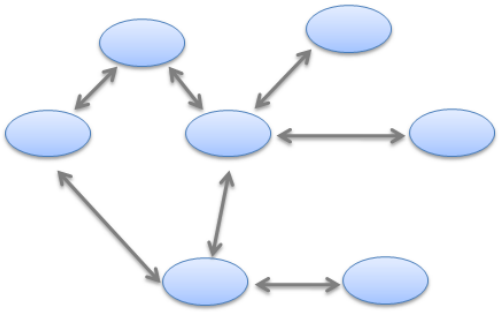
Often referred to as the first generation P2P systems, unstructured overlays remain highly popular and widely deployed in today's Internet. They impose lose constraints on peer neighborhood (i.e, peers are connected in an ad-hoc manner) and content placement (i.e., peers are free to place content anywhere) [QB06].

1.3.2.1 Decentralization Degrees

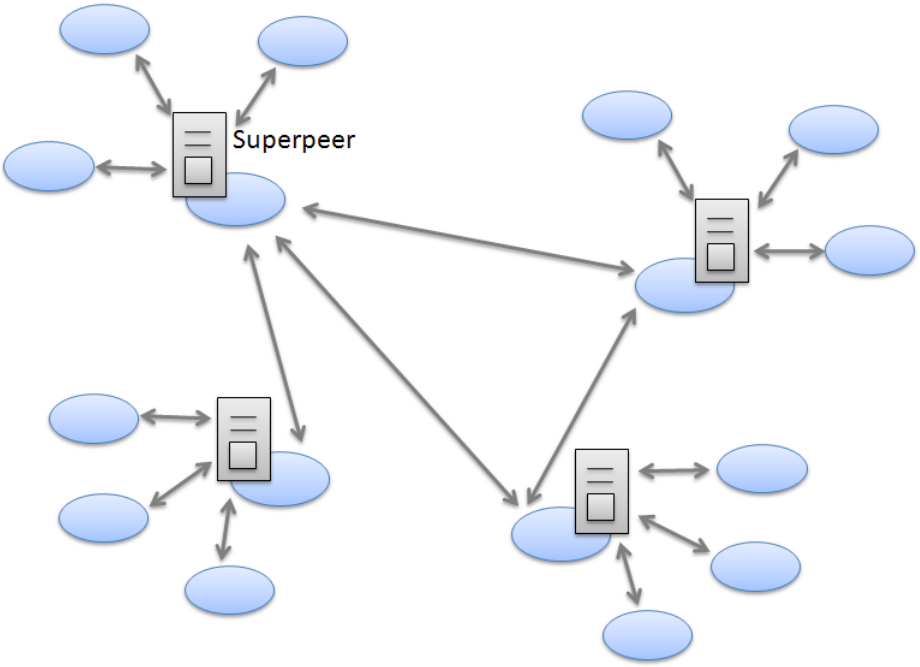
Although P2P systems are supposed to operate in a fully decentralized manner, in practice, we observe that various degrees of decentralization can be applied to the routing protocols of unstructured overlays. Accordingly, we classify unstructured overlays into three groups: *centralized*, *pure decentralized* and *partially decentralized with superpeers*.



(a) Centralized.



(b) Pure decentralized.



(c) Partially decentralized with superpeers.

Figure 1.5: Types of unstructured P2P overlays.

Centralized In these overlays, a central server is in charge of indexing all the peers and their shared content as shown in Figure 1.5a. Whenever a peer requests some content, it directly sends its query to the central server which identifies the peer storing the requested object. Then, the file is transferred between the two peers. The now-defunct Napster[9]

[9] `http://www.napster.com/`

[CG01] adopted such a centralized architecture.

Pure Decentralized In pure decentralized overlays, all peers have equal roles as shown in Figure 1.5b. Each peer can issue queries, serve and forward queries of other peers. Query routing is typically done by blindly flooding the query to neighbors. The flooding mechanism has been further refined, in a way that nowadays we find several variants of flooding like *random walks* and *iterative deepening*. These techniques are explained in more detail in Section 1.3.2.2. Of the many existing unstructured P2P systems, Gnutella [Gnu05, JAB01, Jov00] is one of the original pure decentralized networks.

Partially Decentralized with Superpeers In these overlays, high-capacity peers are assigned the role of superpeers, and each superpeer is reponsible of a set of peers, indexing their content and handling queries on their behalf. Superpeers are then organized in a pure decentralized P2P network and can communicate to search for queries (see Figure 1.5c). They can be dynamically elected and replaced in the presence of failures. Gnutella2 [SR02] is another version of Gnutella that uses superpeers; Edutella [NWQ+02] and FastTrack/Kazaa [LKR06] are also popular examples of hybrid networks.

The higher is the degree of decentralization, the more the network is fault-tolerant and robust against failures, because there will be no single point of failure due to the symmetry of roles. However, the higher is the degree of centralization, the more efficient is the search for content. Thus, the hybrid overlay strikes a balance between the efficiency of centralized search, and the load balancing and robustness provided by means of decentralization. Furthermore, it can take advantage of the heterogeneity of capacities (e.g., bandwidth, processing power) across peers. That is why recent generations of unstructured overlays are evolving towards hybrid overlays.

1.3.2.2 Decentralized Routing Techniques

In decentralized routing, *blind techniques* are commonly used to search for content in unstructured networks. *Blind techniques* route the query without any information related to the location of the requested object. A peer only keeps references to its own content, without maintaining any information about the content stored at other peers. Blind techniques can be grouped into three main categories: *breadth-first-search*, *iterative deepening* and *random walk*.

Breadth-First-Search (BFS). Originally, unstructured systems relied on the *flooding* mechanism which is more formally called *Breadth-First-Search* (BFS). Illustrated in Figure1.6a, the query originator sends its query Q to its neighbors, which in turn forward the message to all their neighbors except the sender and so on. The query is associated with a *Time-To-Live* (TTL) value, which is decreased by one when it travels across one hop in the P2P overlay. At a given peer, the message comes to its end if it becomes redundant (i.e., no further neighbors) or the TTL value becomes zero. Query responses follow the

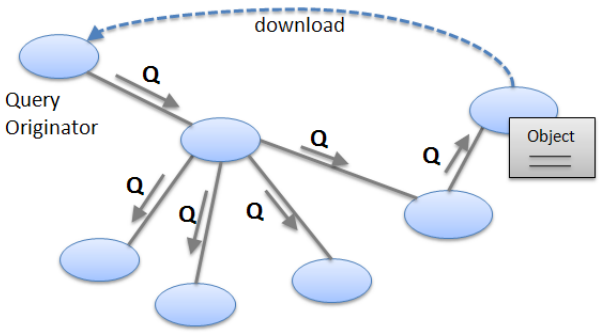
reverse path of the query, back to the requester peer. The main merits of this approach are its simplicity, reliability, and its high network coverage, i.e., a large number of peers could be reached within a small number of hops. However, measurements in [RFI02] have shown that although 95% of any two peers are less than 7 hops away, flooding generates 330 TB/month in a Gnutella network with only 50,000 nodes. This heavy traffic compromises the benefits of unstructured systems and drastically limits their scalability. The reasons behind the traffic burden of flooding are *blindness* and *redundancy*. First, a peer blindly forwards the query without any knowledge about how the other peers can contribute to the query. Second, a peer may receive the same query message multiple times because of the random nature of connections in an unstructured overlay. This can result in huge amounts of redundant and unnecessary messages.

In [KGZY02], *modified BFS* has been proposed in attempt to reduce the traffic overhead of flooding. Upon receiving a query, a peer randomly chooses a ratio of its neighbors to send or forward the query (see Figure1.6b). However, this approach may loose many of the good answers which could be found by BFS.

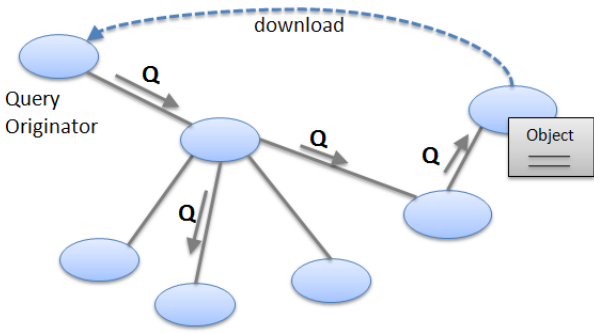
Iterative Deepening. This approach [YGM02, LCC+02] is also called *expanding ring*. The query originator performs consecutive BFS searches with successively larger TTL. A new BFS follows the previous one by expanding the TTL, if the query has not been satisfied after a predefined time period. The algorithm ends when the required number of answers is found or the predefined maximum TTL is reached. In case the results are not in the close neighborhood of the query originator, this approach does not address the duplication issue and adds considerable delay to the response time.

Random Walk. In the standard algorithm, the query originator randomly selects one of its neighbors and forwards the query to that neighbor. The latter, in turn, forwards the query to one randomly chosen neighbor, and so on until the query is satisfied. Compared to the basic BFS, this algorithm reduces the network traffic, but massively increases the search latency.

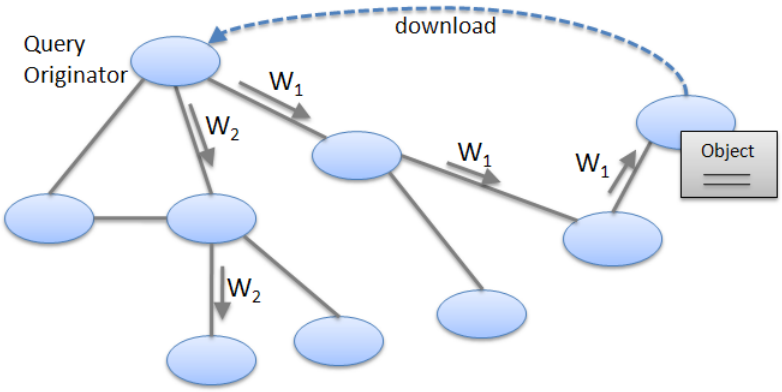
In the *k-walker random walk* algorithm [LCC+02], the query originator forwards k query messages to k randomly chosen neighbors (k is a value specified by the application). Each of these messages follows its own path, having intermediate peers forward it to one randomly chosen neighbor at each step. These query messages are also known as walkers and are shown in (Figure1.6c) as $W1$ and $W2$. When the TTL of a walker reaches zero, it is discarded. Each walker periodically contacts the query originator, asking whether the query was satisfied or not. If the response is positive, the walker terminates. This algorithm achieves a significant message reduction since it generates, in the worst case, $k*TTL$ routing messages, independently of the underlying network. Nevertheless, a major concern about this algorithm is its highly variable performance because success rates are highly variable and dependable on the network topology and the random choices made. In addition, the random walk technique does not learn anything from its previous successes or failures.



(a) BFS or flooding.



(b) Modified BFS.



(c) Random walk.

Figure 1.6: Blind routing techniques of unstructured overlays.

1.3.2.3 Behavior under Churn and Failures

It is well known that P2P networks are characterized by a high degree of churn [GDS+03]. Therefore, it is vital to examine the behavior of P2P networks in highly dynamic environments where peers join and leave frequently and concurrently.

The maintenance of unstructured overlays merely rely on the messages ping, pong and bye: pings are used to discover hosts on the network, pongs are replies to pings and contain in formation about the responding peer and other peers it knows about, and byes are optional messages that inform of the upcoming closing of a connection.

After joining the overlay network (by connecting to boostrap peers found in public databases), a peer sends out a ping message to any peer it is connected to. The peers send back a pong message identifying themselves, and also propagate the ping message to

their neighbors. When a peer gets in contact with a new peer, it can add it as a neighbor in its routing table in a straigthforward manner. A peer that detects the failure or leave of a neighbor simply removes it from its routing table. If a peer becomes disconnected by the loss of all of its neighbors, it can merely repeat the bootstrap procedure to re-join the network [CRB+03].

The measurements in [QB06] show that the bandwidth consumption due to maintenance messages is reasonably low in the unstructured Gnutella system. Peers joining and leaving the Gnutella network have little impact on other peers or on the placement of shared objects, and thus do not result in significant maintenance traffic.

To resume, there are few constraints on the overlay construction and content placement in unstructured networks: peers set up overlay connections to an arbitrary set of other peers they know, and shared objects can be placed at any peer in the system. The resulting random overlay topology and content distribution provides high robustness to churn [QB06]. Furthermore, the routing mechanism greatly rely on flooding which yields randomness and repetitiveness and thus more robustness. Given that a query takes several parallel routes, the disruption of some routes due to peer failures does not prevent the query from being propagated throughout the P2P network.

1.3.2.4 Strengths and Weaknesses

Unstructured P2P systems exhibit many simple yet attractive features, such as high flexibility and robustness under churn and failures. For instance, the freedom in content placement provides maximum flexibility in selecting policies for replication and caching.

Unstructured overlays are particularly used to support file-sharing applications for two main reasons. First, since they introduce no restritions on the manner to express a query, they are perfectly capable of handling *keyword search*, i.e., searching for files using keywords instead of the exact filenames. Second, file popularity derives a kind of natural file replication among peers, which induces high availability. Indeed, peers replicate the copies of files they request when they download them.

However, the main Achilles heel of unstructured systems are their blind routing mechanisms which incur severe load on the network and give no guarantees on lookup efficiency. Because of the topology randomness, a query search necessitates $O(n)$ hops (where n is the total number of peers), generates many redundant messages and is not guaranteed to find the requested object. Many studies such as [RFI02] and [Rit01] claim that the high volume of search traffic threatens the continued growth of unstructured systems. Indeed, the measurements in [RFI02] have shown that although 95% of any two peers are less than 7 hops away, flooding generates 330 TB/month in a Gnutella network with only 50,000 nodes.

1.3.3 Structured Overlays

The evolution of research towards structured overlays has been motivated by the poor scaling properties of unstructured overlays. Structured networks discard randomness and

impose specific constraints on the overlay topology [QB06]. They remedy to the blind search by tightly controlling the content placement. As a result, they provide an efficient, deterministic search: they can locate any object within a bounded number of hops.

More precisely, a structured overlay provides a distributed index scheme, by mapping content to locations (e.g., an object identifier is mapped to a peer address). To achieve this, objects and peers are assigned unique identifiers (respectively *keys* and *IDs*) from the same identifier space (e.g., hashing filename or url for an object and the IP address for a peer). Then, this identifier space is dynamically partitioned among peers, so that each peer is responsible for a specific key space partition. Accordingly, a peer stores the objects or pointers related to objects with respect to its key partition. The topology dictates for each peer a certain number of neighbors. The peer holds a routing table that associates its neighbors's identifiers to their IP addresses. Then a routing algorithm is defined to allow a deterministic key-based search. The main representative of structured overlays is the *Distributed Hash Table* (*DHT*) which is presented and dicussed in the following.

1.3.3.1 DHT Routing

At a fundamental level, DHTs can be viewed as content addressing and lookup engines. A DHT provides *content and peer addressing* via *consistent hashing* [KLL+97]. This technique enables a uniform hashing of values and thereby evenly places or maps content to peers. The addressing mechanism serves as a distributed and semantic-free index, because it gives information about the location of content based on hash-based keys.

The *lookup engine* of the DHT mainly consists in locating the target peer by means of routing over the overlay. The routing protocol tightly depends on the different implementations of DHT and more precisely the *routing geometries* [GGG+03]. Nonetheless, all routing protocols aim at providing efficient lookups as well as minimizing the routing state[10] that should be maintained at each peer. Most of them exhibit almost similar space and time complexity. That is, the routing table of peer contains at most $O(log\, N)$ entries and a lookup is normally performed in $O(log\, N)$ hops where N is the total number of nodes in the DHT [HHH+02].

The routing geometry mainly defines the manner in which neighbors and routes are established. According to [GGG+03], there are 6 basic types of routing geometries: *tree*, *hypercube*, *ring*, *butterfly*, *XOR* and *hybrid*. The main factor that distinguishes these geometries is the degree of flexibility they provide in the selection of neighbors and routes.

Neighbor selection refers to how the routing table entries of a peer are established, whereas route selection refers to how the next-hop can be determined in a routing process. Flexibility in the selection of neighbors and routes has a significant impact on the robustness and locality-awareness properties of the DHT-based system [GGG+03]. When allowing some freedom in the selection of neighbors and routes, one can choose neighbors and next routes, respectively, based on proximity. For instance, if the choice of neighbors is completely deterministic, it prevents the addition of features on top of the initial DHT proposal in order to achieve locality-aware routing tables. Further, flexible

[10] The routing state refers to the routing table of the peer.

selections interfere in failures because they describe how many alternatives are there for the neighbor or the next-hop in case they are down. For instance, if there are no options for a next-hop, or only a few, this may destabilize or interrupt the routing process, which can greatly increase the number of hops or/and the latency.

In the following, we look at 4 geometries, tree, hypercube, ring and hybrid and discuss their flexibility degrees.

Tree. Peer IDs are the leaves of a binary tree of height $log\, N$, where N is the number of nodes in the tree (see Figure 1.7). The responsible for a given key is the peer whose identifier has the highest number of prefix bits which are common with the key. The distance between any two peers is the height of their smallest common subtree. Each peer has $log\, N$ neighbors, such that the hth neighbor is at distance h from the peer. Let us consider the tree of height equal to 3 in Figure1.7. The peer with $ID = 010$ has the peer with $ID = 011$ as its 1st neighbor because their smallest common subtree is of height $h = 1$. Their IDs share a prefix of two bits and differ on the last bit. Similarly, the peer with $ID = 010$ has chosen the peer with $ID = 000$ as its 2nd because their smallest common subtree is of height $h = 2$. Their IDs share a prefix of one bit and differ on the two others. Routing is performed such that the prefix match between the target key

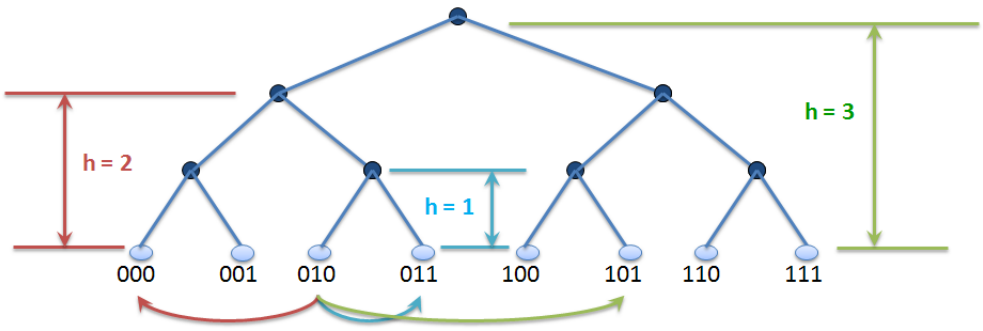
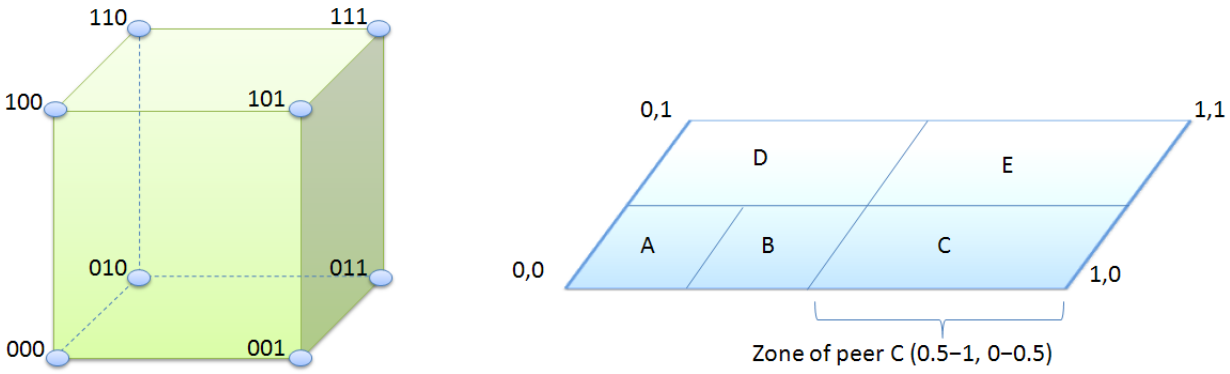


Figure 1.7: Tree routing geometry.

and the ID of the intermediate peer is increased by one at each hop, until reaching the responsible peer. The well-known DHT implementation Tapestry [ZHS+04] falls into this category.

The tree geometry gives a great deal of freedom to peers in choosing their neighbors; when choosing the ith neighbor, a peer has 2^{i-1} options. In the tree example, the peer with $ID = 010$ has $2^{2-1} = 2$ choices for its 2nd neighbor: the peer with $ID = 000$ and the peer with $ID = 001$ because they both belong to the subtree of height $h = 2$. However, this approach has no flexibility in the selection of routes: there is only one neighbor which the message must be forwarded to, i.e., this is the neighbor that has the most common prefix bits with the given key.

Hypercube. This geometry is based on a d-dimensional Cartesian coordinate space that is partitioned into a set of separate zones such that each peer is attributed one zone. Peers have unique identifiers with $log\,N$ bits, where N is the total number of peers of the hypercube. Each peer p has $log\,N$ neighbors such that the identifier of the ith neighbor and p differ only in the ith bit (see Figure 1.8a). Query routing proceeds by greedily forwarding the given key via intermediate peers to the peer that has minimum bit difference with the key. Thus, it is somehow similar to routing on the tree. The difference is that the hypercube allows bit differences to be reduced in any order while with the tree, bit differences have to be reduced in strictly left-to-right order. CAN [RFH+01] uses a routing geometry similar to hypercubes. Figure 1.8b shows a 2-dimensional $[0;1] * [0;1]$ coordinate space partitioned between 5 peers.



(a) Hypercube routing geometry.

(b) Hypercube-like structure of CAN.

Figure 1.8: Hypercube routing geometry.

There are $(log\,N)!$ possible routes between two peers, which provides high route flexibility. However, each peer in the coordinate space does not have any choice over its neighbours coordinates since adjacent coordinate zones in the coordinate space cannot change. Therefore, the high route selection flexibility provided by Hypercubes is at the price of poor neighbor selection flexibility.

Ring. Peers are arranged in a one-dimensional cyclic identifier space and ordered clockwise with respect to their identifiers. Chord [SMK+01] represents the prototypical DHT ring. In Chord, each peer has an m-bit identifier, and the responsible for a key is the first peer whose identifier is equal to or greater that the key. Each peer p has $log\,N$ neighbors such that the ith neighbor has a distance from the peer clockwise in the circle equal to $2^{i-1} mod\, N$. Hence, any peer can route a given key to its responsible in $log\,N$ hops because each hop cuts the distance to the destination by half. In Figure 1.9, the Chord peer with $ID = 8$ maintains 8 entries in its routing table called *finger table*.

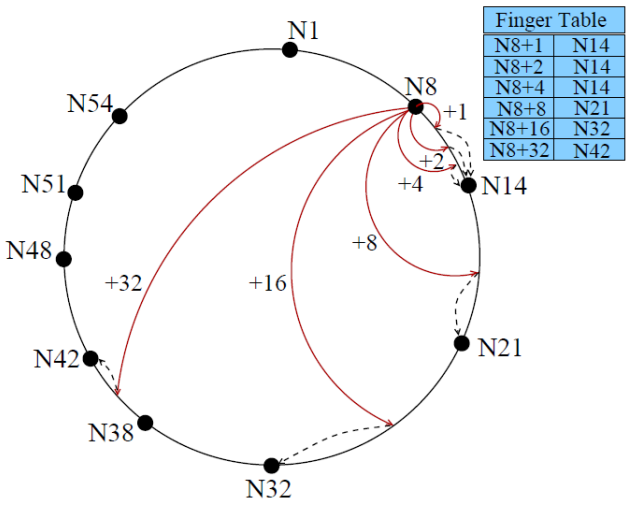


Figure 1.9: Ring routing geometry. Example of Chord with 10 peers and a peer ID of $m = 6$ bits.

Although Chord specifies the set of neighbors for each peer, the ring geometry does not necessarily needs such rigidity in neighbor selection. In fact, the $log\, N$ lookup bound is preserved as long as the ith neighbor is chosen from the range $[2^{i-1} mod\, N, 2^i mod\, N]$. This provides a great deal of neighbour selection flexibility because each peer would have 2^{i-1} options in selecting its ith neighbor. Moreover, to reach a destination, there are approximately $(log\, N)!$ possible routes. Therefore, the ring geometry also provides good route selection flexibility.

Hybrid. This geometry employs a combination of geometries. As a representative example, Pastry [RD01a] is a popular DHT implementation that combines the tree and ring geometries, aiming at a locality-aware routing. To achieve this, each peer maintains a routing table, a leaf set, and a neighborhood set. The routing table resembles the tree structure described previously, while the leaf set acts as the ring in routing. The neighborhood set is used to maintain locality properties. During a lookup process, a peer uses first the tree structure represented by its routing table, and only falls-back to the ring via its leaf set if routing in the tree fails. This is why Pastry provides flexibility in neighbor selection, similar to the tree geometry. However, the matter is more subtile with respect to route selection flexibility. Given that a peer maintains an ordered leafset, it is able to take hops between peers with the same prefix (i.e., between branches of the tree) and still retain the bits that were fixed previously; this however does not necessarily preserve the $log\, N$ bound on the number of hops.

1.3.3.2 Behavior under Churn and Failures

Preserving the topology constraints is crucial to guarantee the correctness of lookup in structured overlays. However, churn and failures highly affect DHTs.

When peer failures or leaves occur, they deplete the routing tables of the existing peers. Recovery algorithms are used to repopulate the routing tables with live peers, so that routing can continue unabated. The recovery can be reactive (upon detecting the failure of a peer referenced by the routing table) like in Pastry [RD01a] or periodic (upon regular time intervals in the background) like in Chord [SMK+01]. Basically, the peer exchanges entries from the routing table with peers from its routing table and accordingly update its routing table. After a single peer leaves the network, most DHTs require $O(log\, N)$ repair operations, i.e., updates of routing tables affected by the leave (N is the total number of peers) . When a peer unexpectedly fails, the DHT needs more time and effort to first detect the failure and then repair the affected routing tables. It should also notify the application to take specific measures so that the content held by failed peers is not lost. Several approaches have been proposed to prevent this problem, most notably the replication of content at peers with IDs numerically close to the content's key [RD01b].

When a new peer joins the overlay network, the DHT should detect the arrival and inform the application of the set of keys that the new peer is responsible for so that the relevant content is moved to its new home. Similarly to leaves and failures, the recovery algorithms should update the routing tables of the peers concerned by the new arrival.

However, if the churn rate is too high, the overhead caused by these repair operations can become dramatically high and could easily overwhelm peers [CRB+03].

Furthermore, recovery protocols take some time to repair and update the routing tables affected by joins and/or leaves. Given that new arrivals and departures are frequent in P2P environments, one must check the *static resilience* of a DHT [GGG+03], i.e., how well the DHT routing algorithm can perform before the overlay has recovered (before routing tables are restored and keys migrated to new homes). DHTs with low static resilience require much faster recovery algorithms to be similarly robust. In such DHTs, requests that fail in routing should retry the lookup after a pause. A DHT with routing flexibility provides high static resilience because it has many alternate paths available to complete a lookup (see Section 1.3.3.1).

The analysis in [RGRK04] has examined the effects of churn on existing DHT implementations and derived two main observations. A DHT may either fail to complete a lookup, or return inconsistent answers (e.g., return the address of a peer that is no more responsible for the requested key). On the other hand, a DHT may continue to return consistent answers as churn rates increase, but it can suffer from a substantial increase in lookup latency.

1.3.3.3 Strengths and Weaknesses

Structured overlays offer strong guarantees on lookup efficiency while limiting the routing overhead. In particular, the ring topology is the best compromise that supports many of the properties we desire from such overlays. They have been used in a

variety of applications such as content storage (e.g., OceanStore [KBC+00], Pastry [RD01b]), multicast services (e.g., Bayeux [ZZJ+01], Scribe [RKCD01]) or large-scale query processing engines (e.g., Pier [HHL+03]).

However, two criticisms arise against these overlays and constitue a major hurdle in the adoption of such systems. First, the tightly controlled topology requires high maintenance in order to cope with the frequent joins and leaves of peers. Moreover, studies [RGRK04] have shown that structured systems exhibit less than desirable performance under high churn rates because routing tables are affected and take time to be repaired. A second criticism concerns the limited flexibility provided by structured systems wrt. the autonomy of peers and the lookup functionality. Peers cannot freely choose their neighbors nor their responsabilities. Further, structured systems are designed in a way to provide key-based lookup which is convenient to exact-match queries. Their ability to support keyword searches and more complex queries is still an open issue.

Thus, structured overlays are the perfect match for applications that seek a scalable and guaranteed lookup but do not witness highly dynamic populations.

1.3.4 Requirements of P2P Systems

Based on this preliminary study on P2P systems, we observe that they introduce new requirements in respect of content sharing. The study in [DGMY02] identifies the following requirements:

- **Autonomy** defines the level of freedom granted to peers, mainly with respect to the placement of content. This is required to give peers proper incentives to cooperate. Indeed, it is usually not desired and rarely enabled to force storing content on peers.
- **Expressiveness** refers to the flexibility in query formulation. It should allow the user to describe the desired content at the level of detail that is appropriate to the target application.
- **Quality of service** has the most influence on user satisfaction. It can be defined with metrics like response time and hit ratio.
- **Efficiency** refers to the efficient use of resources of the P2P network (bandwidth, processing power, storage). Given the high rate of failures and churn, the maintenance protocol should neither compromise the gains with its overhead nor degrade the system performance. Also, efficiency implies that the routing protocol does not overload the network or the peers while not missing the available content.
- **Robustness** means that efficiency and quality of service are provided despite the occurrence of peer failures.
- **Security** is a major challenge given the open nature of P2P networks. With respect to content distribution, one of the most critical issues is the content authenticity which deals with the problem of distinguishing fake documents from original ones. We do not focus on this requirement in our study.

REQUIREMENTS	UNSTRUCTURED	STRUCTURED
AUTONOMY	free to choose neighbors and content	tight control on neighbors and content
EXPRESSIVENESS	keywords	exact-match
QUALITY OF SERVICE	no guarantees	deterministic
EFFICIENCY	efficient maintenance	efficient lookup
ROBUSTNESS	suitable for high churn	problems under high churn

Table 1.1: Comparison of P2P overlays.

Table 1.1 summarizes how the requirements are achieved by the two main classes of P2P networks. This is a rough comparison to understand the respective merits of each class. Obviously, there is room for improvement in each class of P2P networks. Regarding efficiency, structured systems provide a highly efficient lookup at the cost of a significant maintenance overhead, in opposition to unstructured systems.

Beyond this classical classification of P2P systems, there exist new trends in the P2P literature, that focus on other considerations and incur new challenges on the design of a P2P system. This is further investigated in Section 1.4.

1.4 Recent Trends for P2P Content Distribution

We have, so far, discussed P2P systems from a classical perspective. However, today's research is evolving towards more sophisticated issues about P2P systems, from the perspective of content distribution.

Recently, some have started to justify that unstructured and structured overlays are complementary, not competing. It is actually easy to demonstrate that depending on the application, one or the other type of overlay is clearly more adapted. In order to make use of the desirable features provided by each topology, there are efforts underway for combining both in the same P2P systems.

Further, the overlay can be refined through extracting and leveraging inherent structural patterns from P2P networks. These patterns can stem from the underlying physical network (e.g., physical proximity between peers) or be defined at the application layer (e.g., interest-based proximity between peers). Matching the overlay with the underlying physical network greatly contributes in reducing communication and data transfer costs as well as user-perceived latencies. Additionally, leveraging interests of peers to organize them can ease the search for content and guide the routing of queries.

Another recent trend is the usage of gossip protocols as a mean to build and maintain the P2P overlay. Gossiping is also used to feed the overlay with indexing information in order to facilitate content search.

In the following, we present in more detail the aforementioned trends. In Section 1.4.0.1, we detail locality-based overlay matching and the existing solutions along these lines. Then, we present interest-based overlay matching in Section 1.4.0.2. In Section 1.4.0.3, we introduce the usage of gossip protocols in P2P systems. Finally, we review the

existing approaches that combine several overlays in Section 1.4.0.4. Finally, we identify the major challenges to be met when aiming to achieve these new trends and accordingly discuss the aforementioned approaches.

1.4.0.1 Trend 1: Locality-Based Overlay Matching

As introduced in Section 1.3, the overlay topology defines application-level connections between peers and completely abstracts all features about the underlying physical network (e.g., IP level). In other terms, the neighborhood of a node is set without much knowledge of the underlying physical topology, causing a mismatch between the P2P overlay and the physical network. Figure1.4 clearly illustrates the mismatch between a P2P overlay and the underlying Internet. As an example, peer A has peer B as its overlay neighbor while peer C is its physical neighbor. This can lead to inefficient routing in the overlay because any application-level path from peer A towards the nearby peer C traverses distant peers.

More precisely, the scalability of a P2P system is ultimately determined by its efficient use of underlying resources. The topology mismatch problem imposes substancial load on the underlying network infrastructure, which can eventually limit the scalability [RFI02]. Furthermore, it can severely deteriorate the performance of search and routing techniques, typically by incurring long latencies and excessive traffic. Indeed, many studies like [SGD+02] have revealed that the P2P traffic contributes the largest portion of the Internet traffic and acts as a leading consumer of Internet bandwidth. Thus, a fundamental challenge is to incorporate IP-level topological information in the construction of the overlay in order to improve routing performance. This topological information could also be used in the selection of close-by search results to ensure a good user experience. Topological information refers to locality-awareness because it aims at finding peers close in locality. Below, we present the main representative approaches that propose locality-based matching schemes.

Physical Clustering. In [KWX01], clustering has been used to group physically close peers into clusters. The approach relies on a centralized engine to identify clusters of close peers under common administrative control. To achieve this, the central server uses IP-level routing information which is not directly available to end-user applications. Thus, the main drawbacks of this approach are the centralized topology control and the topological information itself, which prevents it from being scalable and robust to churn.

In the context of application-level multicast and media streaming, many solutions aim at constructing a locality-aware overlay because of the strong requirements on the delivery quality. The NICE protocol [BBK02] builds a hierarchy of clusters rooted at the source, whith close peers belonging to the same part of the hierarchy. However, maintaining the hierarchy under churn may incur high overhead and affect performance.

LTM Technique. The LTM (*Location-aware Topology Matching*) technique [LXL+05] targets unstructured overlays. It dynamically adapts connections between peers in a completely decentralized way. Each peer issues a detector in a small region so that the

peers receiving the detector can record the relative delay. Accordingly, a receiving peer can detect and cut most of the inefficient logical links and add closer peers as neighbors. However, this scheme operates on long-time scales where the overlay is slowly improved over time. Given that participants join and leave on short time-scales, a solution that operates on long-time scales would be continulally reacting to fluctuating peer membership without stabilizing.

Locality-Aware Structured Overlays. While the original versions of structured overlays did not take locality-awareness into account, almost all of the recent versions make some attempt to deal with this primary issue. [RSS02] identifies three main approaches.

- *Geographic layout*: the peer IDs are assigned in a manner that ensures that peers that are close in the physical network are close in the peer ID space.
- *Proximity routing*: the routing tables are built without locality-awareness but the routing algorithm aims at selecting, at each hop, the nearest peer among the ones in the routing table. For this, flexibility in routing selection is required as layed out in Section 1.3.3.1.
- *Proximity neighbor selection*: the construction of routing tables takes locality-awareness into account. When several candidate peers are available for a routing table entry, a peer prefers the one that is close in locality. To achieve this, flexibility in neighbor selection is required as pointed out in Section 1.3.3.1.

Pastry [RD01a] and Tapestry [ZHS+04] adopt proximity neighbor selection. In order to preferentially select peers and fill routing tables, these systems assume the existence of a function (e.g., round-trip-time RTT) that allows each peer to determine the physical distance between itself and any another peer. Although this solution leads to much shorter query routes, it requires expensive maintenance mechanisms under churn. As peers arrive and leave, routing tables should be repaired and updated. Without timely repairing, the overlay topology will diverge from optimal condition as inefficient routes gradually accumulate in routing tables.

A design improvement [RHKS02] of CAN aims at achieving geographic layout. It relies on a set of well-known landmarks spread across the network. A peer measures its round-trip time (RTT) to the set of landmarks and orders them by increasing latency (i.e., network distance). The logical space of CAN is then divided into bins such that each possible landmarks ordering is represented by a bin. Physically close nodes are likely to have the same ordering and hence will belong to the same bin. This is illustrated in Figure 1.10. We have 3 landmarks (i.e., L1, L2, and L3) and, accordingly, the CAN coordinate space is divided into 6 bins ($3! = 6$). Since peers N1, N2, and N3 are physically close (see Figure 1.10 (a)), such peers produce the same landmark ordering, i.e., L3<L1< L2. As a result, N1, N2, and N3 are placed in the same bin of the overlay network, and they take distinct neighbor zones (see Figure 1.10 (b)). The same approach applies to other peers. Notice that such approach is not perfect. For instance, peer N10 is closer to N3

than N5 in the physical network whereas the opposite situation is observed in the overlay network. Despite its limited accuracy, this technique achieves fast results and copes well with dynamicity. In addition, binning has the advantage of being simple to implement and scalable since peers independently discover their bins without communicating with other participants. Furthermore, it does not incur high load on the landmark machines: they need only echo ping messages and do not actively initiate measurements nor manage measurement information. To achieve more scalability, multiple close-by nodes can act as a single logical landmark.

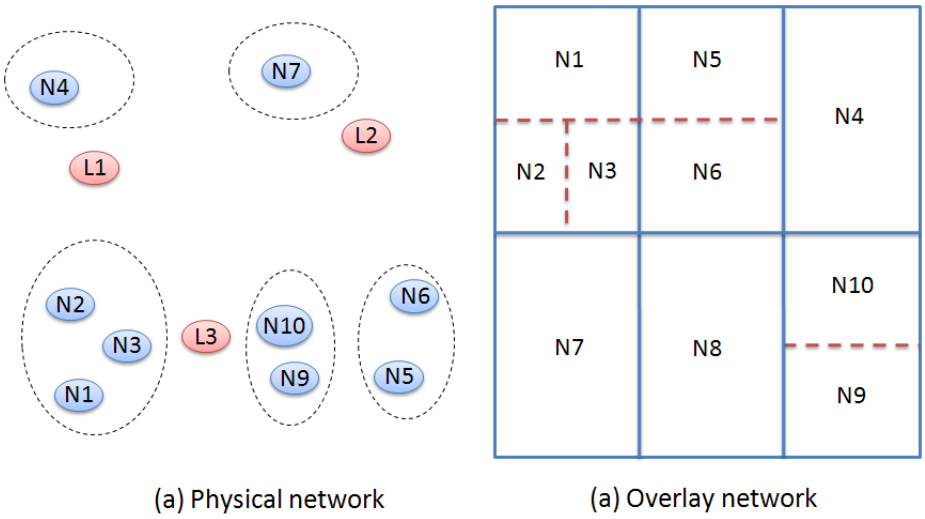


Figure 1.10: Locality-aware construction of CAN

An observation about the aforementioned locality-aware schemes is that the technique used in Pastry and Tapestry is very protocol-dependent and thereby cannot be extended to other contexts in a straigthforward manner, whereas the binning technique can be more generally applied in contexts other than in structured overlay, like unstructured overlays.

1.4.0.2 Trend 2: Interest-Based Topology Matching

In attempt to improve the performance of P2P systems and the efficiency of search mechanisms, some works have addressed the arbitrary neighborhood of peers from a semantic perspective. Recent measurement studies [HKFM04, FHKM04, SMZ03] of P2P workloads have demonstrated the inherent presence of *semantic proximity* between peers, i.e., similar interests between peers. They have shown that exploiting the implicit interest-based relationships between peers may lead to improvements in the search process. In short, they have reached the following conclusion: "if a peer has an object that I am

interested in, it is very likely that he will have other objects that I am (or will be) interested in".

These interest-based relationships can be translated into logical connections between peers that can either replace or be added on top of a peer neighborhood. If we consider Figure 1.4, peer A has peer B as a neighbor specified by its overlay topology and could have extra connections with semantically similar peers like peer D. Then, these semantic connections can be used to achieve efficient search.

In the following, we discuss two representative works along these lines. They were initially proposed for unstructured overlays. When applying one of them, a peer maintains two types of neighbors: its neighbors in the unsructured overlay (e.g., random peers) and its interest-based neighbors. Upon receiving a query, the peer uses its interest-based neighbors first; if this first phase fails the normal search phase is performed via its normal neighbors. In superpeer overlays, the first phase can be used to bypass the superpeers thus alleviating their load.

Semantic Clustering. Garcia-Molina et al. [CGM04] introduces the concept of semantic overlays and advocates their potential performance improvement. Peers with semantically similar content are grouped into clusters together. Clusters can overlap because a peer can simultaneously belong to several clusters related to its content. To achieve this, the authors assume global knowledge of the semantic grouping of the shared documents and accordingly choose a predefined classification hierarchy. Then, each peer decides which clusters to join by classifying its documents against this hierarchy. To join its clusters, the peer finds peers belonging to these clusters by flooding the network. However, It is not clear how this solution performs in the presence of dynamic user preferences.

Interest-Based Shortcuts. In [SMZ03], the concept of shortcut is proposed, allowing peers to add direct connections to peers of similar interests besides their neighbors. The similarity of interests are captured implicitly based on recent downloads and accordingly, interest-based shortcuts are dynamically created in the network: basically, a peer adds shortcuts to peers among those from which it had recently downloaded content. In practice, these shortcuts are discovered progressively while searching for content via flooding. Furthermore, the time for building interest-based groups is non-trivial, and these groups may be no more useful when the peer goes offline and then online again, due to the dynamic nature of P2P networks.

The aforementioned schemes may also be applied to structured overlays. In addition to its routing table, a peer may maintain interest-based neighbors and use them conjunctly. However, this increases the routing state at each peer and incurs extra storage and update overhead.

1.4.0.3 Trend 3: Gossip Protocols as Tools

We now present the usage of gossip protocols in P2P systems. They can serve as efficient tools to achieve new P2P trends in a scalable and robust manner. We particularly focus on gossiping because we greatly rely on these protocols to achieve our thesis purposes.

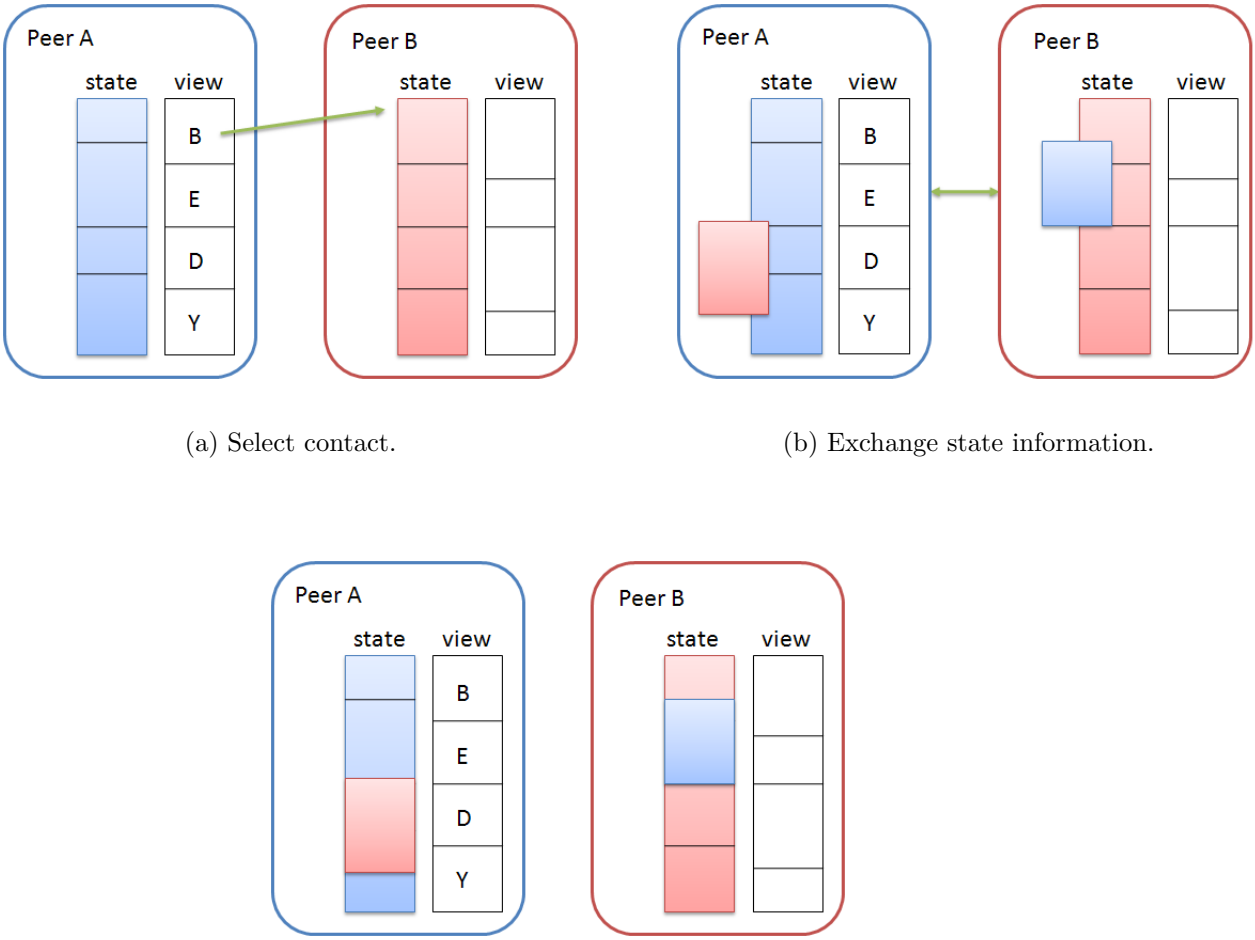
Gossip has recently received considerable attention from researchers in the field of P2P systems [KvS07]. In addition to their inherent scalability, they are simple to implement, robust and resilient to failures. They are designed to deal with continuous changes in the system, while they exhibit reliability despite peer failures and message loss. This makes them ideally suited for large-scale and dynamic environments like P2P systems. In this section, we provide generic definition and description of gossip protocols, then we investigate how P2P systems can leverage these protocols.

Generic Definition Gossip algorithms mimic rumor mongering in real life. Just as people pass on a rumor by gossiping to their contacts, each peer in a distributed system relays new information it has received to selected peers which in their turn, forward the information to other peers, and so on. They are also known as *epidemic protocols* in reference to virus spreading [DGH+87].

Generic Algorithm Description The generic gossip behavior of each peer can be modeled by means of two separate threads: an *active thread* which takes the initiative to communication, and a *passive thread* which reacts to incoming initiatives [KvS07]. Peers communicate to exchange information that depends strictly on the application. The information exchange can be performed via two strategies : *push* and *pull*. A push occurs in the active thread, i.e., the peer that initiates gossiping shares its information upon contacting the remote peer. A pull occurs in the passive thread, i.e., the peer shares its information upon being contacted by the initiating peer. A gossip protocol can either adopt one of these strategies or the combination of both (i.e., *push-pull* which implies a mutual exchange of information during each gossip communication).

Figure 1.11 illustrates in more detail a generic gossip exchange. Each peer A knows a group of other peers or *contacts* and stores pointers to them in its *view*. Also, A locally maintains information denoted as its *state*. Periodically, A selects a contact B from its view to initiate a gossip communication. In a pull-push scheme, A selects some of its information and sends them to B which, in its turn, does the same. Upon receiving the remote information, each one of A and B merges it with its local information and update their state. At that point, how a peer deals with the received information and accordingly update its local state is highly application dependent.

How P2P Systems Leverage Gossip Protocols Gossip stands as a tool to achieve 4 main purposes [KvS07]: *dissemination*, *resource monitoring*, *topology construction* and *peer sampling*. Figure 1.12 illustrates these gossip-based services and how they interfere in a P2P system that is represented by an overlay layer and a search layer.



(a) Select contact.

(b) Exchange state information.

(c) Merge and update local state.

Figure 1.11: Peer A gossiping to Peer B.

Introduced by Demers et al. [DGH+87], **dissemination** has traditionally been the purpose of gossiping. In short, the aim [EGKM04] is to spread some new information throughout the newtork by letting peers forward messages to each other. The information gets propagated exponentially through the network. In general, it takes $O(logN)$ rounds to reach all peers, where N is the number of peers. Figure 1.12 shows that gossip-based dissemination can be used to feed the search layer with indexing information useful to route queries. Basically, a peer can maintain and gossip information about the content stored by other peers and decide accordingly to which peers it should send a query.

Then, gossiping has turned out to be a vehicule of **resource monitoring** in highly dynamic environments. It can be used to detect peer failures [RMH98], where each peer is in charge of monitoring its contacts, thus ensuring a fair balance of the monitoring

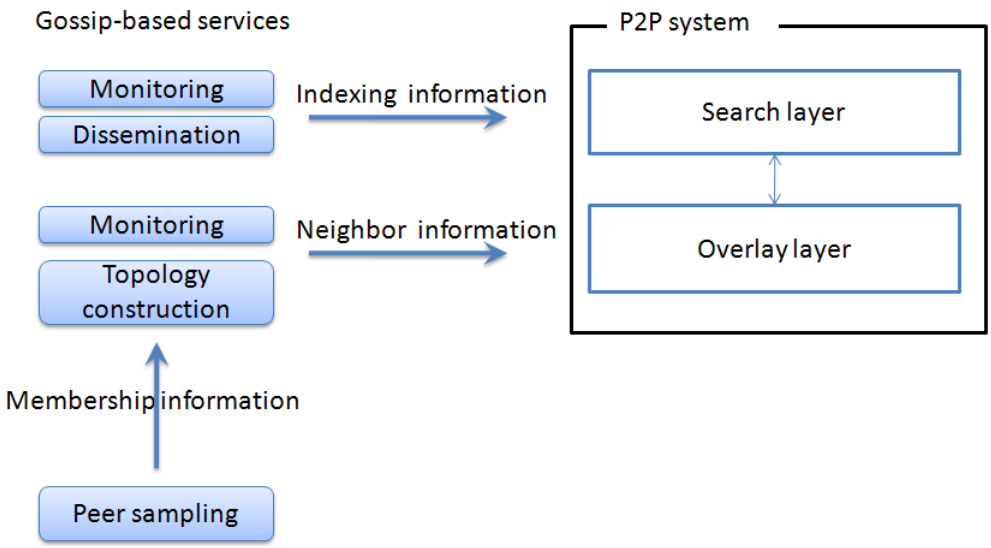


Figure 1.12: How a P2P system can leverage gossiping.

cost. Further, gossip-based monitoring can guarantee that no node is left unattended, resulting in a robust self-monitoring system. In Figure 1.12, the monitoring service is used to maintain the overlay under churn by monitoring a peer's neighbors. In addition, it interferes in the search layer to monitor indexing information in face of content updates and peer failures.

Recently, various researches have explored gossiping as a mean for **overlay construction and maintenance** according to certain desirable topologies (e.g., interest-based, locality-based, random graphs), without requiring any global information or centralized administration. In such systems, peers self-organize under the target topology, via a selection function that determines which neighbors are optimal for each peer (e.g., semantic or physical proximity). Along these lines, several protocols have been proposed such as Vicinity [VvS05] which creates a semantic overlay and T-Man [JB05] that provides a general framework for creating topologies according to some ranking function. Figure 1.12 represents the topology construction service providing peers with specific neighbors and thereby connecting the P2P overlay.

Analyses [JGKvS04] of gossip protocols reveal a high reliability and efficiency, under the assumption that the peers to send gossip messages to are selected uniformly at random from the set of all participant peers. This requires that a peer knows every other peer, i.e., that the peer has *global knowledge of the membership*, which is not feasible in a dynamic and large-scale P2P environment. **Peer sampling** offers a scalable and efficient alternative that continuously supplies a node with new and random samples of peers. This is achieved by gossiping membership information itself which is represented by the set of contacts in a peer's view. Basically, peers exchange their view information, thus

discovering new contacts and accordingly updating their views. In order to preferentially select peers as neighbors, gossip-based overlay construction may be layered on top of a peer sampling service that returns uniformly and randomly selected peers. Well-known protocols of peer sampling are Lpbcast, Newscast and Cyclon [VGS05]. In Figure 1.12, we can see the peer sampling service supporting other gossip-based services and supplying them with samples of peers from the network.

To conclude this section on gossip protocols, we shed light on their salient strenghts and weaknesses.

Strengths Gossip algorithms have the advantage of being extremely simple to implement and configure [Bir07]. Furthermore, they perfectly meet the decentralization requirement of P2P systems since many of them are designed in a way to let peers take local-only decisions. If properly designed, they can balance and limit the loads over participant peers.

Gossiping also provides high robustness which stems from the repeated probabilistic exchange of information between two peers [KvS07]. Probabilistic choice refers to the choice of peer pairs that communicate while repetition refers to the endless process of choosing two peers to exchange information. Therefore, gossip protocols are resilient to failures and frequent changes and they cope well with the dynamic changes in P2P systems.

Weaknesses The usage of gossip might introduce serious limitations [Bir07]. The protocol running times can be slow and potentially costly in terms of messages exchanged. One should carefully tune gossip parameters (e.g., periodicity) in a way that matches the goals of the target appplication.

1.4.0.4 Trend 4: P2P Overlay Combination

We now present a recent trend that is changing the classical categorization of P2P systems. Lately, several approaches have been proposed to build a P2P system over multiple overlays in order to combine their functionalities and leverage their advantages. The combination might involve structured and unstrucured overlays as well as interest- (or semantic) and locality-based overlays. The construction and maintenance of the combined overlays might imply additional overhead which should not compromise the desirable gains. Below, we present and discuss some examplary approaches.

Structured & Unstructured. The approach presented in [CCR04] improves the unstructured Gnutella network by adding some structural components. The motivation behind is that unstructured routing mechanisms can support complex queries but generate significant message overhead. Structella [CCR04] replaces the random graph of Gnutella with the structured overlay of Pastry, while retaining the flexible content placement of unstructured P2P systems. Queries in Structella are propagated using either flooding or

random walks. A peer maintains and uses its structured routing table to flood a query to its neighbors, thus ensuring that peers are visited only once during a query and avoiding duplicate messages. However, this work does not enable the important features of locality and interest awareness.

Interest & Locality-based. The work in [CW04] builds Foreseer, a P2P system that combines an interest-aware overlay and a locality-aware overlay. Thus, each peer has two bounded sets of neighbors: proximity-based (called *neighbors*) and interest-based (called *friends*). Finding neighbors relies on a very basic algorithm that improves locality-awareness slowly with time. Whenever a node discovers new peers, it replaces its neighbors with the ones that are closer in latency. A similar scheme is used to progressively make and refine friends from the peers that satisfy queries of the node in question. Friends are preferentially selected by comparing their content similarity with the target node. However these schemes operate on long-time scale.

Joint Overlay. In [MBK07], the authors leverage the idea of cohabiting several P2P overlays on a same network, so that the best overlay could be chosen depending on the application. The distinctive feature of this proposal is that, in the joint overlay, the cohabiting overlays share information to reduce their maintenance cost while keeping the same level of performance. As an example, they describe the creation of a joint overlay with a structured overlay and an interest-based unstructured overlay using gossip protocols. Thus each peer belongs to both overlays and can alternatively use them.

DHT Layering or Hierarchy. The work in [NT04] organises the structured overlay in multiple layers in order to improve performance under high levels of churn. They introduce the concept of heterogeneity with respect to peer behavior, being altruistic or selfish. The idea is to concentrate most routing chores at altruistic peers; these peers are willing to carry extra load and have the required capabilities to do so. The authors also assume that altruistic peers stay connected more than others. Thus, a main structured overlay is built over altruistic peers, and each one in its turn is connected to a smaller structured overlay of less altruistic peers. Figure 1.13 shows an example of a two-layers DHT, where the main DHT represents the altruistic network and links several DHT-structured clusters. The P2P overlay can be further clustered, resulting into multiple layers.

A similar work is proposed in [SX08] and addesses the problem of load balancing in a heterogenous environment in terms of capacities. Likewise, a main structured overlay is built over high-capacity peers, and each one acts as a super-peer for a locality-based cluster of regular peers. Each peer has an ID obtained by hashing its locality information (using the binning technique of Section 1.4.0.1). A regular peer is assigned to a super-peer whose ID is closest to the peer's ID, which results in regular peers being connected to their physically closest super-peer.

These proposals are orthogonal to our work, as they mainly focus on DHTs performance under heterogeneity. It is not clear how they can support content distribution and location.

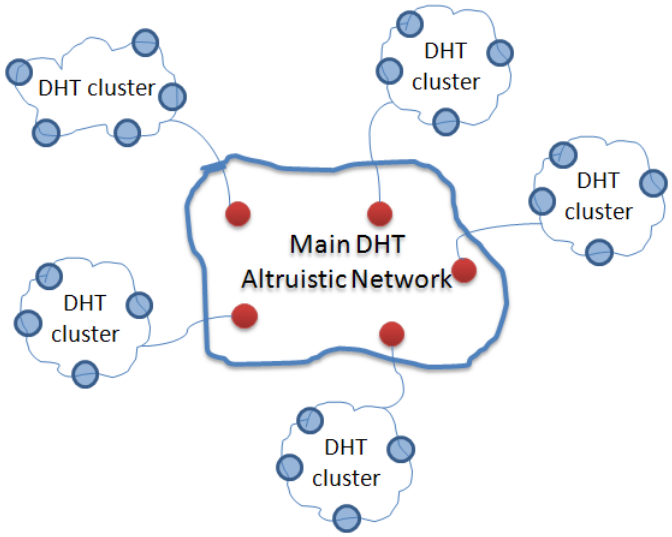


Figure 1.13: A two-layers DHT overlay [NT04].

1.4.0.5 Challenges to Be Met

When refining the P2P network via sophisticated techniques (like locality or interest aware schemes), one should make sure that the overhead is worth the performance improvement. Based on the aforementioned trends, we have identified two major **challenges** that need to be explored:

Challenge 1 *To capture or gather the information (e.g., topological or semantic relationships) in a manner that is both practical and scalable. This should be done without:*

- *requiring global knowledge or centralized administration.*
- *incurring large overheads of messages and/or data transfers on the existing overlay.*

Challenge 2 *To be adaptive to dynamic changes and churn. Indeed, the solution should provide a scheme that can still be valid and effective when new peers join or/and existing ones leave. For this, it should:*

- *operate on short-time scales. Given that participants join and leave on short time-scales, a solution that operates on long-time scales would be continulally reacting to fluctuating peer membership without stabilizing.*
- *avoid grouping peers into a static configuration which does not evolve well as the behavior or characteristics of peers change.*

1.4.0.6 Discussion

In this section, we have reviewed the recent trends in the P2P literature, mainly from the perspective of content sharing. We have seen that they improve the performance of P2P infrastructures but incur additional challenges related to scalability and dynamicity on their design. Table 1.2 summarizes the main approaches that integrate recent P2P trends and evaluates them with respect to the challenges.

Trend	Challenge 1	Challenge 2
Locality-aware schemes		
Physical clustering	no	no
LTM technique	yes	no
Pastry & Tapestry locality-aware scheme	yes	no
Binning technique	yes	yes
Foreseer	yes	no
Interest-aware schemes		
Semantic clustering	no	no
Interest-based shortcuts	yes	no
Foreseer	yes	no
Using Gossip	partially	yes
Joint Overlay	no	no

Table 1.2: Trends vs. challenges.

In short, matching the overlay with a locality or interest-aware scheme could bring great benefits to the P2P system in terms of efficiency and quality of service. However, the schemes should be kept simple and practical. Among the proposed approach, the binning technique is the perfect match to achieve locality-awareness with respect to the challenges. It relies on topological information that is practical and incurs limited overhead (Challenge 1); it also operates fast and can easily adapt to changes (Challenge 2).

Gossiping can be used to build locality and interest-based schemes and can answer the challenges. It can be designed in a way that provides simplicity, decentralization and high robustness. In general, gossip is a tool, not an end in itself. It should be used selectively, in contexts where gossip is the best choice, mainly in the fields of monitoring and dissemination and overlay maintenance. This implies that gossip protocols need to be combined with other tools to build an efficient P2P infrastructure [Bir07]. Further, efficient tuning is needed so that gossip does not incur significant delays and overheads in terms of messages. The message overhead might prevent gossip protocols from fully satisfying Challenge 1.

Finally, we have concluded that structured and unstructured overlays should not be seen as competing but rather complementing each other. Each category provides specific and unique functionalities. Combining different overlays and schemes might reveal interesting, yet very challenging. In particular, the maintenance of several overlays should

not overwhelm the P2P system. An interesting solution is to leverage the combination in the maintenance mechanisms (e.g., exploiting one overlay to maintain the other). Also, gossip can be a potentially effective solution for this issue that requires no centralization if properly designed.

1.5 P2P Content Distribution Systems

In the previous section, we provided a generic presentation of P2P systems which can serve as infrastructures for applications like content distribution. In this section, we deepen our study on P2P content distribution systems. In particular, we examine the existing proposals and identify the shortcomings according to the requirements and challenges identified through this chapter.

Most of the current P2P applications fall within the category of content distribution, which range from simple file sharing, to more sophisticated systems that create a distributed infrastructure for organizing, indexing, searching and retrieving content [ATS04]. P2P content distribution functionalities are achieved via collaboration among peers, scalability being ensured by resource sharing. By distributing tasks across all participating peers, they can collectively carry out large-scale content distribution without the need for powerful and dedicated servers.

In the following, we first give an overview (Section 1.5.1) where we define the context of P2P content distribution and recall the P2P and CDN requirements discussed in the previous sections. Then, we discuss the existing works and enlighten the open issues of P2P file sharing (Section 1.5.2) and P2P CDN (Section 1.5.3).

1.5.1 Overview

Recall that the design of a CDN brings stringent requirements which are *performance*, *reliability* and *scalability* (cf. Section 1.2.3). In contrast to traditional CDNs, a P2P infrastructure relies on peers which are not dedicated servers but autonomous and volunteer participants with their own heterogenous interests. When building a CDN over a P2P infrastructure, it is vital to reconcile and coordinate these requirements with the ones introduced by P2P systems, i.e., *autonomy, expressiveness, efficiency, quality of service, robustness,* and *security* (cf. Section 1.3.4). Let us recapitulate and identify the different correlations between CDN and P2P requirements. Further, we point out where the P2P recent challenges interfere.

Performance, Quality of Service. Performance meets the requirement of quality of service. It is ensured via locality-aware and efficient location of content as laid out previously. While many P2P systems abstract any topological information about the underlying network, locality-awareness should be a top priority in order to achieve short query response times. The locality-aware solution should overcome Challenges 1 and 2,

i.e., it should be kept simple, incur acceptable overhead, operate fast and adapt to churn and high scales.

Scalability, Efficiency. In order to make efficient use of P2P inherent scalability, it is essential to distribute load equitably over peers. This is realized if all peers fairly share the processing of queries as well as the routing load. However, when some peers hold popular content, they may present hot spots, attracting large amounts of queries.

Reliability/Robustness, Autonomy. Reliability can only be ensured by the robustness of the P2P system under the dynamic nature of its peers. There is a strong correlation between robustness and autonomy [DGMY02]. Indeed, churn and failure rates are much higher that in CDN infrastructures because of the autonomous nature of peers. Routing and serving querie can be difficult to achieve as peers join and leave frequently and unexpectedly. Furthermore, the solutions of caching and replication that improve content availability are highly constrained by the *autonomy* of peers.

Efficiency, Autonomy. Decoupling efficiency from autonomy seems to be very challenging, given that most existing techniques tend to sacrifice autonomy to achieve efficiency [DGMY02]. This is because less autonomy allows more control on the content placement and topology such that there exist a deterministic way to locate content within bounded cost. In addition, search seems to be more efficient if the content is replicated. An interest-based scheme might be useful to leverage the interests of peers in the search and replication mechanisms. To be efficient, the scheme must meet Challenges 1 and 2 by being dynamic, practical and scalable.

Before we deepen our analysis of P2P content distribution, let us stand back and get an overview of the context. Figure 1.14 illustrates the relation between the P2P infrastructure and content distribution as its overlying application. The P2P infrastructure provides specific services which are identified by [ATS04] as follows: routing and location, anonymity and reputation management. We focus on P2P infrastructures for routing and location. The operation of any P2P content distribution system relies on a network of peers within which messages must be routed with fault-tolerance and minimum overhead, and through which peers and content can be efficiently located. We have previously seen different infrastructures and algorithms that have been developed to provide such services. As laid out in the previous sections, the infrastructure characteristics, i.e., the topology, the routing protocol, the degree of centralization and structure, play a crucial role in the performance, reliability and scalability of the P2P content distribution system. Figure 1.14 shows that the application layer contains functionalities that are specifically tailored to build content distribution. Among these functionalities, we mention indexing, replication and caching which will be discussed along the next sections.

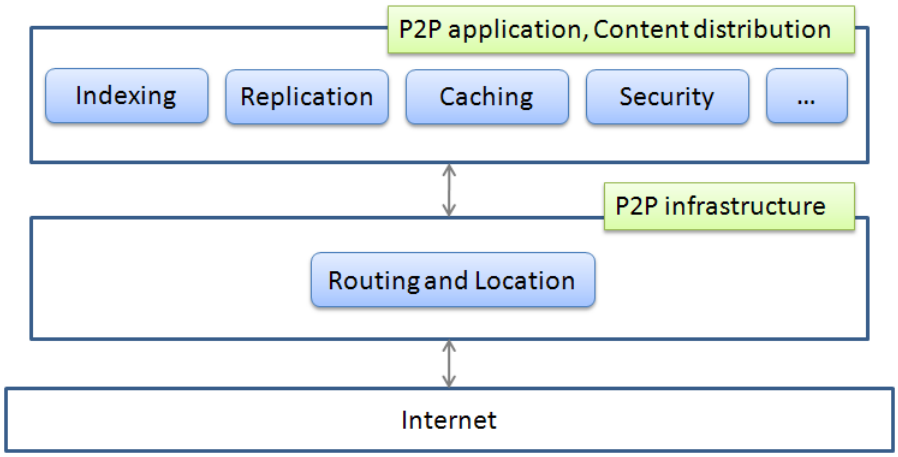


Figure 1.14: P2P infrastructure for content distribution.

1.5.2 P2P File Sharing

File sharing remains widespread in P2P networks. Some of the most popular networks are BitTorrent, FastTrack/Kazaa, Gnutella, and eDonkey. They are generally deployed over unstructured overlays, mainly due to their flexibility and support for keyword search.

File-sharing applications can afford to have looser garantees on the CDN requirements because such applications are meant for a wide range of users from non-cooperating environments [YGM02]. These are typically light-weight applications that adopt a best-effort approach to distribute content and yet are accepted by the user population [ATS04]. Nonetheless, these systems should rigorously aim at keeping the network load at bay to enable a deployment over large-scales.

Since unstructured networks commonly use blind techniques to locate files as discussed in Section 1.3.2.2, many efforts have been made to avoid the large volume of unnecessary traffic incurred by such techniques. As such, *informed techniques* have been proposed, which rely on additional information about object locations to route queries. Typically, a peer can maintain an index of the content provided by other peers and decide accordingly to which peers it should send the query.

Next, we provide more insight into P2P file sharing systems by identifying their inherent properties. Then, we discuss the indexing techniques proposed in this context.

1.5.2.1 Inherent Properties

P2P file sharing exhibit inherent properties that should be well understood in order to design efficient solutions. In a nutshell, they are characterized by a high level of *temporal locality* in queries, involve a *natural replication* of files, and commonly witness keyword queries. Furthermore, P2P file sharing systems are considered as the leading consumer

of Internet bandwidth [SGD+02]. The challenges are thereby to leverage intrinsic aspects (natural replication, temporal locality) and address inherent issues (keyword lookup, bandwidth consumption).

Natural Replication. File sharing systems vehiculate a "natural" replication of files, which is enabled by the flexibility of unstructured overlays with respect to content placement. When a peer requests a file, it downloads a copy which is often made available for upload to other peers. Thus, the more popular a file, the more it is "naturally" replicated and spread into the P2P network [CRB+03, GDS+03].

Temporal Locality. Several analyses [Mar02, LBBsS02, Sri01] of P2P file sharing systems observed that the queries exhibit significant amounts of *temporal locality*, that is, queries tend to be frequently and repeatedly submitted, requesting few popular files. Accordingly, they advocated the potential of caching to capitalize on this temporal locality. Caching is often done for the purposes of improved performance (i.e., higher hit ratio, reduced latencies and bandwidth costs). It can also be viewed as a cost-effective version of replication since it takes advantage of the unused storage resources and can evict copies at any time.

Keyword Lookup. File-sharing systems like Gnutella [Gnu05] vehiculate a simple keyword match. Users often generate queries that contain a set of keywords and peers generate query reponses referring to files whose names contain all the query keywords. Thus, query routing are required to support keyword lookup.

Bandwidth Consumption. Many measurement studies on P2P file sharing (e.g., [KRP05]) shed light on the tremendous bandwidth consumption and its detrimental impact on both users and Internet Service Providers (ISPs). For the end users, their participation into a P2P network swamps all the available bandwidth and renders the link ineffective for any other use. For the ISPs, the P2P traffic is a major source of costs since an ISP handles the file tranfer at the physical network layer. This increase of costs on ISPs is passed on to the user in the form of higher prices to access the Internet. The main reason behind this pertinent problem involves file transfers. P2P files are three orders of magnitude larger than web objects, since the majority of shared files are audio and movies [LBBsS02, SGD+02]. Also, they are randomly transferred between peers without any consideration of network distances.

On the one hand, the studies suggest to cache files in order to remedy this problem. However, this cannot be achieved without relying on a dedicated caching infrastructure. Indeed, in file sharing communities, users rarely accept to store or cache large files on behalf of each others.

On the other hand, the studies of [GDS+03, KRP05] present evidence on the potential bandwidth savings of locality-aware file transfers. Indeed, the analysis in [GDS+03] has shown that there is an untapped locality in file-sharing workload, i.e., a requested file

is likely to be available at peers close to the requester in network locality. This means that there is substancial opportunity to improve file sharing performance by exploiting the untapped locality. In short, a query can be intentionally redirected towards nearby files, to optimize the file transfer.

BitTorrent [PGES05] addresses this issue under a different angle. Basically, a peer downloads multiple fragments in parallel from multiple downloaders of the target file, thus distributing the load among several peers. There are two primary concerns about this approach. First, it ignores locality-awareness by randomly choosing downloaders, which can further accentuate the bandwidth problem. The second concern is the centralized aspects of the search operation which limits scalability and robustnees. To share a file, a peer first creates a metadata file called a *torrent* that contains information about the *tracker*. A tracker is a centralized server that keeps track of all current downloaders of a file and coordinates the file distribution. Peers that want to download the file must first obtain a torrent file for it, and connect to the specified tracker, which tells them from which other peers to download the fragments of the file. BitTorrent provides no way to index torrent files which are thus hosted by specific websites. On-going improvements aim at distributing the tracker's functionality (i.e., the discovery of file downloaders) over the peers via DHT or gossip protocols. BitTorrent can also serve as a P2P CDN to distribute web content and relieve original web servers.

1.5.2.2 Indexing Approaches

In unstructured networks, informed search is achieved by the use of distributed indexes to route queries. Basically, a peer maintains indexes related to the content stored by remote peers. Then, the peer evaluates any received query against its indexes and redirects it to peers that can contribute to it.

In general, an index yields a trade-off between compactness and accuracy, and between maintenance overhead and broadness. The more the index compactly represents the content, more storage efficiency is achieved but more false positives can result from index lookup. At the same time, the more broad is the coverage of the index (i.e, indexing distant content), the more effort and overhead is generated to maintain the indexes in dynamic environments.

An indexing approach should overcome several challenges, so that it does not partially offset the benefits of indexing itself. Below, we identify three main challenges:

- limit the overhead involved in the creation and update of indexes.
- support keyword search.
- introduce locality-awareness.

There are two types of indexes, a *forwarding index* and a *location index*. A *forwarding index* allows to reach the requested object within a varying number of hops, while a *location index* allows to reach the target in a single hop.

The approaches of **forwarding indexes** suply direction information towards the content, rather than its actual location. Two representative approaches are *routing indices* and *local indices*.

Routing Indices. This technique [CGM02] assumes that all documents fall into a number of topics, and that queries request documents on particular topics. Also, each peer stores, for every topic, the approximate number of documents that can be retrieved through each one of its neighbors (i.e., including all the peers accessible from or linked to this neighbor). Figure1.15 illustrates the use of routing indices (RI) over four topics of interest. Considering the RI maintained by peer A, the first row contains the summary of its local index, showing that A has 300 documents (30 about databases, 80 about networks, none about theory, and 10 about languages). The rest of the rows represent compound RI. For example, they show that peer A can access 100 database documents through D (60 in D, 25 in I, and 15 in J).

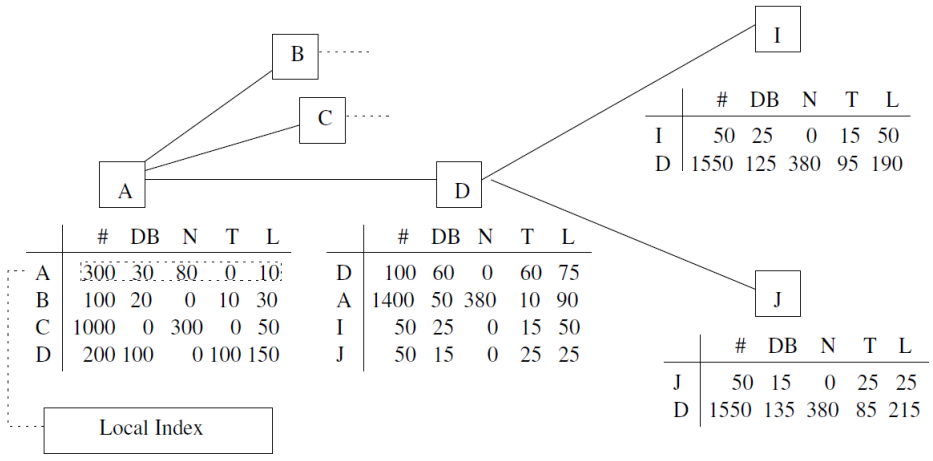


Figure 1.15: Example of routing indices [CGM02].

Local Indices. This approach is proposed in [YGM02]. Each peer maintains an index over the content of all peers within r hops of itself, and can therefore process any received query on behalf of these peers. While a query is routed using BFS (breadth-first-search or flooding), it is processed only at the peers that are at predefined hop distances from the query originator. According to the authors' analysis, the hop distance between two consecutive peers that process the query must be $2 * r + 1$. This allows querying all content without any overlap and reducing the query processing time.

In the aforementioned appoaches, the indexes maintained by each peer would be extraordinarily large, and hence the overhead related to their creation and update may

become prohibitively expensive, thus compromising the benefits. Furthermore, they do not consider locality-awareness for the purpose of reducing bandwidth consumption.

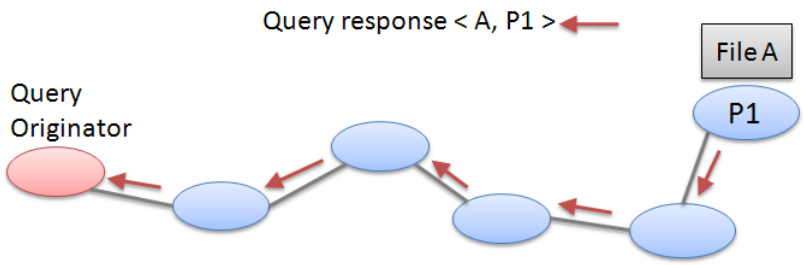
Another category of approaches uses **location indexes** which aim at determining which peers can provide certain content. Examples of such approaches are *index caching*, *intelligent BFS* and *Bloom Filter-based indices*.

Index Caching. The basic concept is to cache query responses in the form of indexes, on their way back to the originator. Recall that a query response contains the file identifier (e.g., filename) and the address of the provider peer that has a copy of the file. The advantage of index caching is that it does not incur additional overhead to create and update the indexes, as they exploit passing-by query responses.

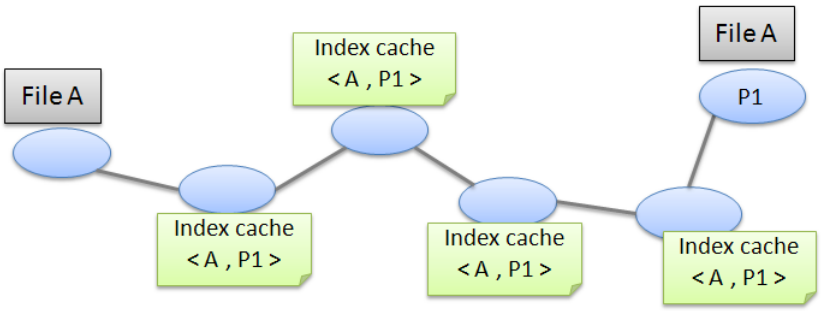
Let us briefly review the different approaches of index caching. Centralized caching [PH03] at the gateway of an organization does not leverage node resources and is likely to produce bottlenecks. *Distributed index caching* is illustrated in Figure 1.16, where a query requesting file A has reached peer P1 that can provide a copy of A (see Figure 1.16a). As normally done in unstructured systems, a query response that contains the filename of A and the address of P1 is sent back to the query originator. Forwarding peers cache the query response as an index for file A and thus can respond to eventual queries requesting file A. *Uniform index caching* [Sri01] consists that each peer caches all passing-by query responses, which results in large amount of duplicated and redundant cached among neighboring nodes (see Figure1.16b). *Selective index caching* addresses the problem of redundancy by selectively caching file indexes and accordingly routing queries. However, none of the existing solutions addresses locality-awareness in file transfers. Next, we describe a typical example of selective index caching, i.e., *DiCAS* [WXLZ06].

DiCAS. Peers are randomly assigned to M groups, with each group being identified by an ID noted Gi. Group IDs are used to restrict index caching in some peers along the query reverse path, in order to avoid redundant indexes among neighbors. Hence, a query response is only cached in peers whose Gi matches the filename in the query response, i.e., G$i = \text{hash}(f) \bmod M$ (see Figure 1.17a). Furthermore, Group IDs help searching for file indexes by routing a query towards peers that are likely to have indexes satisfying the query. To forward a query, a peer selects the neighbors whose Gi matches the string of keywords in the query (see Figure 1.17b). When no such neighbors are found, the query is sent to a highly connected neighbor.

This search is specifically tailored for exact-match queries. Therefore, DiCAS is not adapted for keyword searches which are the most common in the context of P2P file sharing. To illustrate this problem, consider a user looking for a file with name $f = key_1 + key_2 + ... + key_n$. Commonly, the user will employ a query with string of keywords $q = key_1 + key_m + key_n$. Based on DiCAS predefined hashing, the file index is cached in peers with G$i = \text{hash}(f) \bmod M$, while the query is forwarded to peers with G$i' = \text{hash}(q) \bmod M$. Obviously, this approach may mislead the query by redirecting it to



(a) Query response on its way back to the originator.



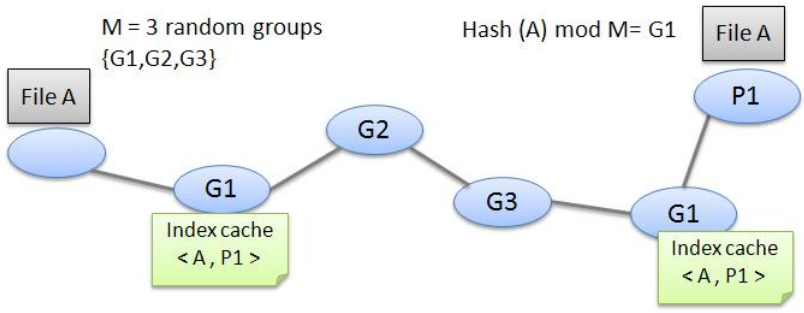
(b) Query response cached by all forwarding peers.

Figure 1.16: Uniform index caching.

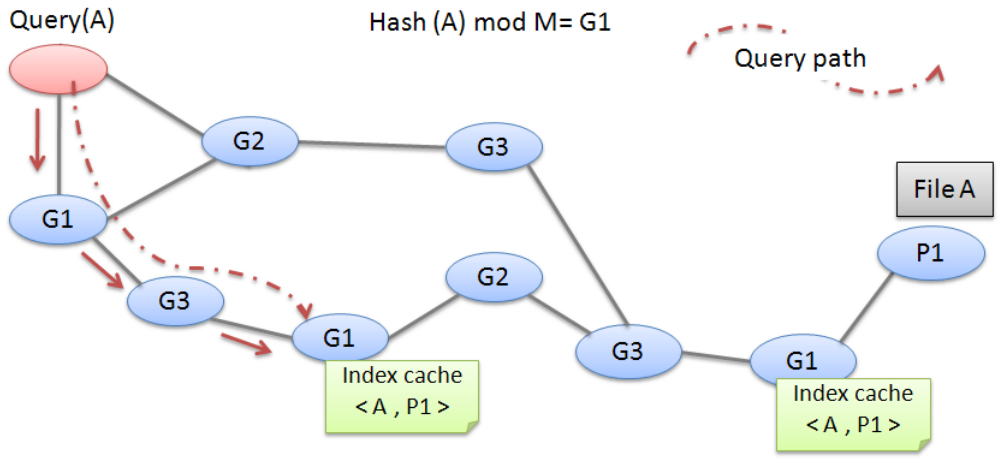
peers that have no indexes for the target file. This results in more flooding overhead and higher response time.

One alternative is to cache file indexes based on the hashing of the query string of keywords, i.e, q. Since multiple combinations of keywords can map to the same filename, it brings back the problem of wide duplication and reduces the efficiency of indexes.

Intelligent BFS. This technique [KGZY02] adapts the basic BFS algorithm. A peer maintains for each neighbor the list of recently answered queries from (or through) this neighbor. When a peer receives a query, it identifies all listed queries that are similar to the newly received query based on some similarity metric, and sends the query to the neighbors that have returned most answers for the similar queries. If an answer is found for the query at a peer, a message is sent to the peers over the reverse path in order to update their statistics. However, this technique produces more routing messages because of update messages. In addition, it can not be easily adapted to the peer departures and file deletions, and it ignores locality-aware aspects.



(a) Query response selectively cached by forwarding peers.



(b) Query selectively routed to neighbors.

Figure 1.17: Selective index caching: DiCAS with M = 3 groups G1,G2,G3.

Bloom Filter-Based Indices. Bloom filters [Blo70] have long been used as a lossy summary technique. The works in [CJ06, CW04] use a Bloom filter to represent the collection of keywords that characterize the objects shared by a peer. By first examining the filter, one can see if a queried file might be at the peer before actually searching the local repository of the peer. Thus, a peer selectively forwards a query to the peers that might satisfy the query. The advantage of using Bloom filters [FCAB98] is that they are space efficient, i.e., with a small space, one can index a large number of data. However, it is possible that a Bloom filter gives a *false positive* answer, i.e., the Bloom filter wrongly returns a positive answer in response to a question asking the membership of a data item. An important feature of a Bloom filter-based index is that it minimizes the maintenance overhead, making it suitable for highly dynamic environments.

In [CJ06], each peer replicates d copies of its Bloom filter and distribute them to its neighbors. Then, peers periodically exchange with each other the Bloom filters they have, so as to widely disseminate them in the P2P network. Each Bloom filter is associated a tag TTL that records the time up to which a Bloom filter is valid. When the time expires, the peer that holds the copy should check with the owner of the copy to obtain a new version. The approach in [CW04] was previously introduced in Section 1.4.0.4 where a peer has two types of neighbors in the P2P overlay: interest-based and proximity-based. Thus, each peer stores the Bloom filters of both types of neighbors. The advantage of this approach over the previous indexing schemes is that it attemps to achieve a locality-aware routing.

1.5.2.3 Discussion

In summary, P2P file sharing is a highly popular application that tolerates some performance limitations and prefers unstructured overlays. However, there is a growing concern regarding its network costs since the P2P traffic overwhelms the Web traffic as a leading consumer of Internet bandwidth. Two main reasons are behind this problem.

First, searching for files is inefficient, generating large amounts of redundant messages. Indexing can help improve search efficiency as it provides information useful for query routing. However, it might imply considerable overhead for the creation and update of indexes. Furthermore, it is highly required from file sharing applications to support keyword lookup. Thus, indexing should not hinder this feature.

Second, the large files are transferred over long network distances, thus overloading the underlying network. Locality-awareness seem to be the best solution for this issue, redirecting queries to close-by files.

On the other hand, there are several inherent properties of P2P file sharing that have not been fully exploited and could be leveraged to attenuate the P2P traffic problem. The most important ones are the temporal locality of queries and the natural replication of files. For instance, index caching leverage temporal locality as it keeps query responses for later queries in order to improve search efficiency.

To conclude, Table 1.3 lists indexing approaches and checks whether or not they answer the different challenges. As an example, the first two approaches do not address the overhead related to their index creation and maintenance. On the same matter, index caching like DiCAS implicitly limits the overhead since it dynamically stores and evicts indexes as query responses pass by. Also, the usage of Bloom filters can achieve, at the same time, maintenance and storage efficiency. Another observation is that most of the approaches do not incorporate locality-awareness in their indexing scheme and thus fail in redirecting queries to nearby locations for short data tranfers.

1.5.3 P2P CDN

Several P2P approaches have been proposed to distribute web content over P2P infrastructures in order to relieve the original web servers. As previously discussed, they

Indexing approach	Overhead efficiency	Keyword lookup	Locality-awareness
Routing indices	no	yes	no
Local indices	no	yes	no
DiCAS	yes	no	no
Intelligent BFS	no	yes	no
Bloom filter indices	yes	yes	yes

Table 1.3: File indexing approaches

can greatly optimize their performance if they take into account recent P2P trends while meeting their challenges (cf. Section 1.4). We classify existing approaches into three main categories: hybrid, unstructured and DHT-based. We also distinguish the currently deployed P2P CDNs. First of all, let us give an overview of caching and replication mechanisms. Then we survey the existing P2P CDNs and investigate if they meet the requirements and leverage the recent trends.

1.5.3.1 Insights into Caching and Replication

In the context of content distribution, content replication is commonly used to improve content availability and enhance performance. More particularly, P2P systems can significantly benefit from replication given the high levels of dynamicty and churn. For instance, if one peer is unavailable, its objects can still be retrieved from the other peers that hold replicas. According to [ATS04], content replication in P2P systems can be categorized as follows.

Passive Replication. It refers to the replication of content that occurs naturally in P2P systems as peers request and download content. This technique perfectly complies with the autonomy of peers.

Active (or Proactive) Replication. This technique consists in monitoring traffic and requests, and accordingly creating replicas of content objects to accommodate future demand.

To improve object availability and at the same time avoid hotspots, most DHT-based systems replicate popular objects and maps the replicas to multiple peers. Generally, this can be done via two techniques. The first one [RFH+01] uses several hash functions to map the object to several keys and thereby store copies at several peers. The second technique consists in replicating the object in a number of peers whose IDs match most closely the key (or in other terms, in the logical neighborhood of the peer whose ID is the closest to the key). The latter technique is commonly used in several systems such as [RD01b, DKK+01].

The study in [CS02] evaluates three different strategies for replication in an unstructured system. The *uniform strategy* creates a fixed number of copies when the

object first enters the system. The *proportional strategy* creates a fixed number of copies every time the object is queried. In the *square-root replication strategy*, the ratio of allocations is the square root of the ratio of query rates. To implement these strategies, the object can be replicated either randomly or at peers along the path from the requester peer to the provider peer. However, it is not clear how the strategies can be achieved in a distributed way (e.g., how to monitor query rate under P2P dynamicity). Further, such proactive replication is not feasible in systems that wish to respect peer autonomy; they may not want to store an object at peers that have not requested it.

Along with replication, there is the classical issue of maintaining consistency between replicas in case of content updates. In this thesis, we do not discuss this issue, however good pointers can be found in our previous work [MPDJP08, DMP07].

Caching The key idea is to cache copies of content as it passes through peers in the network and manage them according to cache replacement policies. In Freenet [CMH+02] for instance, when a search request succeeds in locating an object, the object is transferred through the network node-by-node back to the query originator. In the process, copies of the object are cached by all intermediate nodes.

To shorten query routes and thus reduce search latencies, DHT-based approaches like [RD01b,DKK+01] cache additional copies of the objects along the lookup path towards the peers storing these objects.

1.5.3.2 Deployed Systems

To the best of our knowledge, the P2P CDNs that are currently available for public use mainly comprise CoralCDN [FFM04], CoDeeN [PWP+04] and CobWeb [SRS05]. These systems are deployed over PlanetLab which provides a relatively trusted environment consisting of nodes donated largely by the research community. Basically, they rely on a network of cooperative proxy servers that distribute web content and handle related queries. Such systems cannot be categorized as pure P2P solutions because they are using dedicated servers rather than exploiting client resources. The only P2P characteristic exhibited by these systems is the absence of centralized administartion. We examine one typical example of these systems, CoralCDN.

CoralCDN [FFM04]. CoralCDN relies on a hierarchy of tree-based overlays that cluster nearby nodes. Each level of the hierarchy consists of several overlays, and each overlay consists of the set of nodes whose average pair-wise RTTs are below the threshold defined by this level. A node is member of one overlay at each hierarchy level and detains the same node ID in all overlays to which it belongs. Figure 1.18 illustrates a three-level hierarchy with RTT thresholds of ∞, 60 msec, and 20 msec for level 0, 1, and 2 respectively. It focuses on Node R and only shows the three overlays to which R belongs at each level. R is physically the closest to C_2 among the nodes (C_0, C_1, C_2, C_3) because R and C_2 share the highest-level overlay.

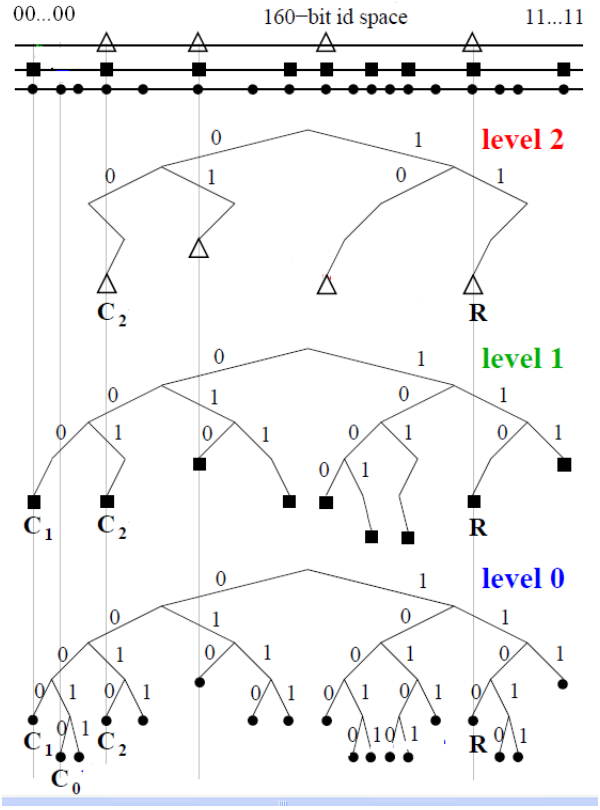


Figure 1.18: CoralCDN hierarchy of key-based overlays [FFM04].

Each overlay is structured according to a tree topology. A key is mapped to several nodes whose IDs are numerically close to the key, in order to avoid hot spots due to popular objects. A node stores pointers related to the object whose key is mapped to its node ID. In Figure 1.18, Node R has the same node ID in all its overlays; we can view a node as projecting its presence to the same logical location in each of its overlays.

Based on this indexing infrastructure, CoralCDN allows to locate web object copies hosted by nearby proxies of CoralCDN: the proxies will be represented by the nodes of the hierarchy. Based on its RTT measurements, a client is redirected via the DNS services to a nearby CoralCDN proxy which eventually provides her the requested object. If not cached locally, the proxy can perform a key-based routing throughout its overlays in order to find a pointer to a remote copy of the object; it starts at the highest-level overlay of the proxy to benefit from network locality then progresses down the hierarchy. Once the object is fetched and locally cached, the proxy inserts pointers to itself, with respect to the object, in the different overlays to which belongs this proxy. To handle dynamicity, pointers are associated with TTL (*fixed time-to-live*) values and are periodically refreshed by their referenced proxy.

1.5.3.3 Centralized Approaches

The first category of approaches [RY05,PS02] relies on the web-server that centralizes and manages the directory information. Basically, the server maintains a directory of peers to which its objects have been transferred in the past and manages the redirection of queries. When a client requests an object, the server returns several peers from the redirection directory. The client first tries to retrieve the object from one of those peers. If this fails, the object is directly served by the server.

To minimize redirection failures in a P2P dynamic environment, OLP [RY05] tries to predict the object lifetime and accordingly selects the peer to which the query should be redirected. However, redirection in OLP does not consider locality-awareness when providing clients with object locations. CoopNet [PS02] tries to incorporate locality-awareness as the web-server sends to the requester client a list of nearby peers providing the requested object. To limit the server redirection, a client connects to the peers provided by the web-server and forms a small network with them. However, there is no well-defined search algorithms within these networks. Moreover, CoopNet does not deal with dynamic aspects because the web-server cannot detect which peers in its directory have failed or discarded their cached objects.

Centralized approaches lack robustness, because whenever the web-server fails, its content is no longer accessible in spite of available peers with cached copies. As with the traditional server/client model, the server is still a single point of failure. Scaling such systems requires replacing the web server with a more powerful one, to be able to redirect the queries of a large audience.

1.5.3.4 Unstructured Approaches

The second category of approaches uses unstructured overlays for their flexibility and inherent robustness. Two representative systems are Proofs and BuddyWeb.

Proofs. Proofs [SRS02] uses an ustructured overlay in which peers continuously exchange neighbors among each other. This provides each peer with a random view of the system for each search operation. Peers keep their requested objects and can then provide them to other participants. To locate one of the object replicas, a query is flooded to a random subset of neighbors with a fixed TTL, i.e., the max number of hops. The continuous randomization of the overlay has the benefit of improving the network fault-tolerance and tends to uniformly distribute the load over peers. However, the blind searches for not not-so popular objects induce heavy traffic overheads and high latencies. Moreover, Proofs does not leverage new trends, and most importantly locality-awareness which is useful to forward queries to close results.

BuddyWeb. BuddyWeb [WNO+02] also uses an unstructured network and blind search to access objects. However, it relies on central servers to provide each newly joining peer with neighbors that share interest similarities with the peer. Therefore, this interest-based scheme does not meet Challenge 1 as it greatly depends on central servers to

gather, manage and provide all the information. These servers can present single points of failures (i.e., SPOF), which makes BuddyWeb vulnerable and hinders its scalability. Similarly to Proofs, BuddyWeb does not take into account locality-awareness.

1.5.3.5 Structured Approaches

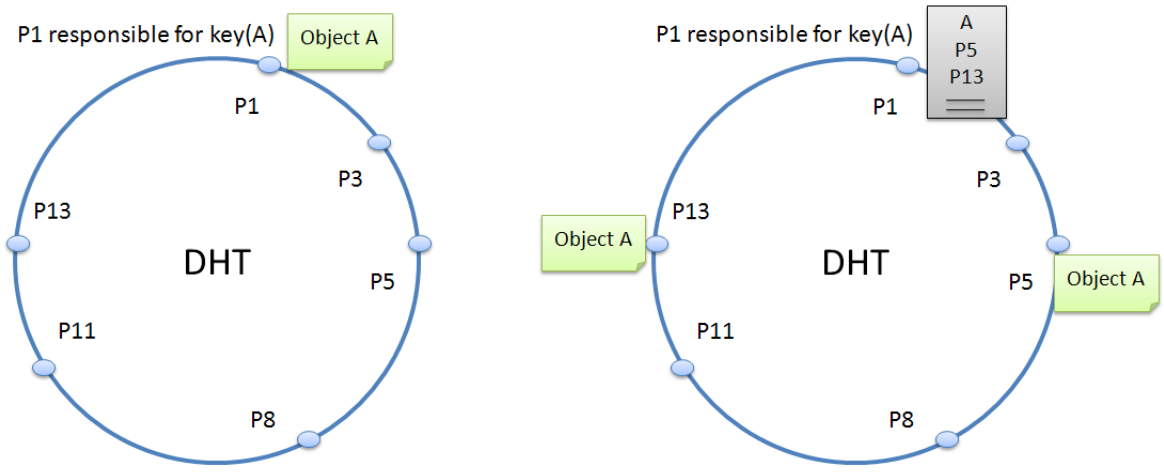
Now, we examine existing approaches that rely on structured overlays in order to benefit from their efficient lookup. We examine Squirrel [IRD02], PoPCache [RFB07] and Backslash [SMB02]. These approaches adopt similar strategies. We have identified two types of strategies, *DHT-Home* and *DHT-Directory*. We also discuss a work that proposes a different approach using a novel DHT, called *Kache*.

DHT-Home Strategy. It places objects at peers with ID numerically closest to the hash of the URL of the object without any locality or interest considerations (see Figure1.19a). Queries find the peer that has the object by navigating through the DHT. To deal with highly popular objects, objects may be progressively replicated along neighbors as the number of requests increases. This is achieved by further forcing peers to store arbitrary content.

DHT-Directory Strategy. The second type of strategy stores at the peer identified by the hash of the object's URL a small directory of pointers to recent downloaders of the object (see Figure1.19b). A query first navigates through the DHT and then receives a pointer to a peer that potentially has the object. Approaches adopting this strategy may be vulnerable to high churn because the directory information is abruptly lost at the failure of its storing peer.

In general, such systems are self-scalable because of the DHT load balancing mechanism and the replication in case of hot spots. However, there are two main drawbacks in the query routing with repsect to the stringent requirement of CDNs on short latencies. First, each query has to navigate through the whole DHT, which implies several routing hops. This can be acceptable in corporate LAN type environments, such that the latency of the network links are a magnitude smaller than the latency of the server. Otherwise, the server will be much faster. Second, unless using a locality-aware overlay combined with proactive replication, the query is served from a random physical location. To conclude, the aforementioned approaches do not exploit recent P2P trends for performance improvement.

Kache. Kache [LGB03] relies on a new form of DHT that increases robustness to churn by increasing memory usage and communication overhead. Basically, peers are organized using a hash function into $\sqrt{N}$ groups where N = total number of peers. This is shown in Figure 1.20 with focus on the peer with ID=110 from group 0. The peer maintains (a) a view of its own group (i.e., peers 30 and 160), and (b) for each foreign group, a small (constant-sized) set of contact peers lying in it (i.e., peer 432). Each entry (group view



(a) Home model.

(b) Directory model.

Figure 1.19: DHT strategies in a P2P CDN.

or contact) carries additional fields such as RTT estimates. Peer 110 also stores directory information related to each single object that is cached in the system and whose URL maps to group 0 by means of hashing. For each such object o, peer 110 has a *directory table* that contains the IP addresses of a bounded set of peers holding a copy of o.

When a peer p downloads a copy of the object o, it creates a directory entry $< o, p >$ and communicates it to its contacts c that belong to o's group. When the directory table of c is full, c performs RTT measurements to keep the directory entries that refer to the closest peers and discard the other entries. Each peer gossips within its group to replicate and spread directory entries; it selects close-by peers from its view to exchange gossip messages. Obviously, peers gossiping and replicating directory entries are not necessarily interested in this information. Furthermore, since directory information is highly replicated, aggressive updates are required when referenced peers discard their content or leave the network.

Kache is robust against failures, because all peers in the same group store pointers of all the objects mapped onto the group. Moreover, locality-awareness is incorporated through the RTT-based routing tables. Lookups are bounded by $O(1)$, thus scaling does not influence lookup time. However, the resources necessary to maintain routing information increases as the number of peers increases.

1.5.3.6 Discussion

Table1.4 summarizes the performance behavior of the P2P-CDN approaches previously described. Obviously, none of them fully satisfy the requirements that we have identified along this chapter. An important observation is that most of the approaches do not focus

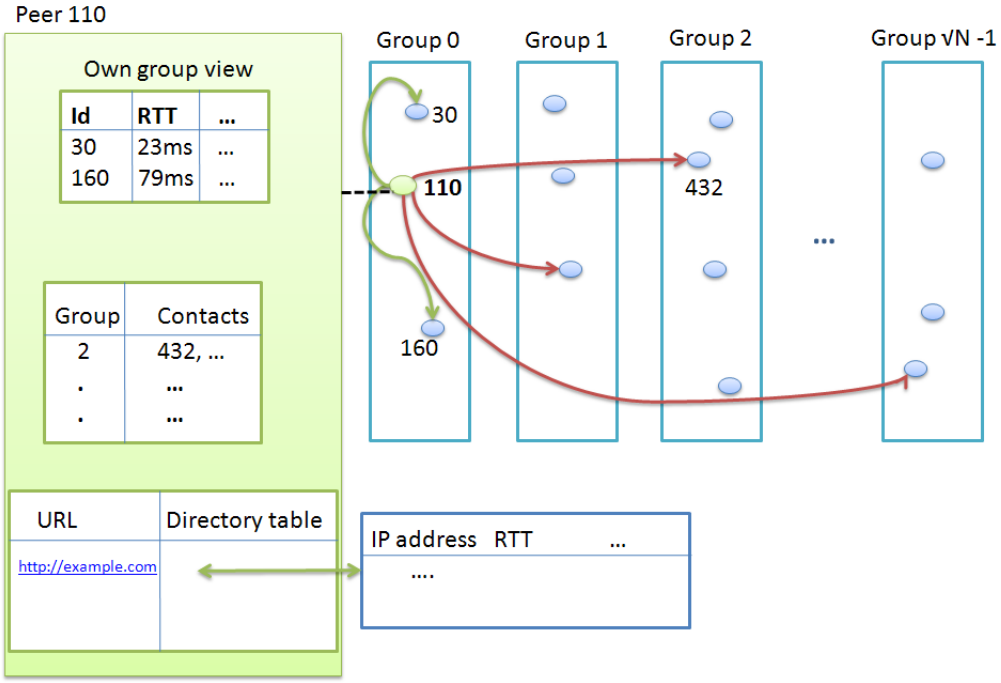


Figure 1.20: A Kache system with peers distributed across $\sqrt{N}$ groups, and soft state at a typical peer [LGB03].

on scalability, and often target small local networks.

In CoralCDN, users are not involved in the P2P network: they use the P2P CDN but do not contribute any resources to it. An increase of the number of users requires more investment by adding proxy caches to the CoralCDN. OLP is unsuitable for P2P systems as it is not scalable (i.e., bottlenecks) nor robust to churn (i.e., SPOF) due to its centralized nature, and it does not consider locality-awareness. CoopNet has similar limitations, except that it supports locality-aware redirection of queries. Proof derives its robustness from the randomness of unstructured overlays, but in return suffers from their scalability issues due to flooding overhead and lacks locality-awareness. BuddyWeb does not cope with dynamic and large-scale participation of peers because its construction mechanism is centralized, and thus is not adapted for real P2P environments. Kache addresses most of the requirements, and most importantly achieves robustness by replicating and gossiping indexing information. However, Kache scalability comes at the cost of a significant storage overhead on every peer. DHT-Directory approaches do not provide robustness as the performance of query handling is directly affected by peer failures. In comparison, DHT-Home aproaches rely on DHT robustness which incurs high costs and breaks the autonomy of peers. In addition, the aforementioned approaches do not specifically incorporate locality-awareness which is a major requirement of P2P-CDN.

SYSTEM	OVERLAY	ROBUSTNESS	SCALABILITY	LOCALITY	AUTONOMY
CoralCDN	hierarchy of proxies	yes	more proxy investment	yes	-
OLP	centralized	SPOF	server bottleneck	no	yes
CoopNet	centralized	SPOF	server bottleneck	no	yes
Proof	unstructured	randomness	flooding overhead	no	yes
BuddyWeb	unstructured	SPOF	server bottleneck	no	yes
Kache	gossip/DHT	replication	overhead	yes	yes
DHT-Directory	structured	directory loss	yes	no	yes
DHT-Home	structured	DHT robustness	yes	no	no

Table 1.4: Summary of P2P-CDNs

1.6 Conclusion

The objective of this chapter was to provide a consice, yet comprehensive study of P2P content distribution, with a view to building our own P2P CDN. For this, we reviewed the state-of-the-art for traditional and P2P content distribution in order to identify the shortcomings and highlight the challenges.

First, we identified the traditional requirements for CDNs which are performance, scalability and reliability, and we discussed the mechanisms needed to fulfill each requirement. We also shed light on CDN open issues, mainly in terms of scalabilty and its significant costs. We focused on the potential savings and benefits in using P2P technology as a cheap and efficient alternative for commercial CDNs.

Second, we explored P2P systems from the perspective of content sharing and shed light on the design requirements that are crucial to make efficient use of P2P self-scalability. The main relevant requirements are autonomy, expressiveness, efficiency, quality of service, and robustness.

Third, we presented the recent P2P trends that can improve the performance of P2P content distribution but incur additional challenges. The trends that we identified are locality-aware and interest-aware overlay matching, gossip usage and overlay combination. The challenges are to keep the solutions simple, avoid centralized management and large overheads, operate fast and dynamically adapt to changes and massive scales. Along these lines, matching the overlay with a locality- or interest-aware scheme could bring great benefits to the P2P system in terms of efficiency and quality of service. Another recent trend is the combination of different overlays and schemes, which can reveal very challenging. In particular, the maintenance of several overlays should not overwhelm the P2P system. Gossip protocols can serve as potentially effective means to achieve these new trends as it provides simplicity, decentralization and high robustness to churn. However, gossip should be properly designed and tuned to avoid significant delays and message overheads.

Finally, we focused on the two P2P applications that derive from content distribution, P2P file sharing and P2P CDN. We investigated both fields and reviewed existing approaches. In the context of file sharing, there is a growing concern about the network costs since the P2P traffic is the leading consumer of Internet bandwidth, mainly due

to search inefficiency and long file transfer. While the top priority is to exploit locality-awareness in order to serve queries from close-by locations, most existing works do not address this issue. Regarding P2P CDN, they have stringent performance requirements that are quire different to what is expected from a file-sharing system. They should be highly robust, efficient and scalable, while taking into account the autonomy of peers. Existing P2P CDNs do not answer all the important requirements. Most importantly, they are not designed to achieve high scalability as they target small scales. Therefore our thesis focuses on providing a complete solution for P2P content distribution that tackles the requirements by leveraging recent trends while answering the challenges.

Chapter 2

Locality-Aware P2P File Sharing

Abstract.Though widely deployed for file-sharing, unstructured P2P systems aggressively exploit network resources as they grow in popularity. The P2P traffic is the leading consumer of bandwidth, mainly due to search inefficiency, as well as to large data transfers over long distances. This critical issue may compromise the benefits of such systems by drastically limiting their scalability. In order to reduce the P2P redundant traffic, we propose Locaware, which performs index caching while supporting keyword search and keeping the overhead at bay. Locaware aims at reducing the network load by redirecting queries to available nearby results. For this purpose, Locaware leverages inherent properties like temporal locality and natural file replication and exploits locality-awareness.

2.1 Introduction

Despite the emergence of sophisticated topologies, file-sharing communities favor unstructured overlays due to their loose constraints. Indeed, in systems that mainly involve users from non-cooperating organizations, high maintenance is neither required nor affordable. Also, flooding-style search can be adept at content discovery and highly attract file sharing communities, especially because it allows a query to be expressed in a flexible manner rather than strictly requiring the exact filename. Unfortunately, as these systems grow in popularity, they aggressively exploit network resources, typically by consuming huge amounts of bandwidth. This issue severely threatens the scalability of unstructured file sharing systems. The main reason behind this is that search techniques tend to be

very inefficient, generating huge traffic and sometimes providing a bad user experience. In attempt to improve search efficieny and reduce the unnecessary traffic, index caching [PH03, Sri01, WXLZ06] is used in a way that limits the extent of flooding. However, most approaches have salient limitations because they trade either storage efficiency or flexibility for efficiency and most importantly do not address locality-awareness.

In this chapter, we propose an indexing solution for P2P file sharing, that supports keyword loookup, implements locality-awareness and aims at storage and maintenance efficiency. Our solution, called *Locaware*, brings twofold benefits:

- capture temporal locality of query workload and exploit natural replication in order to improve search efficiency and reduce redundant traffic.

- capture network locality of query workload in order to optimize file transfer and minimize bandwidth consumption.

To achieve these benefits, Locaware leverages inherent properties of P2P file sharing and combines a set of techniques that are simple, yet effective. The main contributions of this chapter are based on our material published in [DPV07,DP09] and can be summarized as follows:

- *locality-aware and selective index caching*: The technique aims at reducing the P2P unnecessary bandwidth consumption while achieving storage efficiency. A peer intercepts query responses and selectively caches several indexes per file, along with information about their physical locations. As a consequence, a peer answers a query by providing several possibilities. This approach aims at finding a nearby copy for the purpose of optimal file transfer, while it improves the file availability.

- *keyword-based query routing*: The technique allows locating relevant files and avoids flooding overhead. To support keyword queries, Bloom filters are used to compactly represent files, which enables some expressiveness and flexibility in the query formulation.

We evaluated the performance of our solution through simulation using PeerSim, and showed the effectiveness of Locaware in improving search efficieny and limiting bandwidth consumption. In short, Locaware improves success rate of selective index caching solutions by almost 30%. Further, it reduces transfer distance by 14 % compared to several other approaches.

Roadmap: The rest of this chapter is organized as follows. In Section 2.2, we first define the models of P2P File Sharing and index caching in order to identify the requirements and state the problem. Section 2.3 presents Locaware and describes its design and implementation. Section 2.4 depicts a performance evaluation of our solution through simulation. Finally, we conclude in Section 2.5.

2.2 Problem Definition

In this section, we first define the models of P2P file sharing and index caching on which is based this work. Then, we precisely state the problem we address in this chapter.

2.2.1 P2P File Sharing Model

Let us now define the assumptions we make on the P2P file sharing model. We assume an unstructured P2P network operating similar to Gnutella [Gnu05]. Peers are highly dynamic and autonomous, failing or leaving the network at any moment. They share files of any type specified by the application. To search for a file, a user employ in its query a string of keywords related to the target filename. The query is then flooded with a fixed TTL over the P2P network. Peers respond back to the query, with files that have all the keywords of the query in their filenames. A query response contains the filename and the IP address of a peer providing the requested file. Query responses follow the exact reverse path of their query, back to the requester peer. The latter downloads the file via direct connection with the provider peer and eventually becomes a provider for the file in question.

2.2.2 Index Caching Model

In order to limit redundancy and duplication, we consider a selective index caching technique where each peer maintains an index cache and selectively caches passing by query response. Thus, each entry of an index cache is a *file index* that associates a filename to the address of a provider peer. We group the group concept of DiCAS [WXLZ06] where each peer randomly chooses a group ID noted Gi (G$i_n \in [0\,..\,M-1]$ with M a system parameter). A Gi *matches* a filename F if the following condition is satisfied:

$$\mathrm{G}i = \mathrm{hash}(f) \bmod M \tag{2.1}$$

Query responses are only cached in peers whose group ID matches the hashing of the filename in the query response. We do not perform index caching based on keywords, as it results in less efficiency and more duplication as pointed out in Section 1.5.2.2 of Chapter 1.

2.2.3 Problem Statement

Our objective is to fully exploit the benfits of index caching in order to limit the wasted bandwidth. Here, we precisely state the requirements that have controlled our design choices.

- **Flexibility:** the index caching strategy should be coupled with a tecnhique that efficiently routes the query towards relevant file indexes. This technique should ultimately support keyword searches which are more prevalent and important than

exact-match queries in file-sharing systems. Recall that an exact-match query employs a string of keywords that is exactly conform to the requested filename, while a keyword query selects a subset of the filename keywords. The popularity of keyword queries stem from two reasons. First, the same file may be stored by different peers under slightly different names. Second, a user often queries a file using the partial filename.

- **Locality-awareness:** the approach should incorporate locality-awareness in order to optimize file transfer. In fact, the benefits of index caching can be significantly compromised if indexes randomly redirect queries while relevant files are available at nearby peers. That is why indexes could further limit the consumed bandwidth if they can give some indication about the physical location.

- **Availability:** the index cache should leverage the natural replication of files to provide query responses with more guarantees on file availability. Typically, a peer p caches for a file F an index referring to one provider peer and ignores all other providers of F. In consequence, p redirects all queries for F to the one provider peer it knows. This approach has two limitations. First, the index in question may quickly expire because the provider can disconnect or discard its file at any moment. Thus, the queries remain unsatisfied which leads to further and repetitive searches. Also, given the temporal locality of P2P queries, the provider peer may become quickly overloaded.

2.3 Locaware Design and Implementation

To fulfill the aforementioned requirements, we propose *Locaware*, a locality-aware approach for P2P file sharing networks. In the section, we present the detailed design of Locaware. First, we explain how Locaware uses Bloom filters for the purpose of keyword searches. Then we discuss the index caching startegy, and finally we present the algorithms supporting query search.

2.3.1 Bloom Filters as Keyword Support

In order to support keyword queries, we need a scheme to model the keywords of the filenames that are cached in an index cache. The idea is to summarize the set of keywords that can be found in an index cache in order to send the summarized information to the neighbors and guide their redirection of queries. A summary minimizes the update overhead because it is not affected by every single file addition or deletion. Furthermore, the requirements on a summary representation that is adapted to Locaware are small size and low false hit ratio. This is because these summaries need to be propagated to neighbors and thus should not incur a significant bandwidth overhead. Bloom Filters provide a straightforward mechanism to build such summaries.

2.3.1.1 Bloom Filters

A Bloom filter [Blo70] is a simple space-data structure for representing a set of elements S, in order to support membership queries. When querying a Bloom filter, it never returns false negatives but it may lead a false positive when it suggests that an element belongs to the set S even though it does not.

More precisely, a Bloom filter is a bit array of size m, constructed using k independent hash functions $h_j : S \rightarrow 1, ..., m$ with $1 \leq j \leq k$ (the unique assumption on the hash functions is randomness). The bits of the Bloom filter are initialized to zeros and set as follows. For each element $x \in S$, all bits in the positions $h_1(x), ..., h_k(x)$ of the Bloom filter are set to one. Thus, the Bloom filter can be used to check whether an element y belongs to the set S by using the k hash functions: one should only verify if all the bits at positions $h_1(y), ..., h_k(y)$ are set to one. If any of these bits is not set to one, then we can be sure that S does not contain y. If all the bits are set, then S may contain y. Figure 2.1 illustrates an example, where the represented set might contain "Joe", because bits 1, 3, 8 are set. The set definitely does not contain "Suzan" because bit 6 is false.

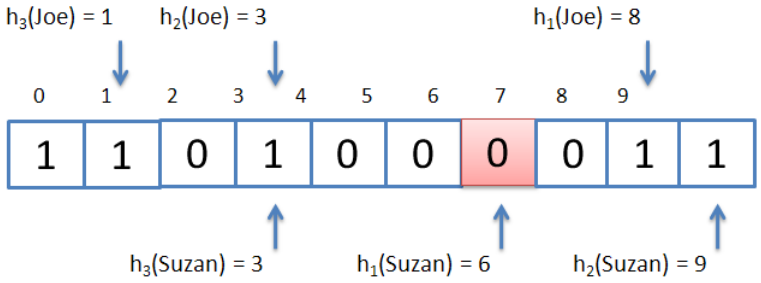


Figure 2.1: A Bloom filter of $m = 10$ bits.

The salient feature of Bloom filters is that there is a clear tradeoff between the storage amount allocated to the Bloom filter and the probability of false positives. The number of false positives falls exponentially as the size m of the Bloom filter increases. According to the analysis in [FCAB98], an optimal trade-off can be achieved for $(1/2)^k \simeq 0.6185^{m/n}$, where n is the size of the set represented by the Bloom filter. Thus, the false positive rate can be controlled at an acceptable level if the parameters are set appropriately.

To handle membership changes, we can maintain, for each bit i in the array, a count $c(i)$ of the number of times this bit has been set to 1. When an element x is inserted into or deleted from the set S, the counts $c(h_1(x)), ..., c(h_k(x))$ are incremented or decremented by 1 accordingly. Thus, bit i is turned on when its count $c(i)$ changes from 0 to 1. Also, the bit is turned off when its count changes from 1 to 0. In practice, allocating 4 bits per count is amply sufficient [FCAB98].

2.3.1.2 Maintaining a Bloom Filter for the Index cache

Each peer maintains a Bloom filter that represents the set of keywords of all cached filenames in its index cache. Whenever the peer caches a file index, it inserts each keyword

of the cached filename as an element of its Bloom filter. A Bloom filter BF *matches* a query $q = \{key_1, ..., key_n\}$ if the following condition is satisfied:

$$\forall key_i \in q;\ key_i \in BF \tag{2.2}$$

Neighboring peers exchange their group IDs. In addition, each one replicates its Bloom filter and sends a copy to each one of its direct neighbors. Thus, a peer stores on behalf of each neighbor its group ID and its BF. As a result, the peer can query its neighbors' Bloom filters to selectively route a query.

A Bloom filter is built incrementally as new filenames are inserted in the index cache and existing ones discarded. Updating the BF locally is done automatically since membership changes are supported by Bloom filters. Copies of BF held by neighbors must also be updated. A peer delays the propagation of its BF updates to its neighbors until the rate of new changes in its index cache reaches a threshold. The peer can either specify which bits in the bit array have flipped or send the whole array, whichever is smaller.

2.3.2 Locaware Index Caching

In order to provide more accurate and efficient responses to queries, we introduce locality-awareness in index caches, in terms of information about the network locality of the file provider. In addition, we exploit natural file replication, based on the fact that a peer which has recently requested a file F is likely to have it and can thereby serve subsequent requests for F. This aims at providing a peer with several indexes per file so that it can selectively answer a query according to the locality of the query originator. Below, we first introduce how we implement locality-awareness then we describe our index caching technique and its cache control policy.

2.3.2.1 Locality-Awareness

To model network localities in a simple way, we use the common binning technique [RHKS02] and adapt it to our context. We have chosen this technique as it meets the challenges of practicality, scalability and dynamicity as described in Section 1.4.0.1. Recall that peers can be grouped into virtual bins that reflect their network localities based on RTT measurements. We thereby associate to each possible bin a locality Id noted $locId$. Upon joining the network, each peer computes its own $locId$ and refreshes it regularly.

2.3.2.2 Locality-Aware Indexes

As introduced in Section 2.2.2, we assume that index caching is based on the hashing of the whole filename as in DiCAS. This means that a peer whose group ID matches a filename caches all passing-by distinct indexes of the corresponding file.

The index cache of a peer may hold for a cached filename, several provider addresses and their $locId$s (see Figure 2.2). To achieve this, a query response should contain both

the address and the $locId$ of the file provider. Additionally, it includes the address and the $locId$ of the query originator, which will be considered as a new provider by peers intercepting the responses. In other terms, a peer that is forwarding a query response checks if the vehiculated filename matches its group ID. If so, the peer extracts from the query response the address information about both the provider and the query originator to cache them. This is illustrated in Figure 2.2 where peers are assigned to three groups G1, G2 and G3 (i.e., $M = 3$). P2 requests the file whose name F matches G1; its query has reached a peer from G1 that has an index for F related to provider P1. The latter peer generates a query response $< F, P1, locId = 1 >$ and caches a new index for F related to the evenual provider P2. Then, peers of G1 forwarding the query response cache two indexes for F, one related to P1 and another to P2.

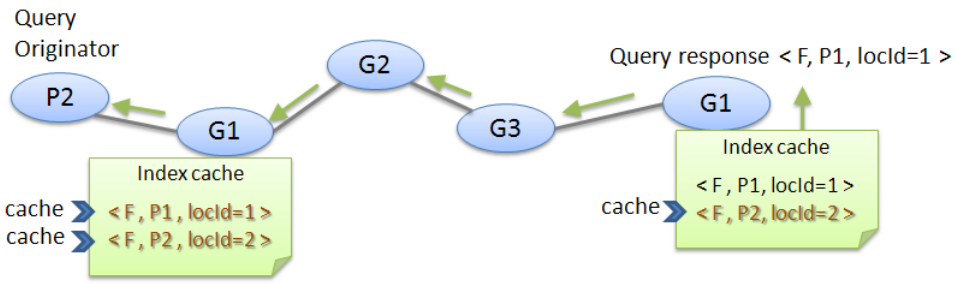


Figure 2.2: Locaware. Caching indexes of filename F; hash(F) mod M = G1.

2.3.2.3 Controlling the Cache Size

The cache size refers in our solution to the temporary storage space allocated at a peer for the index cache. Given the inherent heterogeneity of P2P systems, each peer contributes with a different amount of memory. The maximal amount of memory that a peer can invest is denoted by $maxMemo$. When its cache size surpasses its $maxMemo$, the peer discards some of its index cache content.

To perform an efficient and simplified cleanup, the peer determines the excess entries which has the lowest expected utility. It proceeds with the following cleanup consisting of two phases:

1. If there are more than two cached indexes for the same filename and $locId$, only the two most recently cached entries are kept while the rest is discarded.

2. If after the previous cleanup the cache size remains in excess, the peer searches for the filename with the largest number of entries and discards the ones that are the least recently cached.

The above strategy ensures a level of freshness in the response index while controlling its storage size. Further, it would tend to spread load rather evenly across the provider peers.

2.3.3 Locaware Query Searching

Locaware adapts query search to fully exploit its index caching strategy. Algorithm 1 describes how a peer processes and routes a query. As introduced previously, the query is propagated along with the address information of the query originator: the string of keywords is noted as q while the query originator as p_q. Upon the reception, the peer first performs a lookup over its index cache for a filename F that can satisfy q and extracts the set of indexes related to F (noted $I(F)$) (i.e., line 2). If such subset is found, the peer tries to select from $I(F)$ the indexes that refer to providers with the same $locId$ as the query originator (i.e., lines 3-7). Random indexes related to F might also be selected to have enough indexes in the query response (between 5 and 10). As a result, the query response would contain two subsets of indexes related to the requested F, one that complies with the $locId$ of the query originator and another that consists of random locIds for availability guarantees. This is to guarantee that the requester will find an available copy of its file despite peer failures and file deletions.

Algorithm 1 - process(q) at each peer

$I(F)$: set of cached indexes for filename F
$I(F, locId)$: set of cached indexes for filename F and specific $locId$
qr: query response that may be generated for q
$info(p_q) := \langle p_q.locId, p_q.address \rangle$

1: **receive** $\langle q, info(p_q) \rangle$
// Check local index cache
2: $I(F) \leftarrow index\_cache.$**lookup**$(q)$
3: **if** $I(F)$ is not empty **then**
4: $I(F, p_q.locId) \leftarrow I(F).$**lookup**$(p_q.locId)$
5: **if** $I(F, p_q.locId)$ is not empty **then**
6: $subset \leftarrow I(F, p_q.locId).$**select_subset**()
7: **end if**
8: $randomSubset \leftarrow I(F).$**select_subset**()
9: $qr \leftarrow \langle info(p_q), subset, randomSubset \rangle$
10: **send back** qr
11: **break**
12: **end if**

In case the peer did not find any index that can satify the query (i.e., $I(F)$ is empty), the query is forwarded to some of its direct neighbors (i.e., lines 13-22). To avoid missing relevant indexes that are cached at neighbors, a peer relies on the Bloom filters received

from its neighbors. It checks for each neighbor if the query matches the corresponding BF and accordingly redirects the query. In case none of its neighbors' BF match the query, the peer proceeds to a routing based on group IDs. If none of the previous routing strategies succeed, the peer forwards the query to the neighbor with the highest connectivity degree.

```
    success := false
    // Check neighbors' Bloom filters
13: for each neighbor i do
14:     if (BF_i.matches(q)) then
15:         send(q) to neighbor i
16:         success ← true
17:     end if
18: end for
19: if success then
20:     break
21: end if

    // Check neighbors' group IDs
22: for each neighbor i do
23:     if (groupId_i.matches(q)) then
24:         send(q) to neighbor i
25:         success ← true
26:     end if
27: end for
28: if success then
29:     break
30: end if

    // Select, as a last resort, neighbor c with highest connectivity
31: select neighbor c
32: send(q) to neighbor c
```

2.3.4 Storage and Bandwidth Considerations

In this section, we discuss the trade-off incurred by our design choices and more specifically their costs in terms of bandwidth and storage requirements.

2.3.4.1 About Bloom Filters Usage

A peer stores its Bloom filter as well as its neighbors' filters. A Bloom filter provides a trade-off between its memory requirements and its false positive ratio. Hence, given a response index with 50 filenames of 3 keywords in average, an optimal representation by

a Bloom filter needs a negligeable amount of memory, varying between 0.15 and 0.6 KB.[1] Since the average connectivity degree d of a peer is equal to 3 in Gnutella, then the average storage space allocated for Bloom filters at a peer is equal to $(3 + 1) * 0.15 = 0.6\,KB$, which is very small.

Recall that a peer propagates the updates of its Bloom filter to its direct neighbors. Bloom filter changes reflect filename additions and/or deletions in the corresponding response index. The update propagation is delayed until the percentage of new changes reaches a threshold α. Let f_{upd} be the update frequency which depends on the value α. The size of the Bloom filter[2] ,i.e., $1.2\,Kb$, is small enough to be transmitted at each update. Given that d is the average number of neighbors per peer, then at each update transmission, the peer sends d messages. Thus, the number of messages transferred per node and per second is $d * F_{upd}$ and the number of bits transferred per node and per second is $d * F_{upd} * 1.2\,Kb$ ($3.6 * F_{upd}\,Kb$ for $d = 3$). Since some staleness in the Bloom filter can be acceptable, we can tune α and thus F_{upd} in a way that significantly minimizes the update cost.

2.3.4.2 About Locality-Awareness

Recall that our approach involves caching multiple indexes per file. Hence, a query response can consist of several indexes pointing to different providers of the same file. In contrast, DiCAS is limited to one index per query response.

Concerning the storage requirements due to the extension of the response index, we have proposed a strategy to bound the cache size at each peer.

The transmission of I indexes associated to L locIds generates a larger query response. Let us show that it is still amply acceptable. Given the following parameters (4 bytes for an IP address, 1 byte for a filename and 7 bits = 0.875 bytes for a locId[3]), a query response is equal to $1 + 4 * I + 0.875 * L$ bytes. For $I = 5$ IP addresses divided between $L = 2$ locIds, the resulting traffic is limited to 22.75 bytes, i.e., $0.182\,kb$, which is insignificant compared to the huge size of the P2P shared files.

2.4 Performance Evaluation

In this section, we evaluate the performance of Locaware. First, we discuss our evaluation methodology based on the design goals of Locaware. Then, we present our experimenal setup and our experimental results. Finally, we summarize and highlight the pertinent lessons we have learnt.

[1] To support changes, 4-bit counts are added to a BF (i.e., extra memory = $0.6\,KB$). These counts are only stored locally and not propagated to neighbors

[2] $0.15\,KB$ is equivalent to $1.2\,Kb$ in data communication

[3] We focus on scalability with approximate topology information. Thus, a small number of landmarks should suffice us. For 5 landmarks, we get 120 possible locIds

2.4.1 Evaluation Methodology

Locaware design aims at improving index caching schemes and fully exploiting their benefits in the context of P2P-file sharing. To evaluate Locaware's performance, we examine the effectiveness of two main contributions: the keyword search support and the locality-aware provider selection. For this purpose, we compare the behavior of Locaware against three index caching schemes: unifom index caching (i.e., *UIC*), Dicas designed for filename search (i.e., *Dicas-filenames*) and Dicas designed for keyword searches (i.e., *Dicas-keywords*) (presented Section 1.5.2.1). Further, we need to check that Locaware reduces the traffic overhead incurred by flooding. That is why the comparison also covers the standard Gnutella flooding approach without caching (i.e., *Flooding*).

The efficiency of any search technique can be investigated under two angles: *user satisfaction* and *cost* [YGM02]. An efficient technique should seek to reduce the cost, mainly in terms of bandwidth consumption, which can be used to justify the algorithm scalability. At the same time, it should focus on optimizing user-perceived quality of service. Thus, we normally observe a trade-off between cost and user satisfaction.

To evaluate **user satisfaction**, we use the metric below:

- **success rate** or **hit rate**: the rate of queries successfully satisfied to all submitted queries. A query miss occurs if the search has not located any copy of a satisfying file as the number of search hops reached the fixed TTL. In respect to Locaware, success rate can reflect how efficient are the index caching and its adaptive search strategies in supporting keyword searches.

To quantify **cost**, we rely on two types of overhead:

- **search traffic**: the average number of messages produced by a query, including query response and download messages. This metric is highly significant since the main objective of index caching is to limit the search traffic.

- **transfer distance**: the average network distance over, in terms of latency, over which the object is transited when transferred from the provider peer to the requester peer. Higher distances generally involve more intermediate links and nodes to carry the traffic, which contributes to the aggregate network utilization and may overlaod the network. This metric quantifies the locality-awareness in the provider's selection. The lower is the transfer distance, the closer in locality are the provider and the requester, and consequently the lower is the bandwidth consumed by the file transfer.

2.4.2 Experimental Setup

We evaluate the performance of our proposed solution via simulation over PeerSim [JMJV], a Java-based simulator specifically tailored for P2P protocols.

A simulation consists of one experiment driven by cycles and events. The execution of an experiment is broken into cycles during which peers inject queries into the network and/or process events which are mainly incoming queries. We set the number of cycles

to 100 and map a cycle to 1000 ms. An experiment runs first an initialization phase that configures the network and the workload according to specific parameters.

2.4.2.1 Configuring the P2P Network

We generate an unstructured P2P overlay of 1000 peers with an average connectivity degree of 3. P2P networks are typically built on top of the Internet, which consists of nodes connected by links of variable latencies.In order to properly simulate a P2P environment, the simulator should reproduce this link heterogeneity. PeerSim has an event-driven framework that enables us to model the latency of each individual link. However, it does not provide support for simulating bandwidth and CPU resources. We generate an underlying network of nodes with links of latencies varying between 10 and 500 ms. To model latencies, our solution is inspired by BRITE [MLMB02] and works basically as follows. Peers are placed in a 2-dimentional Cartesian coordinate space, called *plane*. The latency between two peers is proportional to the geometrical distance between them on the plane.

To implement localities, we use 4 landmarks, which results in 24 possible locIds, because a larger number of landmarks will scatter the peers into many different localities. For instance, given 5 landmarks, i.e., 120 locIds, we only obtain an average of 8 peers with the same locId. In consequence, it would be quite difficult to find for a given requester peer, a provider with the same locId. Thus, a small number of landmarks is well-adapted to our simulation model. Furthermore, we adjust the provider selection strategy as follows: when a requester peer does not find a provider with matching locId amongst its received indexes, it measures its RTT to the set of available providers and chooses the one with the smallest RTT.

2.4.2.2 Configuring the Workload

We generate a pool of 3000 files and assign each one of them a filename that is a string of 3 keywords, randomly chosen from a pool of 9000. The intialization phase provides each peer with a repository of files and a query distribution. The repository contains 3 files that are randomly chosen from the pool and is shared with the P2P network. The query distribution maps which query the peer performs at each simulation cycle, at the rate of 0.00083 queries per second per peer. A query refers to a file chosen from the pool according to Zipf distribution and contains a string of one or more keywords randomly chosen from the filename in question.

A query search is bounded by a TTL equal to 7. Thus, it stops either if it reaches a file copy or index satisfying the query or if the number of hops attains the TTL. To set the size of the Bloom filter associated to each index cache, we look at the analysis done in [FCAB98]. We consider that an index cache contains at most 50 filenames of 3 keywords (the total number of indexes would be larger since several addresses could refer to the same filename). Thus, we set the Bloom Filter size to 1200 bits, which can provide an optimal representation with a negligeable amount of memory.

2.4.3 Experimental Results

We have conducted three experiments and examined the evolution of the aforementionned metrics (i.e., search traffic, success rate, transfer distance) as the number of queries in the system increases. Each experiment compares the performance of Locaware against the 4 approaches listed previously.

2.4.3.1 Search Traffic

The first experiment, illustrated in Figure 2.3, evaluates the search traffic. Locaware like Dicas approaches, reduces the search traffic wrt. flooding and UIC by 98% and 96%, respectively. Thus, Locaware preserves the main objective of selective index caching and routing by drastically limiting the search overhead. This acheivement is vital for scalability in P2P file sharing.

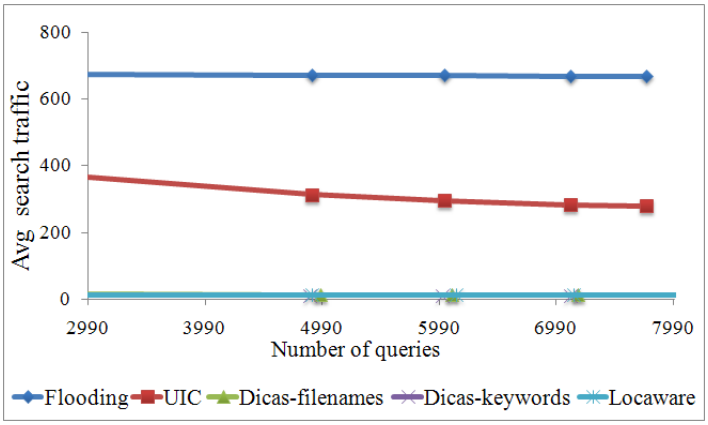


Figure 2.3: Search traffic evolution.

2.4.3.2 Success Rate

The second experiment, illustrated in Figure 2.4, evaluates the success rate. On the one hand, as expected, flooding and UIC outperform other techniques. Indeed, they provide each query with a large search scope at the cost of extensive traffic overhead. This is the trade-off yielded by selective routing schemes (like Dicas and Locaware) which restrict the number of peers visited by a query for the purpose of minimal overhead. In short, the loss in success rate is the price paid in order to make unstructured P2P file sharing systems scalable.

On the other hand, Locaware offers a substancial compensation over Dicas: it increases success rate by 21% wrt. Dicas-filenames and 29% wrt. Dicas-keywords. The main reason behind this significant improvement is that Locaware provides an efficient support for keyword queries and avoids missing results held by neighbors. In contrast, Dicas relies on the group ID based routing, which can often mislead keyword queries.

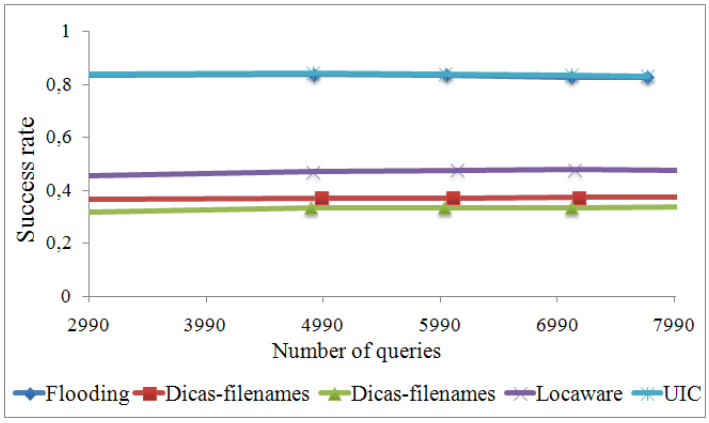


Figure 2.4: Success rate evolution.

Therefore, given that keyword queries are inherent to P2P file sharing, Locaware meets a fundamental requirement of these systems.

Based on these experimental results, Locaware seems well-adapted to P2P file sharing systems because it achieves the optimal trade-off between search efficiency (i.e., success rate) and overhead (i.e., search traffic).

2.4.3.3 Locality-Awareness

In the last experiment, we evaluate locality-awareness through measuring the transfer distance, which is shown in Figure 2.5. Thanks to Locaware, the average transfer distance is decreased by 14% compared to other approaches. Also, an interesting observation that could be derived from this experiment is that Locaware shows improvement with the increase of queries, unlike the other approaches which remain stable. This is because Locaware leverages the natural file replication to serve queries from close-by providers. In fact, as more queries for a particular file are generated and served, there will be more providers of this file available in different physical locations. Eventually, a query for this file is more likely to find a provider within the same locality as the requestor.

To further investigate locality-awareness, we measure the percentage of queries satisfied from the same locality as the requesters, or in other words the percentage of locality-aware file tranfers. The results are shown in Figure 2.6. Locaware realises 27% locality-aware tranfers, thus achieving an improvement of 40% compared to the other approaches.

Transfer distance have a salient impact on performance and scalability, especially that we deal with large file transfers. Lower distances limit the number of intermediate physical links and nodes that carry the burden of large data. This can significantly reduce the overall costs on the network and help in minimizing the response time perceived by the user.

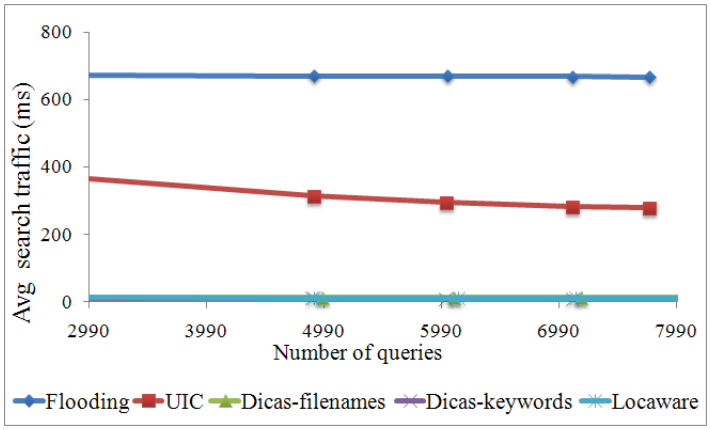


Figure 2.5: Transfer distance evolution.

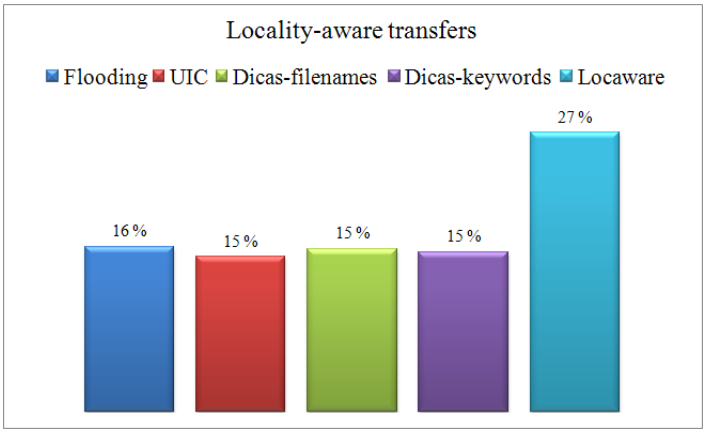


Figure 2.6: Distribution of locality-aware file transfers

2.4.4 Lessons Learnt

At the end of this chapter, we highlight the lessons we have learnt in our experiments and give insight on further direction improvements. The lessons are particularly pertinent to the usage of Bloom filters, the locality-awareness and the grouping of peers.

In respect to Bloom filters, we aimed at high efficiency, typicially by improving both response time and succcess rate. Despite the significant improvement achieved in success rate, the experiments revealed that Bloom filters were not fully profitable for query routing, i.e., they did not always provide significant assistance and information to guide a query. Two main reasons justify this observation. First, a Bloom filter can only give an optimal representation of the direct neighbors' index cache and cannot guide to further forwarding. By querying the neighbors' BF, we can mainly avoid missing requested indexes if they are cached at neighbors. Second, the Bloom filter optimality lowers the probability for the query to find the matching BF: peers do not often find neighbors with matching BF and thus end up using matching group IDs. Therefore, we should investigate

a more elaborate usage of Bloom filters in order to fully exploit their benefits.

Concerning locality-awareness in file transfer, we were motivated by its great impact on bandwidth consumption. However, we aspired for more improvement in transfer distance than the actual experimental results. In fact, the locality-awareness is not incorporated into query routing. The algorithm is not designed to find the good providers; it is limited to selecting the good provider when such candidate is available among the received indexes. Thus, we believe that the improvement would be much more significant with a locality-aware routing that focuses on efficiency finding close-by providers.

Finally, we believe that search can be more efficient if the groups are based on peer interests instead of being random. Each group can be associated to a specific profile and files can be mapped to groups based on their content (e.g., file topics). As a simple example, in the context of video files, a goup profile could refer to the topic "comedy" (or could combine several topics like "comedy" and "english"). A peer selects its group based on its preferences and acordingly caches file indexes that might interest it. Therefore, a peer can benefit from its index cache for its own queries before propagating them to neighbors. Although this scheme needs to be deeply explored, we think that it can achieve significant gains.

2.5 Conclusion

In this chapter, we focused on P2P file sharing in unstructured systems and addressed the problem of excessive bandwidth consumption. Our solution, Locaware, leverages inherent properties of P2P-file sharing environments, i.e., temporal locality and natural replication for more efficient search and less traffic, and network locality for lower bandwidth consumption in file transfer.

Locaware consists of a locality-aware and selective index caching with efficient support for keyword search. Basically, Locaware aims at improving the efficiency of finding nearby copies of the requested files. Further, its achievements are realized with a perfectly acceptable overhead in terms of storage and bandwidth requirements.

Through simulation, we showed that Locaware significantly improves the success rate of selective indexing caching solutions and reduces the traffic of flooding solutions. Most importantly, the results demonstrated that Locaware can limit wasted bandwidth and reduce network resource usage.

The results motivate us to elaborate more on Bloom filters and locality-awareness, in order to achieve greater performance improvement. On the one hand, the impact of locality-awareness could be more significant and its benefits intensified if exploited in query routing. On the other hand, Bloom Filters could be explored for more sophisticated search and caching techniques.

Chapter 3

Locality and Interest Aware P2P CDN

Abstract. The P2P model seems to be the perfect match to build a scalable and low-cost CDN. However, building a P2P CDN that can provide a performance similar to commercial CDNs, can reveal very challenging, as it involves autonomous and volunteer participants. In this chapter, we propose a P2P-CDN called Flower-CDN, that supports under-provisioned websites with large user-base, by strictly relying on their user communities rather than dedicated and reliable servers. To achieve this, we cope with the autonomous behavior of peers by leveraging their interests. We incorporate locality-awareness to enable low-cost collaborations and redirect queries to close-by content. Flower-CDN infrastructure efficiently combines DHT and gossip protocols, interest and locality-based schemes, and exploits their advantages while avoiding their limitations.

3.1 Introduction

Content Distribution Networks (CDN) are well-known technologies for distributing the content of web-servers to large audiences. The main mechanism is to replicate requested content at strategically placed dedicated machines. As they intercept and serve the clients' queries, these technologies decrease the workload on the original web-servers, reduce bandwidth costs, and keep the client's perceived latency low. Unfortunately, non-profit websites (e.g., related to charities, social organizations, scientific associations, etc.) often cannot afford the expenses of deploying and administrating a dedicated CDN

infrastructure. Nevertheless, such websites often attract substantial loads, either due to their international audience or by being referenced by other popular websites. Thus, their under-provisioned servers become easily overloaded with queries and may fail to maintain an acceptable quality of service to their clients. Furthermore, remote clients experience long latency even if the server is not overloaded. Thus, what these websites need is a distributed content distribution infrastructure that can quickly deliver the content at large scale but that does not imply the large costs of traditional CDNs.

We believe that the P2P model is a perfect match to build such a scalable and cheap CDN for popular and under-provisioned websites by exploiting the underutilized resources of their user communities. In fact, many projects have demonstrated that users are willing to contribute to organizations whose cause they support (e.g., fund-raising and editing in Wikipedia, sharing idle computer resources in SETI@home, etc.).

Our basic idea is simple and conceptually similar to file-sharing applications: After a peer has retrieved a web-page, it caches it and provides it to other peers that request it. Thus, once a web-page is cached by peers, successive requests can be served from the P2P network, alleviating the load on the web-server. However, CDNs have stringent performance requirements that are quite different to what is expected from a file-sharing system. Any CDN has to focus on two performance metrics: *response time* and *hit ratio*. Traditional CDN replicate most of the content at strategic locations and thus, the CDN can serve many client requests leading to a high hit ratio. Additionally, response times are short if efficient routing algorithms find replicas close to the client in network locality. Traditional CDN generally incorporate *locality-awareness* into their query routing mechanism as it has the potential to dramatically reduce response times as well as bandwidth consumption and thus increase system scalability.

However, building a P2P-based CDN is not a straightforward endeavor. The system should respect the autonomy of peers, typically by assigning them duties and placing content at them according to their interests. Otherwise, peers will not have enough incentives to cooperate. Finally, we have to make locality-awareness a top priority in order to achieve short query response times.

Considering all these issues, we propose a locality- and interest-aware P2P CDN, *Flower-CDN*, that enables any under-provisioned website to efficiently distribute its content, with the help of the non-profit community interested in its content. Our solution exhibits several unique characteristics that enable us to overcome all of the above mentioned challenges. It combines the strengths of both structured and unstructured P2P networks, exploiting DHT efficiency and gossip robustness. The content of this chapter is mainly based on our material published in [DPK09a].

- Flower-CDN introduces a novel DHT usage and management, called D-ring, that relies on a new locality- and interest-aware key service. It helps new peers to quickly find peers in the same locality that are interested in the same website.

- We propose the organization of peers that share the same locality and are interested in the same website into unstructured overlay clusters (called *petals*). Within a petal, peers use gossip protocols to exchange information about their content and contacts,

allowing Flower-CDN to maintain accurate information despite dynamic changes in order to support eventual queries.

- We use this novel two-layered architecture consisting of a D-ring and petals to provide hybrid locality-aware query routing. The D-ring ensures a reliable access for new clients, while subsequent searches are performed within the petals. Thus, most of the query routing takes place within a local cluster leading to short query search and local data transfer.
- We present a cost analysis of our gossip protocols. The analysis aims at quantifying the overhead and guiding the configuration of gossip parameters.
- Finally, we present a detailed performance evaluation. Our experimental results show that Flower-CDN can reduce lookup latency by a factor of 9 and the transfer distance by a factor of 2, compared to an existing P2P CDN. Moreover, Flower-CDN incurs very acceptable overhead in terms of gossip bandwidth, which can also be tuned according to hit ratio requirements and bandwidth availability.

Roadmap. We organize the remainder of this chapter as follows. Section 3.2 provides an overview of Flower-CDN. Section 3.3 extensively describes the D-ring model with its different features and services, while Section 3.4 presents the Petal Model and its gossip-based management. The design choices of Flower-CDN are justified in Section 3.5. A cost analysis that focuses on the gossip overhead of our approach is given in Section 3.6. Simulation methodology and results are presented in Section 3.7 before concluding in Section 3.8.

3.2 Flower-CDN Overview and Preliminaries

Flower-CDN is designed to support a set W of websites ws, each of which has its own requestable content (e.g., set of web-pages and documents). A website ws is supported by Flower-CDN as long as there are a sufficient number of clients willing to participate on behalf of ws in order to enjoy a better access for the content of ws.

We implement locality-awareness in Flower-CDN as we did for Locaware in Chapter 2 via the binning technique [RHKS02]. A peer measures its RTT to a set of well-known landmarks spread across the network; and orders them by increasing latency. Physically close peers are likely to have the same landmark ordering. Thus, each possible ordering identifies a locality loc: $1 \leq loc \leq k$ with k the total number of localities.

Figure 3.1 illustrates the architecture of Flower-CDN. Participant peers belonging to the same locality loc and interested in the same website ws build together an unstructured overlay noted **petal**(ws, loc), using gossip protocols. These peers, called *content peers* and noted $c_{ws,loc}$, cache, manage and exchange content of ws, thus considerably relieving the server of ws from its query load. Flower-CDN charges one peer of each $petal(ws, loc)$, the role of a *directory peer* (noted $d_{ws,loc}$): $d_{ws,loc}$ knows about all content peers $c_{ws,loc}$ and keeps information about their stored content.

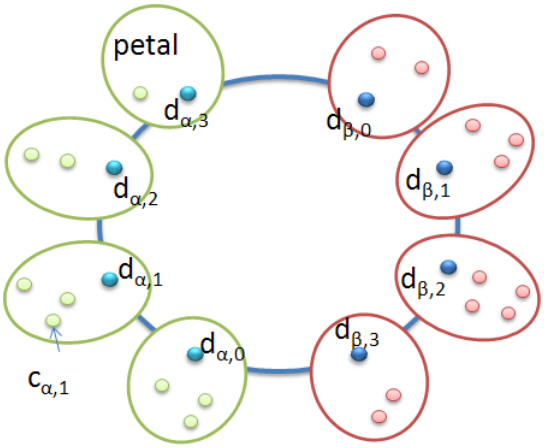


Figure 3.1: Flower-CDN architecture with websites α and β and four localities.

Directory peers are also embedded in **D-ring**, a structured overlay based on a *Distributed Hash Table* (*DHT*), to support queries coming from new clients, that request objects of W for the first time. That is Flower-CDN relies on a hybrid architecture consisting of a set of independent petals linked via one directory overlay (i.e., D-ring).

Instead of querying server ws, a new client located in loc, submits its query to D-ring and gets directed to the directory peer in charge of ws in loc i.e., $d_{ws,loc}$. Then, $d_{ws,loc}$ tries to resolve the query while relying on its petal or some neighboring petals related to ws. The query is hence redirected to some content peer $c_{ws,loc}$ that holds the requested object; $c_{ws,loc}$ *serves* the query, i.e., it directly transfers the object to the client. Then, the client can join $petal(ws, loc)$ as a content peer $c_{ws,loc}$, if it is willing to contribute storage resources with respect to the content of ws. For further queries, $c_{ws,loc}$ searches directly in its $petal(ws, loc)$ instead of relying on D-ring.

In summary, all peers that are willing to support a certain website $ws \in W$ become part of one of the petals of ws helping ws to distribute its content. We denote this set of peers as P_{ws}:

$\forall ws \in W : P_{ws} = \bigcup_{0 \leq loc < k} petal(ws,loc)$.

Peers interested in this same website and thus dealing with the same content, may exchange summaries of their content to help searching for queries.

3.3 D-ring Model

The directory overlay *D-ring* is a structured overlay with a novel DHT mechanism that leverages interests and network localities of peers to construct the overlay and efficiently route queries. In this section, we first describe the different architectural aspects of D-ring (i.e., key management and directory structure), then we discuss the functionality of

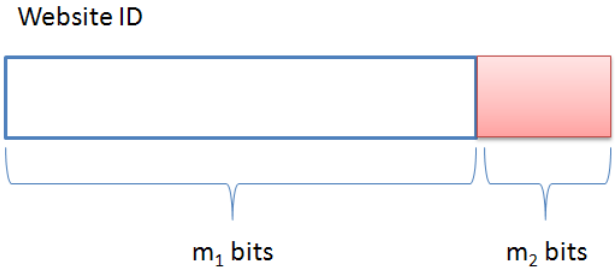


Figure 3.2: Peer ID structure in D-ring.

D-ring which consists of a P2P directory service.

3.3.1 Key Management

In order to ensure a fast lookup, D-Ring can be integrated into any existing structured overlay based on a standard DHT (e.g., Chord [SMK+01], Pastry [RD01a]). For each website $ws \in W$, the directory overlay enables k participant peers from P_{ws}, where k is the number of localities, to join as directory peers for ws: each locality loc is covered by a directory peer $d_{ws,loc}$, to empower locality-aware redirection of queries. In the example of Figure 3.1, Flower-CDN covers 2 websites α and β and 4 localities, i.e., $k = 4$. Thus, both websites α and β have 4 directory peers in D-ring.

In DHT-based systems, peer identifiers (noted ID) are chosen from an identifier space $S = [1 \cdots 2^m - 1]$; where m is the ID length in bits. Based on these identifiers data placement is then typically determined by a hash function which maps data identifiers to peer identifiers. That is, every object receives a key, and the peer with the ID closest to the object key is responsible for storing the object or pointers to the locations of object replicas. When a client looks for an object with a given key, it now contacts any peer in the DHT and the request is routed through the DHT until the peer with the ID closest to the object key is found. This routing service takes typically in the order of $log(n)$ hops where n is the number of peers in the DHT.
In Flower-CDN, we do not want to map data items to peers but we want that a query for website ws posed by a peer in locality loc quickly finds the directory peer $d_{ws,loc}$. To achieve this and exploit the existing DHT infrastructure, we only have to assign a directory peer a very specific peer ID, namely an identifier based on the website and locality it represents. As shown in Figure 3.2, the m bits of a peer ID are split into 2 segments, a *website ID* and a *locality ID*:

- **locality ID**:
  - identifier of the locality to which the directory peer belongs. It is expressed using the lowest bit-segment of length m_1.

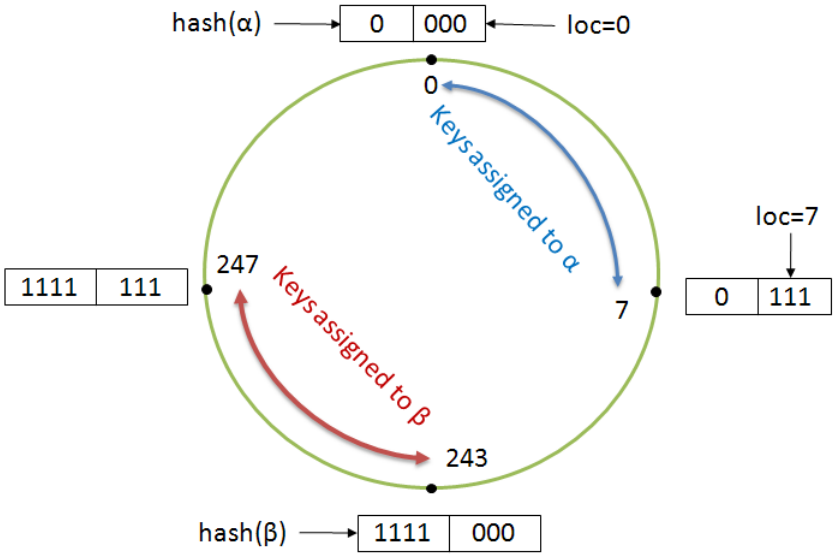


Figure 3.3: D-ring distribution of keys given that $k = 8$ and $W = \{\alpha, \beta\}$.

- A locality is mapped to an ID between $[0 \cdots k-1]$; m_1 should be chosen such that $2^{m_1} \geq k$.

- **website ID**:
  - identifier of the website which the directory peer serves. It is expressed using the highest bit-segment of length $m_2 = (m - m_1)$.
  - The website ID related to ws is obtained uniformy at random from the the subspace $S' = [1 \cdots 2^{m_2} - 1]$. The identifier is obtained by hashing the url of ws (noted $hash(ws)$).

Directory peers in the same locality have the same locality ID. Moreover, directory peers for the same website have the same website ID; they have successive peer IDs and therefore are neighbors on D-ring. As shown in Figure 3.1, for website β, $d_{\beta,0}$ is succeeded by $d_{\beta,1}$, then $d_{\beta,2}$, etc. The same order applies to website α. If a query for an object of website ws is now submitted to D-Ring from locality loc, it is not the object key that is the input for the DHT routing service. Instead the search key is the concatenation of ws and loc. The underlying DHT infrastructure will then find $d_{ws,loc}$ as its peer ID exactly matches the search key.

An example is given in Figure 3.3 with $k = 8$, $W = \{\alpha, \beta\}$, 4 bits for the website ID and 3 bits for the locality ID. With $hash(\alpha) = 0$, the website ID related to α is 0. To obtain the range of peer IDs assigned to the directory peers of α, we vary the locality ID from 0 and 7 (i.e., $(k-1)$) and concatenate it to the website ID of α. Thus, peer IDs and

search keys for α range between 0 and 7. Similarly, with $hash(\beta) = 15$, keys for β range between 240 and 247.

3.3.2 Directory Tools

In the following, we use the notation d_{ws,loc_i} when we need to differentiate between directory peers of the same website ws wrt. different localities. Besides its DHT-based routing table, a directory peer d_{ws,loc_i} maintains:

1. *Directory-index*(ws, loc_i): a directory that indexes the content of ws stored in *petal*(ws, loc_i). The directory contains an entry for each content peer c_{ws,loc_i}, consisting of 3 fields:

   - information about the address of c_{ws,loc_i} (e.g., IP address)
   - age field useful for failure and leave detection (presented in Sec. 3.4.1)
   - list of object identifiers (e.g., hash(url)) describing the content held by c_{ws,loc_i}

   We say that d_{ws,loc_i} has a complete *view* of *petal*(ws, loc_i), represented by its directory-index.

2. A small set of *Directory-summaries*(ws, loc_j): these are summaries of directory-indexes maintained by other directory peers d_{ws,loc_j} $(i \neq j)$. d_{ws,loc_j} refers to any other directory peer of ws that d_{ws,loc_i} knows via its routing table. *Directory-summary*(ws, loc_j) is represented by a Bloom filter, in a similar way as has been done for cache summaries in [FCAB98], using the identifiers of the objects listed in *directory-index*(ws, loc_j).

Figure 3.4 shows a part of D-ring and focuses on the directory peer $d_{\beta,1}$ and three content peers for $(\beta, 1)$, namely A, B and C. $d_{\beta,1}$ maintains *directory-index*$(\beta, 1)$ that lists, for each peer in *petal*$(\beta, 1)$, their objects (e.g., A holds objects x and y which are initially provided by website β). Moreover, $d_{\beta,1}$ stores directory summaries received from its direct neighbors i.e., $d_{\beta,0}$ and $d_{\beta,2}$.

3.3.3 P2P Directory Service

D-ring acts as a P2P directory service for clients wishing to use and contribute to Flower-CDN. Mainly, it provides two functionalities. First, it supports first queries coming from new clients of W and handles them instead of the original webservers. Second, D-ring serves as a reliable access to Flower-CDN for those new participants: by routing its first query over D-ring, a client is guided to the petal related to its locality loc and its interest ws and thus joins as a directory peer or content peer.

Based on the standard DHT routing service, D-ring routes query messages targeting a website ws and a locality loc using a key composed of the website ID of ws and the locality ID of loc (noted $ID_{ws,loc}$). Given that $ID_{ws,loc}$ also represents the ID of $d_{ws,loc}$

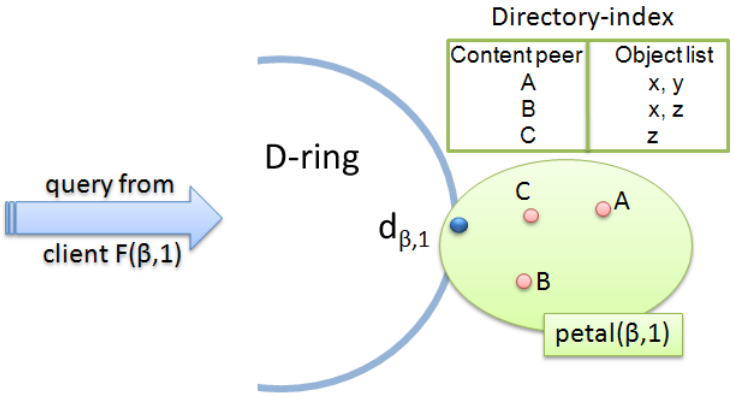


Figure 3.4: Query submitted by F, a new client of β in locality $loc = 1$.

(cf. Sec.3.3.1), the message is normally delivered to the target directory peer $d_{ws,loc}$. In case $d_{ws,loc}$ has not joined D-ring yet, the message reaches one of its direct neighbors on D-ring (i.e., which has the numerically closest ID to $ID_{ws,loc}$).

A new client of website ws that is located in loc routes its first query over D-ring using $ID_{ws,loc}$. In case the directory peer in charge of ws wrt. loc (i.e., $d_{ws,loc}$) does not exist, the new client joins D-ring to be $d_{ws,loc}$ using the standard DHT join procedure (see Sec.4.3 for a deep explanation). Otherwise, the new client joins $petal(ws, loc)$ as a content peer via the existing directory peer. Below, we first detail how a query of a new client is handled by an existing directory peer, then, we discuss how the client joins its petal as a content peer.

3.3.3.1 Query Processing

Consider $query(o_{ws})$, a query that is submitted by a new client and that requests an object of the content of ws noted o_{ws}. Upon receiving $query(o_{ws})$, d_{ws,loc_i} processes it as shown in Algorithm 2. d_{ws,loc_i} searches first its directory index for the requested object o_{ws}. If $directory\text{-}index(ws, loc_i)$ shows that o_{ws} is stored by some content peer c_{ws,loc_i}, d_{ws,loc_i} redirects $query(o_{ws})$ to c_{ws,loc_i} after checking its aliveness. Otherwise, d_{ws,loc_i} queries the directory summaries, to check if some d_{ws,loc_j} might have the requested object in its directory index. In case d_{ws,loc_j} is found, $query(o_{ws})$ is redirected to d_{ws,loc_j} which proceeds with **process**$(query(o_{ws}))$. When no satisfying directory or content peer is found, $query(o_{ws})$ is redirected to the website ws.

Figure 3.4 shows a new client F of website β with a query q for object x. Assuming that client F is located in $loc = 1$, q is forwarded to $d_{\beta,1}$ which searches its directory index for x. Then, $d_{\beta,1}$ redirects q to content peer A or C, which holds a copy of x and thus can serve the query. If F requests object x' which is not contained by any peer in $petal(\beta, 1)$, $d_{\beta,1}$ first checks its *directory-summaries* for $(\beta, 0)$ and $(\beta, 2)$ to see if they might have x'

in their directory index. If it appears so, $d_{\beta,1}$ redirects q accordingly to either $d_{\beta,0}$ or $d_{\beta,2}$. Otherwise, $d_{\beta,1}$ redirects q to the website β.

Algorithm 2 - process(*query*(o_{ws})) at d_{ws,loc_i}

```
c_{ws,loc_i} ← directory-index(ws, loc_i).lookup(o_ws)
if c_{ws,loc_i} != null and c_{ws,loc_i} is alive then
    redirect query(o_ws) to c_{ws,loc_i}
else
    d_{ws,loc_j} ← directory-summaries.lookup(o_ws)
    if d_{ws,loc_j} != null and d_{ws,loc_j} is alive then
        redirect query(o_ws) to d_{ws,loc_j}
    else
        redirect query(o_ws) to ws
    end if
end if
if sameWebsite(d_{ws,loc_i}, client) == true and sameLocality(d_{ws,loc_i}, client) == true
then
    directory-index(ws, loc_i).add(client, o_ws, 0)
end if
```

3.3.3.2 Joining the Petal

After processing its query, the client interested in ws and located in loc joins $petal(ws, loc)$ as a content peer $c_{ws,loc}$. As shown in the end of Algorithm 2, the appropriate $d_{ws,loc}$ adds a new entry in its directory index: the client with its requested object and age zero. Furthermore, the client is provided with a list of contacts from its petal to achieve its integration. The next section brings more insight into this issue.

3.4 Petal Model

As previously introduced, $petal(ws, loc)$ consists of a directory peer $d_{ws,loc}$ and several content peers $c_{ws,loc}$, all of which reside in locality loc and are interested in the content provided by ws. $Petal(ws, loc)$ expands progressively as more clients of ws in loc join Flower-CDN.

Each $petal(ws, loc)$ provides a search infrastructure for queries of content peers $c_{ws,loc}$. Once a client has become a content peer $c_{ws,loc}$, any subsequent queries that the client poses for website ws directly use $petal(ws, loc)$ instead of D-ring. For this purpose, within the petal, content peers gossip to exchange and discover other content peers $c_{ws,loc}$ and summaries of their stored content (more details are given in Sec. 3.4.1). Hence, $c_{ws,loc}$ can search the summaries of its $petal(ws, loc)$ to see where a copy of its requested object might be stored. In the remaining of this section, we describe how a petal is managed via gossip protocols. Then, we present how a query is processed within a petal.

3.4.1 Gossip-Based Management

Gossip-style communication is used throughout a petal to disseminate summaries and their updates in an epidemic manner. Peers also gossip to discover new members in their overlay and to detect failed ones. We chose gossip-style communication for 3 reasons. First, it enables robust self-monitoring of clusters: each peer is in charge of monitoring a few random others, sharing the monitoring cost and thus ensuring load fairness [VGS05]. Second, it eases information dissemination, such that peers discover new content and new peers providing some content [EGKM04]. Finally, it is easy to deploy, robust and resilient to failure.
Basically, gossip proceeds as follows: a peer p_i knows a group of other peers or *contacts*, which are maintained in a list called *p_i's view*. Periodically (with a gossip period noted T_{gossip}), p_i selects a contact p_j from its view to gossip: p_i sends its information to p_j and receives back other information from p_j. The gossip algorithm used in Flower-CDN is inspired by gossip-based approaches for P2P membership management, such as [VGS05].

3.4.1.1 Gossip Tools

To support gossip, each $c_{ws,loc}$ locally manages the following elements:

1. *content-list*($c_{ws,loc}$): a list of the object identifiers of the content currently held by $c_{ws,loc}$. The list is used during gossip exchanges in two ways:
   - current *content-summary*($c_{ws,loc}$): a summary of the current *content-list*($c_{ws,loc}$) built using a Bloom filter.
   - Δ*list*($c_{ws,loc}$): a sublist that reflects the new changes in the list (i.e., object deletion or insertion) wrt. a threshold of changes (detailed later in this section)
2. *view*($c_{ws,loc}$): a partial view of *petal*(ws, loc), which contains a fixed number V_{gossip} of entries, each one referring to some other $c'_{ws,loc}$. A view entry referring to a contact $c'_{ws,loc}$ contains 3 fields:
   - information about the address of $c'_{ws,loc}$ (e.g., IP address)
   - age: numeric field that denotes the age of the entry since the moment it was created (not an indication of $c'_{ws,loc}$'s lifetime)
   - *content-summary*($c'_{ws,loc}$)

Whenever $c_{ws,loc}$ gossips with $c'_{ws,loc}$, $c_{ws,loc}$ updates the entry related to $c'_{ws,loc}$ in *view*($c_{ws,loc}$) as follows: the age of $c'_{ws,loc}$ is set to zero, and a current *content-summary*($c'_{ws,loc}$) is received from $c'_{ws,loc}$; thus the age zero refers to the most recent entry status. Periodically (i.e., with period T_{gossip}), $c_{ws,loc}$ increments by 1 the age of all its view entries. Thus, a high age reflects that $c_{ws,loc}$ has not heard recently about $c'_{ws,loc}$ in order to refresh its view entry.

When $c_{ws,loc}$ joins *petal*(ws, loc), *view*($c_{ws,loc}$) is initialized upon its first contact with a peer from its petal (i.e., another $c'_{ws,loc}$ or $d_{ws,loc}$). In Figure 3.4, the new client F that

has contacted $d_{\beta,1}$ for a query, may initialize its view in two different ways. In case its query is served from some $c_{\beta,1}$ (e.g., A), F's view is initialized from a subset of A's view. In all other cases (i.e., query served from ws or $petal(\beta, 2)$), it is $d_{\beta,1}$ that provides F with a subset of its view; then, F's initial view will not have content summaries but will progressively fill them via gossip exchanges.

3.4.1.2 Gossip Behavior

The gossip behavior of each content peer $c_{ws,loc}$ is illustrated in Algorithm 3: the active behavior describes how $c_{ws,loc}$ initiates a periodic gossip exchange, while the passive behavior shows how $c_{ws,loc}$ reacts to a gossip exchange initiated by some other content peer $c''_{ws,loc}$. For simplicity, we refer to $view(c_{ws,loc})$ in the algorithm by $view$.

Algorithm 3 Gossip behavior of $c_{ws,loc}$

```
// active behavior
loop
    wait(T_gossip)
    view. increment_age()
    c'_{ws,loc} ← view.select_oldest()
    viewSubset ← view.select_subset()
    gossipMsg ← ⟨content-summary(c_{ws,loc}), viewSubset⟩
    send gossipMsg to c'_{ws,loc}
    receive gossipMsg' from c'_{ws,loc}
    viewEntry ← ⟨c'_{ws,loc}, 0, content-summary(c'_{ws,loc})⟩
    buffer ← merge(view, gossipMsg'.viewSubset, viewEntry)
    view ← buffer.select_recent()
end loop

// passive behavior
loop
    waitGossipMessage()
    receive gossipMsg'' from c''_{ws,loc}
    viewSubset ← view. select_subset()
    gossipMsg ← ⟨content-summary(c_{ws,loc}), viewSubset⟩
    send gossipMsg to c''_{ws,loc}
    viewEntry ← ⟨c''_{ws,loc}, 0, content-summary(c''_{ws,loc})⟩
    buffer ← merge(view, gossipMsg''.viewSubset, viewEntry)
    view ← buffer.select_recent()
end loop
```

The active behavior is launched after each time interval T_{gossip}. After incrementing the age of its view entries, $c_{ws,loc}$ selects from its view: (1) $c'_{ws,loc}$, the oldest contact via **select_oldest**() and (2) *viewSubset*, a random susbet of L_{gossip} view entries (

$0 < L_{gossip} \leq V_{gossip}$) via **select_subset**(). Then, $c_{ws,loc}$ sends to $c'_{ws,loc}$ $gossipMsg$, a message that contains *viewSubset* and a current *content-summary*($c_{ws,loc}$). $c_{ws,loc}$ receives in exchange, $gossipMsg'$ containing similar information from $c'_{ws,loc}$; $c_{ws,loc}$ creates $viewEntry$, a view entry related to $c'_{ws,loc}$, with the age 0 and the current summary of $c'_{ws,loc}$. The procedure **merge**() collects in a buffer all the entries from both the local view and the received information from $c'_{ws,loc}$, and discards the duplicates: if 2 entries related to the same contact exist, only the instance with the smallest age value is kept. Then, the procedure **select_recent**() selects the most recent V_{gossip} entries from the buffer i.e., the ones with the smallest age values, in order to limit the view size to V_{gossip}.

The passive behavior is triggered when $c_{ws,loc}$ receives a gossip messsage containing summary and view information from some content peer $c''_{ws,loc}$. Then, $c_{ws,loc}$ answers by sending back a gossip message with its own summary and view information, and updates its local view via **merge**() and **select_recent**() as described previously.

Through both active and passive behaviors of Algorithm 3, $c_{ws,loc}$ and its gossip partner, i.e., $c''_{ws,loc}$ or $c'_{ws,loc}$, exchange their current content summaries; they add new view entries of each other in their local views or refresh the existing ones in case they already know each other.

3.4.1.3 Push Behavior

Recall that the first access to $petal(ws, loc)$ is provided by D-ring via its directory peer $d_{ws,loc}$ which maintains a complete view (or directory-index) of its petal. $d_{ws,loc}$ handles first queries of new clients targetting $petal(ws, loc)$ and may provide them, in some cases, an initial view of $petal(ws, loc)$ to allow them to integrate it.

To maintain the *director-index(ws,loc)* up-to-date, each content peer $c_{ws,loc}$ needs to regularly communicate with $d_{ws,loc}$. For this purpose, $c_{ws,loc}$ keeps track of the current $d_{ws,loc}$ and maintains, in its view, a special entry for $d_{ws,loc}$ that only contains its address and its age information (noted *dir-info*). $c_{ws,loc}$ periodically increments the age of *dir-info*, as it does with all its view entries. $c_{ws,loc}$ sends its *dir-info* along with every gossip message sent to another content peer. This process spreads continuous updates about the directory peer throughout its petal, which also serves to detect its failure and ensure the recovery (further explanation is given in Sec. 4.3.2).

Given that a content peer may request and access new content, $c_{ws,loc}$ sends updates about its newly stored objects to $d_{ws,loc}$, using *push messages*. As depicted in Algorithm 4, $c_{ws,loc}$ monitors the changes (i.e., the newly stored objects) in *content-list*($c_{ws,loc}$) noted *list* for simplicity; whenever the percentage of new changes reaches a predefined threshold, $c_{ws,loc}$ creates $\Delta list$ to be pushed to $d_{ws,loc}$ (via **extract_changes**()). Then, $c_{ws,loc}$ resests to 0 its age field of $d_{ws,loc}$. Further, object evictions due to cache expiration or replacement policies are reported to $d_{ws,loc}$ as new changes via push messages.

As shown in Algorithm 5, $d_{ws,loc}$ periodically increments the age fields of its view entries. Upon the reception of a push message from $c_{ws,loc}$, $d_{ws,loc}$ resets to zero the age of $c_{ws,loc}$'s entry in *directory-index*(ws, loc). Then, using $\Delta list$, $d_{ws,loc}$ updates the list of objects stored by $c_{ws,loc}$ in its directory index.

Algorithm 4 Push behavior of $c_{ws,loc}$

```
loop
    counter ← list. count_changes()
    if counter ≥ threshold then
        Δlist ← list. extract_changes()
        pushMsg ← ⟨Δlist⟩
        send pushMsg to d_{ws,loc};
        reset_age(d_{ws,loc})
        counter ← 0
    end if
end loop
```

Algorithm 5 Behavior of $d_{ws,loc}$

```
// active behavior
loop
    wait(T_gossip)
    view. increment_age()
end loop

// passive behavior
loop
    waitPush_Message()
    receive msg from c_{ws,loc}
    reset_age(c_{ws,loc})
    directory-index.update(c_{ws,loc}, push.Δlist)
end loop
```

A directory peer also has to maintain its directory summaries, which are summaries of the directory-indexes of other directory peers. A directory peer pushes a refreshed directory summary to its neighbor directory peers when the percentage of new object identifiers (that are not reflected in the old summary) reaches a predefined threshold. This delayed propagation is warranted as [FCAB98] has shown that directory summaries do not have to be updated every time the related directory index changes. Hence, the use of directory summaries has low demand on bandwidth and memory, while achieving a low probability of false positives.

3.4.2 Query Processing

A content peer processes its own queries as well as other queries coming from its petal. Incoming queries are sent by content peers or the directory peer on behalf of a new client.

Consider $query(o_{ws})$, a query that requests an object of the content of ws noted o_{ws}. Upon receiving $query(o_{ws})$, $c_{ws,loc}$ processes it as shown in Algorithm 6. First, $c_{ws,loc}$

checks its own *content-list*. In case o_{ws} is locally cached, $c_{ws,loc}$ serves the query by directly transferring the object to the query originator. Then, if the query originator is a new client, $c_{ws,loc}$ adds it to its view: the entry is associated to an age equal to zero and a null content-summary. To let the new peer join the petal , $c_{ws,loc}$ sends it a subset of its view so that it initializes its empty view.

In cas the object is not found locally, $c_{ws,loc}$ forwards the query based on its *content-summaries*. However, if $c_{ws,loc}$ has recently joined the petal, it might not have received content-summaries yet. Therefore, it redirects the query to its directory peer. Otherwise, $c_{ws,loc}$ queries the content-summaries to check if any $c'_{ws,loc}$ of its view might have the requested object o_{ws}. Many potential candidates may exist (i.e., many $c'_{ws,loc}$ that seem to have o_{ws}) but some of them may have disconnected. $c_{ws,loc}$ randomly scans through the candidates; it tries to contact them and discards the unavailable ones until finding an available $c'_{ws,loc}$ (i.e., lines 13-21). Then $query(o_{ws})$ is redirected to $c'_{ws,loc}$ which proceeds with **process(***query*(o_{ws})**)**. When no satisfying content peer is found, $query(o_{ws})$ is redirected to the website ws (i.e., line 22).

Algorithm 6 - process(*query*(o_{ws})**)** at $c_{ws,loc}$

```
1: if content-list(c_ws,loc).contain(o_ws) then
2:     serve query(o_ws)
3:     if originator is new then
4:         view.add_Contact(originator, 0, null)
5:         send viewSubset to originator
6:     end if
7:     break algorithm
8: else
9:     if content-summaries is empty then
10:        redirect query(o_ws) to d_ws,loc
11:        break algorithm
12:    end if
13:    loop
14:        c'_ws,loc ← content-summaries.lookup(o_ws)
15:        if c'_ws,loc != null and c'_ws,loc is available then
16:            redirect query(o_ws) to c'_ws,loc
17:            break algorithm
18:        else
19:            view.remove_Contact(c'_ws,loc)
20:        end if
21:    end loop
22:    redirect query(o_ws) to ws
23: end if
```

By serving queries, Flower-CDN enables progressive replication of an object throughout the $petal(ws, loc)$, based on its popularity in the locality loc. Therefore, at

the redirection of queries for o_{ws} by the directory peer $d_{ws,loc}$, the load would tend to be spread rather evenly across the set of content peers $c_{ws,loc}$ holding copies of o_{ws}.

3.5 Discussion of Design Choices

In this section, we argument our design choices, mainly related to the usage of DHT and gossip protocols, and the hybrid architecture.

We have chosen to build D-ring over a DHT to provide an efficient and reliable lookup that guarantees that new clients can find their petals and join Flower-CDN. However, we previously raised concern about DHT limitation in terms of maintenance overhead under churn. As an example, Chord [SMK+01] requires $O(log^2N)$ messages to update the P2P overlay about a single newly joining peer. In a network of 30000 peers, we obtain 220 update messages. This message overhead does not only increases the network load but it also introduces more delay in DHT lookup operations as update messages take some time to get to all concerned peers and repair routing information. D-ring alleviates this problem and brings more robustness. Since only a selective set of participants take part of D-ring, its size remains bounded because one directory peer represents a whole petal. For instance, for a network of 30000, suppose that Flower-CDN supports 100 websites in 6 localities, we obtain in average 50 peers in each petal and a D-ring of $100 * 6 = 600$ directory peers. Thus, a new directory peer only needs 85 messages to update other peers' routing tables.

Another crucial design choice is the usage of gossip protocols for petal management. They are involved in the construction and maintenance of the petal's unstructured overlay since they provide simplicity and robustness. They are also in charge of the dissemination and monitoring of content-summaries because they can perfectly adapt to dynamic changes. Flower-CDN remediates to their overhead in terms of messages and delay by confining them in localities such that gossip exchanges only engage close-by peers.

Our last important design choice is the hybrid architecture that combines DHT, gossip-based overlays, locality- and interest-aware schemes. The maintenance of all these schemes is combined and merged in the same protocols to limit the overhead under churn and dynamicity. This issue is fully addressed in the next chapter.

3.6 Cost Analysis

In this section, we analyze the overhead of our gossip-based approach which is used to spread the changes in content summaries. Furthermore, the analysis aims at guiding the configuration of gossip parameters in order to minimize the overhead.

Let us consider a change in the content summary of a particular content peer, called *author*, as a *rumor* to be propagated via gossip. We analyze a single rumor noted as S and measure the number of messages required to spread S throughout a petal of size P. Notice that a content summary is a compact representation of the content stored by a

peer, and whenever the peer's content is updated due to a new object insertion or deletion, it does not necessarily affect the summary. This is why in our analysis we assume that the updates on summaries are not frequent.

Let $R(x)$ be the number of peers that become aware of S during round x, and $msg(x)$ the number of messages that disseminate S during round x. In the following, we first observe the evolution of $R(x)$ and accordingly $msg(x)$ with the number of rounds x. Then, we compute the number of rounds f required to spread S throughout a petal of size P, i.e., to reach $R(f) = P$ where f represents the final round. Finally, we measure the final number of messages $M(f)$ generated during the f rounds to spread S in the petal.

Following common practice, e.g. [DHA03], in our analysis we do not take into account the peers that join and leave the system during the rumour propagation.

Round 1

The author includes S in its gossip message and sends it to one contact of its view. The number of messages used for spreading S during this round is $msg(1) = 1$. The number of peers that are now aware of S is $R(1) = 2$, i.e., the author and the contacted peer.

Round (x+1)

In round $(x+1)$, $R(x)$ peers are aware of S. Each aware peer p selects the oldest contact from its view and sends to it its own summary together with a randomly selected subset of summaries from its view. The author of the rumor propagates it in all rounds, while other aware peers include S in their gossip message with probability p_S ; $p_S = L_{gossip}/V_{gossip}$ because L_{gossip} is the number of summaries randomly selected among the peer's view summaries, i.e., V_{gossip}.

Some of the peers to which an aware peer sends the rumor may have already received it in a previous round, e.g. from another peer. We should exclude these peers from the ones that become aware in round $(x+1)$. Let $p_{aware}(x)$ be the probability of choosing a contact that is aware by the end of round x, i.e. that became aware during some round previous to round $(x+1)$. Thus, the probability of choosing an unaware contact in round $(x+1)$ is $p_{unaware}(x) = 1 - p_{aware}(x)$. An aware peer p can gossip to one of $(P-2)$ peers, since p cannot gossip to itself nor to the contact it gossiped to in the previous round. Out of $(P-2)$, there are $(R(x)-1)$ peers aware of S, given that $R(x)$ is the total number of aware peers including p. Thus, $p_{aware}(x) = (R(x)-1)/(P-2)$ and $p_{unaware}(x) = 1 - (R(x)-1)/(P-2)$.

From the point of view of the author peer, the probability of choosing an unaware contact in round $(x+1)$ is $p_{aware/author}(x) = (R(x)-2)/(P-2)$. The author does not send the rumor S to its previous contact that is already aware of S. Thus,
$p_{unaware/author}(x) = 1 - (R(x)-2)/(P-2)$.

Based on the above discussion, the number of peers that are aware of S in the $(x+1)$

rounds is:

$$R(x+1) = R(x) + 1 * p_{unaware/author}(x) + (R(x) - 1) * p_S * p_{unaware}(x) \quad (3.1)$$

The expression is explained as follows. The number of aware peers after $(x+1)$ rounds is equal to the number of peers previously aware, i.e., R(x), and the number of peers newly aware contacted by some of the $R(x)$ peers during round $(x+1)$. The contact of the author is a newly aware peer with a probability $p_{unaware/author}(x)$. Only a p_S fraction of the $(R(x)-1)$ other aware peers (i.e., non author peers) forward S to their contacts. Out of the $(R(x)-1) * p_S$ contacted peers, a $p_{unaware}(x)$ fraction are newly aware of S.

The rumor propagation keeps going until a final round f where $R(f) = P$, i.e., until the whole petal becomes aware of S. If we replace round $(x+1)$ by the final round f in Equation 3.1, we obtain:

$$R(f) = R(f\text{-}1) + 1 * p_{unaware/author}(f\text{-}1) + (R(f\text{-}1) - 1) * p_S) * p_{unaware}(f\text{-}1) \quad (3.2)$$

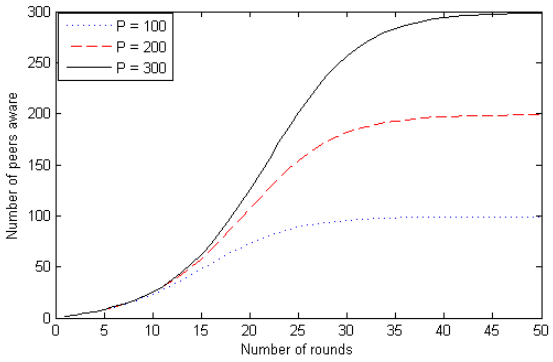
Let us set $p_S = \alpha$ and $1/(P-2) = \beta$ and convert Equation 3.2 to polynomial form. Then, we obtain:

$$R(f) = \alpha\beta R^2(f\text{-}1) + (1 - \beta + \alpha + 2\alpha\beta)R(f\text{-}1) - \alpha\beta - \alpha + 2\beta + 1 \quad (3.3)$$

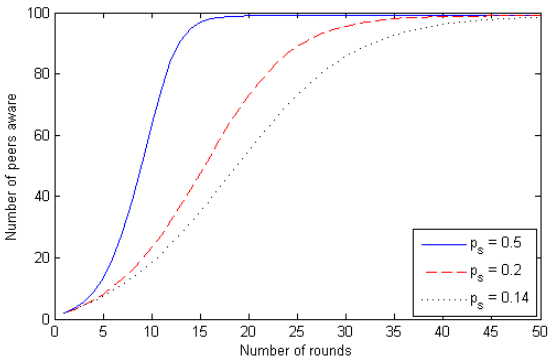
Equation 3.3 can be illustrated by a curve for some given values of p_S and P.
In Figure 3.5a, we set $p_S = 10/50$ and plot three curves, each one for a different P (i.e, $P = 100, 200, 300$). We can see that $R(f) = P$ after 35 rounds for $P = 100$; after 40 rounds for $P = 200$ and after 45 rounds for $P = 300$. This result reflects a common property of gossip protocols: the larger is the size of the petal, the more is the number of rounds needed to propagate a rumor.
In Figure 3.5b, we set $P = 100$ and plot three curves, each one for a different p_S (i.e, $p_S = 10/20, 10/50, 10/70$). We can see that $R(f) = P$ after 20 rounds for $p_S = 10/20$; after 35 rounds for $p_S = 10/50$ and after 45 rounds for $p_S = 10/70$. Indeed, a higher p_S implies that content peers that are aware of a rumor S are more likely to propagate S in every gossip exchange. That is why S is propagated faster througout the petal.

As a result, we can conclude that the intra-petal gossiping has a good convergence speed with respect to the number of rounds. Note that the selection of the gossip period T_{gossip} effectively regulates the speed of gossiping in real time. However, it does not affect the protocol's emergent behavior or its convergence speed.

Let us now compute the number of messages needed for propagating the rumor S. The messages that propagate S in round $(x+1)$ are the gossip messages carrying S in round $(x+1)$ and sent by the peers that are aware of S at the beginning of round $(x+1)$ (i.e., $R(x)$). Thus, the total number of such messages is $msg(x+1) = 1 + (R(x)-1) * p_S$, which reflects one message sent by the author peer and the messages sent by the rest of the aware peers (i.e., $(R(x)-1)$) with probability p_S. After f rounds, the final number



(a) Petal size P vs. number of gossip rounds.



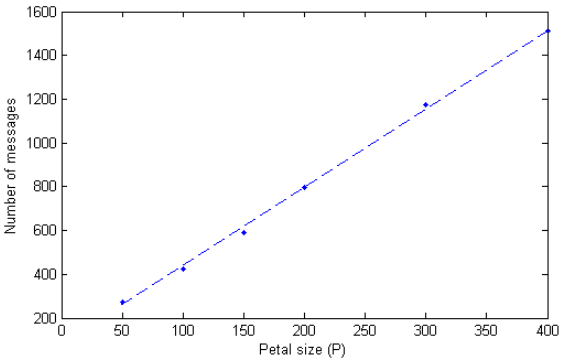
(b) Probability p_S vs. number of gossip rounds.

Figure 3.5: Impact of petal size and probability on the number of rounds required to spread the rumor in the petal.

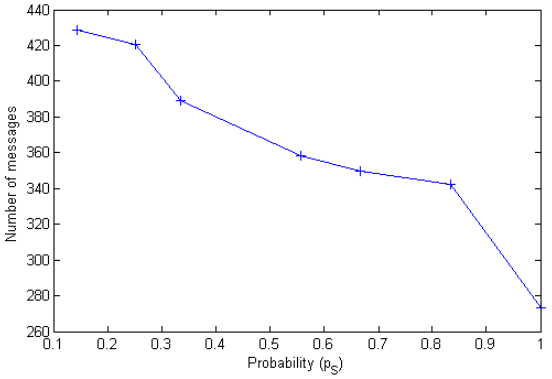
of messages $M(f)$ generated for spreading S into the petal is:

$$M(f) = \sum_{x=1}^{f} msg(x) = \sum_{x=1}^{f} [1 + (R(x) - 1) * p_S] \tag{3.4}$$

In Figures 3.6b and 3.6a, we illustrate the variation of $M(f)$ with p_S and P, respectively. In Figure 3.6a, we set $p_S = 10/50$ and vary P. As shown, the number of messages increases linearly with the increasing petal size, which once again asserts the property of gossip protocols.



(a) Petal size P vs. number of gossip messages.



(b) Probability p_S vs. number of gossip messages.

Figure 3.6: Impact of petal size and probability on the number of messages required to spread the rumor in the petal.

In Figure 3.6b, we set $P = 100$ and vary p_S. Interestingly, when increasing p_S from 0.1 and 1, the number of messages decrease by 35 %. This is because increasing p_S reduces

the number of rounds which has a great impact on reducing the number of redundant messages, i.e., messages sent to peers already aware of R. In fact, with a higher p_S, the rumor tends to be widely propagated from the first rounds during which it is more likely to reach unaware peers. Given that the propagation of R is achieved within fewer rounds, the number of redundant messages is significantly reduced.

Figure 3.7 shows the total number of views exchanged during the f rounds with increasing p_S. We derived this figure from Figure 3.6b, i.e., by multiplying p_S by $M(f)$, because each gossip message contains a fraction p_S of the view. As shown, with increasing the value of p_S, the number of sent views increases.

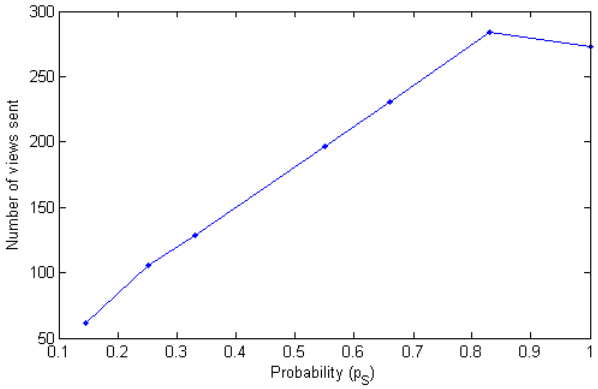


Figure 3.7: Impact of probability p_S on the number of views exchanged to spread the rumor.

Concluding Remarks

The results of our analysis show that our gossip-based approach spreads the rumors with a reasonable communication cost, i.e., less than 4 messages per petal member (see Figure 3.6a). Notice that we have obtained this cost in the worst case, i.e., where there is only one rumor in each message. However, if there are n rumors in each message ($n > 1$), the number of messages per rumor and petal member is less than $4/n$.

The results of our analysis also help us to configure the p_S parameter based on the view size, in order to optimize the communication cost of our gossip-based approach. This configuration is done particularly by studying the behavior of the curves depicted in Figures 3.6b and 3.7. When the view size is small (e.g., $V_{gossip} = 5$ entries), i.e., when the dominant factor for the communication cost is the number of messages, the optimal value for p_S is equal to 1, because it gives the lowest value for the number of messages (see Figure 3.6b). The value of $p_S = 1$ is acheived by setting L_{gossip} equal to V_{gossip} when configuring Flower-CDN. In contrast, when the view size is large (e.g., $V_{gossip} = 50$ entries), i.e., when the dominant factor for the communication cost is the number of sent views, a small p_S gives a better communication cost (see Figure 3.7).

3.7 Performance Evaluation

We evaluate the performance of Flower-CDN through simulations. Our performance evaluation consists mainly in quantifying the gains due to locality-awareness in Flower-CDN. Furthermore, we evaluate the price to be paid for achieving these gains, by examining the trade-off between hit ratio and gossip bandwidth consumption. For these purposes, we use the metrics below:

- **Background traffic**: the average traffic in bits per second (bps) experienced by a content or directory peer due to gossip and push exchanges.
- **Hit ratio**: the fraction of queries satisfied from the P2P system. Hit ratio is an indicator of the degree of server load relief achieved, given that the fraction of queries reflected by the hit ratio are not redirected to the server.
- **Lookup latency**: the average latency taken to resolve a query and reach the destination that will provide the requested object (original server or content peer). Lookup latency is an indicator of the system's search efficency, because it measures how fast objects are found.
- **Transfer distance**: the average network distance, in terms of latency, from the querying peer to the peer that will provide the requested object. Used with queries satisfied from the P2P system, the transfer distance reflects how well the system exploits the locality-awareness in finding close results to clients.

In the following, we first present our evaluation methodlogy and argue the choice of simulation parameters, then we discuss the results.

3.7.1 Evaluation Methodology

We conduct simulation-based experiments using PeerSim [JMJV], a Java-based simulator specifically tailored for P2P protocols. PeerSim provides an event-driven framework that enables us to model the latency of each individual link; however, it does not provide support for simulating bandwidth and CPU resources. Given that P2P networks are built on top of the Internet, we generate an underlying topology of peers connected with links of variable latencies; the model inspired by BRITE [MLMB02] assigns latencies between 10 and 500 ms. Flower-CDN localities are modeled using the landmark-based technique [RHKS02]. We use $k = 6$ localities that are non-uniformly populated.

Given that D-ring relies on a DHT-structured overlay, we choose Chord overlay [SMK+01] for its simplicity; we simulate its routing and churn stabilization protocols and adapt its key management service as explained in Sec.3.3.1, to be able to simulate the D-ring protocol. To construct D-ring overlay, we assume that Flower-CDN supports $|W| = 100$ websites, which results in $k * |W| = 600$ directory peers.

We compare Flower-CDN with the DHT-Directory approach that is widely employed in the P2P CDN litterature (cf. Section 1.5.3.5). Recall that, in DHT-Directory, all

participant peers are part of one structured overlay based on a traditional DHT. For each requested object, a small directory of pointers to recent downloaders of the object. The storing peer, which is comparable to our directory peer, is identified by the hash of the object's identifier without any locality or interest considerations. A query always navigates through the DHT and then receives a pointer to a peer that potentially has the object. We chose the DHT-Directory strategy because it shares some similarities with Flower-CDN wrt. the directory structure. This makes a comparison easier and at the same time allows us to see the effects of locality-based petals and their gossip-based management.

Each experiment is run for 24 hours mapped to simulation time units. In order to keep the load at bay, we restrict the query generation to 6 *active* websites of W. For our query workload we use synthetically generated data because available web traces reflect object accesses while we are interested in website accesses. Each active website provides 500 objects which are requestable and cacheable (e.g., web page of 10-100 KB, though we do not model object size). Our simulation model assumes no correlation between different website communities and applies zipf distribution for object requests submitted to each ative website of W [BCF+99]. The websites involved in our system are small specialized sites: each site speaks directly to the specific needs and interests of its committed community. Hence, they dominate their target niches and get considerable traffic. A peer only poses queries for objects unavailable in its local storage (i.e., it never issues the same query more than once). Moreover, we assume that a content peer has enough storage potential to avoid replacing its stored content through the experiment's duration. As a peer only stores content it has requested, this is a reasonable assumption given the usual browsing activity of individual users.

Experiments start with a stable D-ring: for each couple (website, locality), there is one directory peer with an empty directory. Petals related to the 6 active websites, are built progressively during the simulation as new clients join in. Queries are generated with a rate of 6 queries per second, distributed between the 6 active websites [1]. For each query intended to a given website ws, two selections are carried out: (1) a new client or a content peer of ws is chosen from a random locality as the query originator, and (2) the queried object is selected, using Zipf law, among ws objects. Then, new clients become content peers and join their corresponding petal. When a petal reaches its maximum size noted $petalSize$ (set by default to 100), no new clients may join the petal. With this, we avoid that the directory peer is overloaded with the maintenance of the petal information. In consequence, the petals of a given website evolve at different rythms and sizes. Eventually, we should have up to $N = |W| * k * petalSize$ participant peers. However, since we are only looking at 6 active websites, $N = |W| * k + (6 * k * petalSize)$ which is equal to 4200 participant peers in the current configuration.

The main simulation parameters are summarized in Table 3.1. *Summary size* denotes the size of the Bloom filter representing the content summary; we assume that the maximum number of objects held by a content peer is limited by the total number of

[1] We could not submit larger workloads because of the simulator limitations in terms of memory constraints. However, the chosen workload still gives us a good understanding of the relative behavior.

Table 3.1: Simulation Parameters

Parameter	**Values**
Latency	10-500 ms
Nb of localities k	6
Nb of websites $\|W\|$	100
Nb of participants	2400
Nb of objects/website	500
Query rate	6 queries per second
Summary size	8*500 bits
Push threshold	0.1; 0.5; 0.7
View size V_{gossip}	20; 50; 70
Gossip period T_{gossip}	1 min; 30 min; 1 hour
Gossip length L_{gossip}	5; 10; 20

objects provided by its website, thus we set *summary size* according to the analysis in [FCAB98], to minimize both false positives and storage requirements. *Push threshold* refers to the percentage of new changes beyond which a content peer launches a push exchange with its directory peer (cf. Secction 3.4.1.3). V_{gossip} refers to the view size and T_{gossip} to the gossip period as described in Section 3.4.1 while L_{gossip} refers to the maximum size of the view subset exchanged in a gossip round. More details about the tuning of these gossip parameters are given in the following sections.

3.7.2 Trade off: Impact of gossip

The first experiments evaluates the trade-off of Flower-CDN. Therefore, we investigate the impact of background traffic, on the performance of Flower-CDN, by varying the gossip parameters: *gossip length* (L_{gossip}), *gossip period* (T_{gossip}) and *view size* (V_{gossip}). We also varied *push threshold*; but we do not show the results which illustrate similar performance (i.e., almost same gains and same trade-off) for different values of *push threshold* (0,1; 0,5; 0,7). Thus, these experiments also help in tuning the gossip parameters and adapt them to our protocol.

In each experiment, we vary one of the 3 gossip parameters (L_{gossip}, T_{gossip}, V_{gossip}) and fix the two other parameters; then after 24 simulation hours, we collect the results for each parameter value. Table 3.2 lists the results obtained for the 3 experiments, in terms of hit ratio and background bandwidth. Due to lack of space, we do not show lookup latency and transfer distance results which are quite unaffected by the gossip parameters' variation.

Table 3.2(a) shows the results of the variation of L_{gossip}. When increasing the gossip length, more information is sent at each gossip exchange and thus more background bandwidth is consumed at each involved peer. Indeed, if L_{gossip} increases from 5 to 20, the background bandwidth increases by a factor of 4 as shown in Table 3.2. Yet, the increase in hit ratio is not substantial.

Table 3.2: Impact of Gossip.

L_{gossip}	HIT RATIO	BACKGROUND TRAFFIC
5	0.823	37 bps
10	0.86	74 bps
20	0.89	147 bps

(a) Varying L_{gossip} with (T_{gossip} = 30 min; V_{gossip} = 50)

T_{gossip}	HIT RATIO	BACKGROUND TRAFFIC
1 min	0.94	2239 bps
30 min	0.86	74 bps
1 hour	0.81	37 bps

(b) Varying T_{gossip} with (L_{gossip} = 10; V_{gossip} = 50)

V_{gossip}	HIT RATIO	BACKGROUND TRAFFIC
20	0.78	74 bps
50	0.86	74 bps
70	0.863	74 bps

(c) Varying V_{gossip} with (L_{gossip} = 10; T_{gossip} = 30 min)

Table 3.2(b) shows the results of the variation of T_{gossip}. When increasing the gossip period, gossip exchanges are more spaced and thus less fequent, which has a similar effect on bandwidth consumption as the decrease of gossip length. Background bandwidth is reduced by a factor of 60 by augmenting T_{gossip} from 1 minute to 1 hour, while the hit ratio is decreased by 0.13.

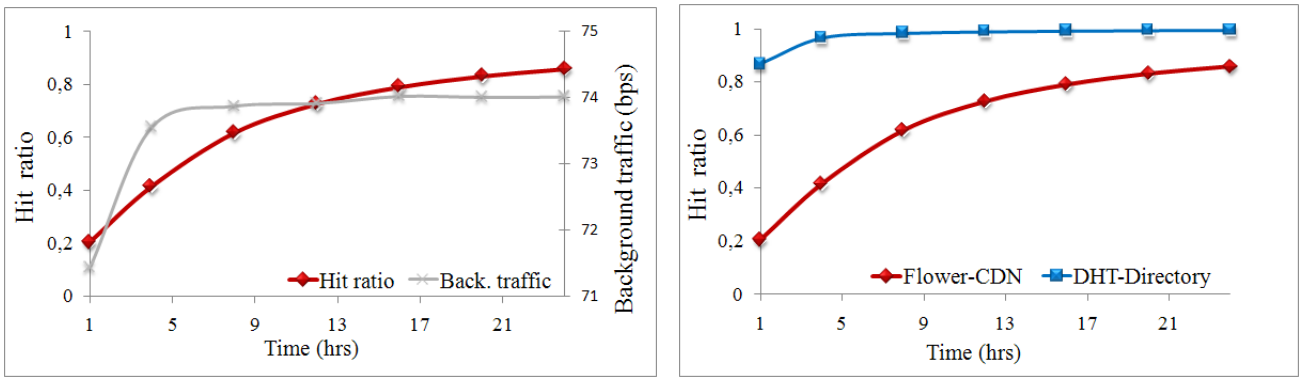
Therefore, the choice of the 2 gossip parameters (L_{gossip} and T_{gossip}) is a trade-off between two factors: (1) the application requirements for hit ratio convergence speed, i.e., how fast Flower-CDN reaches a maximal hit ratio, and (2) the network available resources in terms of network bandwidth availability. For relatively fast convergence, i.e., hit ratio of 0.86 within 24 hours, we could set T_{gossip} = 30 min and L_{gossip} = 10. A peer would experience 74 bps, which is very low bandwidth that could be sustained even by modem connections. For less demanding applications with limited bandwidth availability, we could set (T_{gossip} = 1 hour, L_{gossip} = 10) or (L_{gossip} = 5, T_{gossip} = 30 min) resulting in the negligible amount of 37 bps per peer.

Table 3.2(c) illustrates the results of the variation of V_{gossip}. As shown, increasing the view size does not affect bandwidth consumption, while the hit ratio presents a slight increase of 0.083 when enlarging the view from 20 to 70 contacts. In fact, a larger view size only requires more storage space but does not affect the amount of information exchanged between content peers.

For the rest of the simulation, we set T_{gossip} = 30 min, L_{gossip} = 10 and V_{gossip} = 50, because this setting provides good performance with an acceptable overhead in terms of background traffic (i.e., on average 74 bps per peer). However, we believe that different

query workloads and churn rates may influence the results for T_{gossip} and L_{gossip} which should be tuned accordingly.

To conclude, we show in Figure 3.8a the variation of background traffic and hit ratio with time, for the setting chosen above. The hit ratio keeps on increasing with time, given that copies of queried content are progressively spread into the different petals as more queries are generated and thus more content peers are served. While the hit ratio continues to improve, the background traffic stabilizes at 74 bps after 5 hours.



(a) Trade off with bandwidth in Flower-CDN.

(b) Flower-CDN vs. DHT-Directory.

Figure 3.8: Hit ratio evolution in static environment.

3.7.3 Hit ratio

The following results compare DHT-Directory and Flower-CDN wrt. hit ratio. Figure 3.8b shows that the hit ratio eventually converges to 1 for both DHT-Directory and Flower-CDN, but convergence takes longer for Flower-CDN given that the search space is partitioned into petals. In fact, after 24 hours, the hit ratio of Flower-CDN is less than that of DHT-Directory by 13%. This difference can be justified by the following. Once a copy of an object o_{ws} is stored in DHT-Directory, a subsequent query for o_{ws} searches all the overlay and eventually finds it in case of a stable environment. In comparison, Flower-CDN restricts the search for o_{ws} in the target *content-overlay*(ws, loc_i) wrt. locality of the client (i.e., loc_i) as well as *content-overlay*(ws, loc_j) where d_{ws,loc_j} is a direct neighbor of d_{ws,loc_i} on D-ring (guided by the directory summaries as explained in Sec. 3.3.3), in order to achieve locality-awareness. Moreover, an object o_{ws} becomes available in *content-overlay*(ws, loc) only after a peer from the overlay has submitted a query for o_{ws}. Thus, once a copy of o_{ws} is available in each content-overlay, Flower-CDN achieves a hit ratio similar to DHT-Directory wrt. o_{ws}.

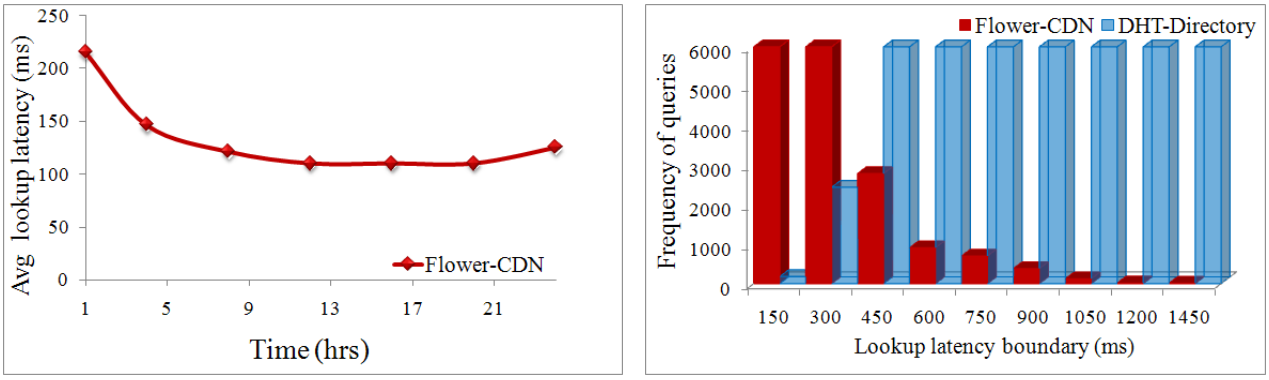
In general, a smaller hit ratio means less queries are served from the P2P and instead go to the server. This is not bad as long as the server is not overloaded. Furthermore, as we will see in the next paragraph, DHT-Directory achieves the better hit ratio by using

peers as content providers that are far away from the requester. In practice, it might be faster to retrieve requested objects from the server than a far away peer.

3.7.4 Locality-awareness

Here, we evaluate the gains due to locality-awareness in Flower-CDN, by measuring lookup latency and transfer distance. Again we compare with DHT-Directory which does not leverage locality-awareness.

The first experiment measures the lookup latency. Figure 3.9a shows the variation of the average lookup latency of a query with time: the lookup latency starts by decreasing and stabilizes around 120 ms shortly after the system warms up (i.e., less than 5 hours in this experiment). Figure 3.9b shows the latency distribution of queries for both solutions: 87% of our queries are resolved within 150 ms while 61 % of DHT-Directory's queries take more than 1050 ms. In Flower-CDN, only first queries of new participants have to go through D-ring and result in long lookup latencies. Afterwards, queries are resolved within the local petal, achieving very short delays. In contrast, DHT-Directory routes every single query through the DHT. Thus, we conclude that the locality-aware hybrid overlay of Flower-CDN performes very well in providing efficient lookup.



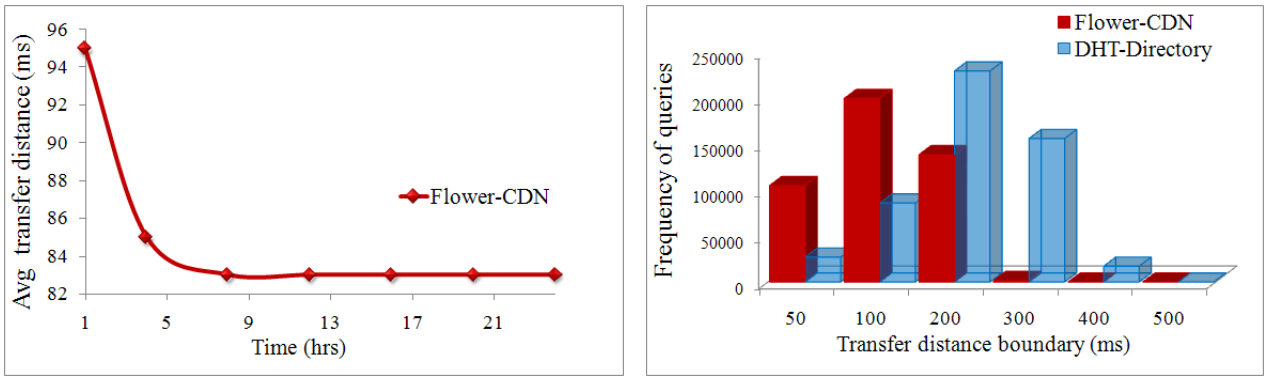
(a) Evolution with time.

(b) Query distribution.

Figure 3.9: Lookup latency in static environment.

The second experiment focuses on transfer distance. We are interested in this metric because it has a significant impact on network usage and object download speed which affects response times perceived by users. At the underlying network level, higher distances generally involve more intermediate links and nodes to carry the traffic, which contributes to the aggregate network utilization and may overlaod the network. Furthermore, additional delays are introduced by the extra stages traversed by the data, due to acknowledgments and retransmissions at each visited node, etc. Figure 3.10a shows the variation of the average transfer distance of a query with time: the transfer distance is high at first when object transfers (i.e., downloads) are done via the original servers. After the warm-up period the transfer distance drops significantly to 80 ms when many

transfers start to be performed within the same locality. Figure 3.10b shows the transfer distance distribution of queries for both solutions: 59 % of our queries are served from a distance within 100 ms compared to 17% of DHT-Directory's queries. Thus, Flower-CDN provides excellent results by reducing the average tranfer distance by a factor of 2 in comparison with DHT-Directory. Flower-CDN ensures data transfers over short distances, which limits the network load and reduces the response times perceived by users.



(a) Evolution with time.

(b) Query distribution.

Figure 3.10: Transfer distance in static environment.

3.7.5 Discussion

We learnt two main lessons through our set of experiments. First, the usage of gossip when confined in petals appears to be quite efficient with an acceptable overhead in terms of bandwidth consumption. Moreover, the bandwidth overhead could be adapted to the available network resources by tuning the gossip parameters, while respecting hit ratio requirements. Second, combining structured and gossip-based overlays with locality-aware considerations proved to be quite performing especially in performing fast searches (i.e., low lookup latency) and finding close-by results (i.e., low transfer distance). In Flower-CDN, D-Ring is only used to provide a first reliable access, for new participant peers wrt. a petal. Afterwards, they become part of this petal and direct subsequent queries directly to the petal instead of D-ring. In contrast, DHT-Directory relies on the DHT-based overlay for every single query leading to high lookup latencies. Furthermore, DHT-Directory's DHT contains all peers while D-ring only contains the subset of directory peers. Thus, D-ring is smaller and therefore, routing is faster than in DHT-Directory. Moreover, although not measured in our experiments, the high lookup rates very likely also lead to higher loads on DHT participants.

3.8 Conclusion

In this chapter, we proposed Flower-CDN, an interest and locality-aware P2P CDN, that enables a website to efficiently distribute its content, with the help of the community interested in its content. Without relying on any dedicated servers, Flower-CDN offers an efficient routing infrastructure for the community's queries. Flower-CDN's infrastructure intelligently combines DHT efficiency for reliable lookup with gossip robustness for self-monitoring. Furthermore, it exploits peer interests and localities in order to organize participant peers and serve queries. The P2P directory service, D-ring, relies on a novel DHT mechanism that can be easily integrated into existing structured overlays, whereas the petals are constructed and managed via cheap gossip protocols. We analytically and empirically analysed our gossip protocols to efficiently tune their parameters and control their overhead. Through simulation-based experiments, Flower-CDN showed high performance especially in performing fast searches and finding close-by results. Furthermore, gossip incured acceptable overhead in terms of bandwidth consumption, which could be adapted to the available network resources and hit ratio requirements. Our results demonstrate that the design choices of our hybrid architecture are perfectly adapted to the context.

The next step would be to focus on the robustness and scalability of our new infrastructure. These major requirements prove whether or not Flower-CDN can compete with commercial CDNs and yet avoid their prohibitive costs.

Chapter 4

High Scalability and Robustness in a P2P CDN

Abstract. A primary concern about a P2P CDN is its ability to handle dynamic and large-scale participation of peers. Flower-CDN should be robust to churn and failures and prevent these frequent events from disrupting the architecture efficiency. Furthermore, Flower-CDN should be able to support massive scales while providing unaltered performance and avoiding bottlenecks. In this chapter, we provide Flower-CDN with high scalability and robustness under churn and massive scale. For this, we propose PetalUp-CDN, a highly scalable version of Flower-CDN, that dynamically adapts to variable rates of participation and prevent overload situations. Moreover, we design maintenance protocols that can efficiently detect and recover from failures and churn via low-cost gossiping.

4.1 Introduction

When designing a CDN over a pure P2P infrastructure, particular challenges arise because the peers are autonomous and volunteer participants. In particular, churn rate is much higher than in dedicated CDN infrastructures, i.e., peers unexpectedly leave or join by thousands the P2P network. Indeed, several analysis [GDS+03, SW04] shed light on this issue and show that the median time between when a node joins the network until it leaves, varies from one hour to a few minutes. To achieve robustness, the system should seek to minimize the performance degradation caused by the inherent dynamicity of peers.

However, it should make sure to keep the maintenance overhead at bay so that it does not offset the gains in robustness. In the P2P literature, the works [LGB03,RY05] that propose a P2P CDN are designed to operate in a dynamic environment. OLP [RY05] greatly relies on a central server which inevitably presents a single point of failure. Kache [LGB03] borrows gossip robustness at the cost of agressively replicating directory information that eventually should be updated under dynamic changes, which yields a significant overhead in terms of storage and update.

Flower-CDN should be well-prepared in face of churn, treat failures as normal occurences and prevent these frequent events from affecting the system reliability. Given that the essence of Flower-CDN (partly) lies in its locality and interest-aware architecture, it is crucial to maintain this architecture and preserve its functionality despite dynamic changes. For this, we have considered every scenario related to peer dynamicity and proposed the appropriate maintenance protocol that ensures a graceful recovery of the system. In particular, the failure of a directory peer is efficiently detected and replaced, via gossip protocols, by content peers of the petal since they share the same interest and belong to the same locality. The directory information is progressively reconstituted via gossiping while relying on summaries during the recovery phase.

Another challenge is to guarantee the scalability of the P2P CDN under massive scales. Scalability involves unaffected performance while avoiding bottlenecks and overload situations. However, most of the existing P2P CDNs do not focus on this issue as pointed out in Section 1.5.3.6. In our case, Flower-CDN aims at keeping the petals at a manageable size, so that their directory peers are not overloaded with the maintenance of the directory information. More precisely, Flower-CDN restricts the number of participant peers that can contribute to the system, by limiting the size of each petal. A petal's size is constrained by the capacities (processing, storage, bandwidth) of the directory peer currently in charge of this petal. However, the P2P system may attract more participant peers than the system's predefined capacity. To address this issue and warrant the extensive deployment of Flower-CDN to larger scale, we have designed *PetalUp-CDN*, an approach that extends Flower-CDN. The key idea is to increase the number of directory peers in a petal as the number of content peers increase. Basically, several directory peers could share the management of a given petal, mainly in indexing the petal's content and servicing new clients. This multi-directory scheme dynamically adapts to variable rates of participation, thus achieving scalability while preserving performance. Furthermore, it improves the reliability of the system since multiple directory peers maintain indexing information related to the same petal.

Our contributions which are partly published in [DPK09b] can be summarized as follows:

- We propose PetalUp-CDN, which dynamically adapts to increasing numbers of participants in order to avoid overload situations in the context of a large-scale application. Additionally, PetalUp-CDN deals efficiently with reverse contexts where peers progressively depopulate the system.

- We describe how to maintain PetalUp-CDN (and Flower-CDN) in face of dynamic

changes and failures, by relying on low-cost gossip protocols and a locality-aware maintenance protocol for our novel D-ring.

- Finally, we present an empirical analysis of scalability and robustness under churn. Our generic solution outperforms an existing P2P-CDN with respect to hit ratio by 40% and reduces response time by a factor of 12 under high levels of dynamicity. Moreover, our approach leverages larger scales to achieve higher improvements.

Roadmap. PetalUp-CDN is described in Section 4.2. Section 4.3 discusses the maintenance protocols that ensure the robustness of Flower-CDN and PetalUp-CDN under churn. Finally, Section 4.4 presents our empirical analysis before the conclusion.

4.2 PetalUp-CDN

PetalUp-CDN is a scalable version of Flower-CDN that dynamically adapts to variable rates of participation. In the following, we first clearly define the problem that PetalUp-CDN addresses. Given that PetalUp-CDN mainly affects D-ring, we then describe the architecture of D-ring and its evolution according to the dynamicity of the P2P network.

4.2.1 Problem Statement

In Flower-CDN, one directory peer $d_{ws,loc}$ is in charge of $petal(ws, loc)$ and is assigned three main tasks. First, it routes the queries of new clients over D-ring. For this it maintains a routing table provided by the underlying DHT of D-ring. Second, it provides an access to the petal for new clients of ws in locality loc and processes their first queries based on its directory information. Second, it indexes the content shared by all the content peers $c_{ws,loc}$ and maintains these indexes under churn and dynamic changes. Accordingly it receives regular push and keepalive messages from each $c_{ws,loc}$ in the petal.

To prevent the directory peer from being overloaded with its tasks, Flower-CDN limits the size of the petal, i.e., the number of clients with respect to a website and a locality that can use and participate to Flower-CDN. For this, the maximum size of a petal can be fixed a priori: it can be a system parameter that is tuned by the engineers according to some predictions like the rate of participation and the average capacity of a participant (capacity in terms of processing, bandwidth and storage). Moreover, whenever a directory peer is overloaded, it can simply retire by leaving D-ring, and then it would be automaticallly replaced (more details are provided by the maintenance protocol of D-ring in Section 4.3.2).

However, accurate a priori prediction is not a straighforward endeavor. Furthermore, and most importantly, the rate of participation with respect to a petal could exceed the average capacity of one potential directory peer. This implies that many clients could be prevented from contributing to the aggregate capacity of a petal in terms of processing, bandwidth and storage.

To resolve the aforementioned problem, one could split a petal into several sub-petals of manageable sizes. However, it severely reduces the search scope of content peers as they would not be able to access the content of their interest that is stored by peers in the same locality but in a different sub-petal.

PetalUp-CDN should be designed in a way that allows several directory peers to share the management of the same petal. To maintain the locality- and interest-aware architecture and its high performance unaffected, additional challenges need to be addressed.

- adapt D-ring architecture in order to support several directory peers per petal.
- implement D-ring evolution in a dynamic way that does not affect the performance of the P2P directory service.
- adapt the petal's management to the changes in order to preserve the efficiency of content search inside a petal.

In the following, we first describe the architectural changes applied to D-ring, then present the dynamic evolution of D-ring, and finally the adapted petal management.

4.2.2 D-ring Architecture in PetalUp-CDN

The current structure of D-ring cannot support more than one directory peer for each couple (ws, loc). Since the problem resides in the key management service of D-ring, PetalUp-CDN adapts this service to scale-up D-ring.

In PetalUp-CDN, directory peers for each couple (ws, loc) consecutively join D-ring. The number of directory peers in charge of each $petal(ws, loc)$ increases progressively as the number of clients for ws in loc increases.

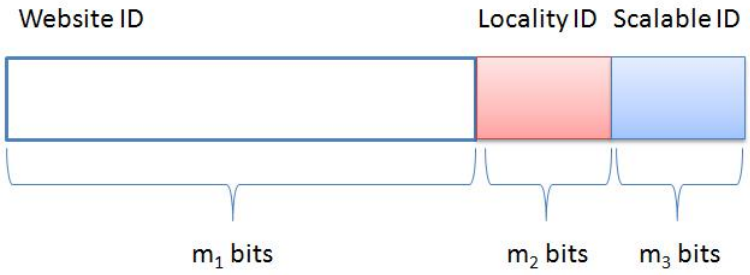


Figure 4.1: Peer ID structure in D-ring of PetalUp-CDN.

Recall that D-ring assigns to $d_{ws,loc}$ a peer ID that concatenates the ID of ws and the ID of loc. PetalUp-CDN introduces another ID of m_3 additional bits where m_3 is a system parameter. This *scalable ID* is suffixed to the peer ID as shown in Figure 4.1.

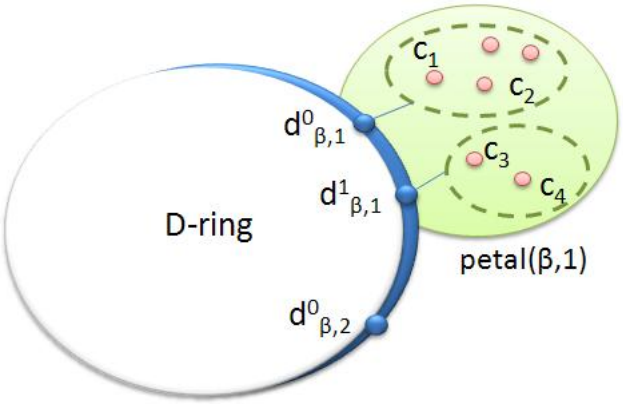


Figure 4.2: Example of $petal(\beta, 1)$ in PetalUp-CDN.

We thereby obtain 2^{m_3} consecutive peer IDs for each couple (ws, loc) instead of only one. Thus, we may have up to 2^{m_3} instances of each $d_{ws,loc}$, noted $d^i_{ws,loc}$ (with $0 \leq i < 2^{m_3}$). As a result, all directory peers for the same website and locality have successive peer IDs and are neighbors on D-ring. This settlement helps directory peers of the same petal efficiently share directory information by exchanging directory-summaries (cf. Section 3.3.2). Furthermore it is vital for the gradual construction of D-ring

Each directory peer $d^i_{ws,loc}$ manages a partial view noted $view(ws, loc)^i$ and thereby a partial $directory\text{-}index(ws, loc)^i$ of $petal(ws, loc)$. The view of a directory peer refers to its diretcory-index, thus both terms can be used interchangeably. More formally, we can state that for each website ws and locality loc, we have two properties:

Property 1 $\forall i, j \ / \ i \neq j : \quad view(ws, loc)^i \cap view(ws, loc)^j = \emptyset$

Property 2 $petal(ws, loc) = \bigcup_{0 \leq i < 2^{m_3}} view(ws, loc)^i$

By having multiple directory peers in charge of a petal, the failure of one or more of these directory peers will not lead to a complete loss of directory information, and will allow the system to continue in a slightly-reduced capacity. Moreover, these additional directory peers are not carrying redundant information, but each one is responsible for maintaining information about a part of the petal. An example of PetalUp-CDN configuration is illustrated in Figure 4.2 which focuses on $petal(\beta, 1)$. Two directory peers $d^0_{\beta,1}$ and $d^1_{\beta,1}$ share the management of $petal(\beta, 1)$. Thus, they manage each one a subset of the content peers $c_{\beta,1}$.

4.2.3 D-ring Evolution in PetalUp-CDN

The petals expand progressively as new peers join and shrink as existing ones leave. To keep the load on directory peers at bay, D-ring follows the evolution of the petals and accordingly may expand or shrink. However, the expansion and shrink should not disrupt the architecture of D-ring nor its performance in routing queries. In the following, we discuss how to address this issue.

4.2.3.1 D-ring Expansion

Directory peers of $petal(ws, loc)$ are created sequentially, starting from $d^0_{ws,loc}$. A new directory peer is created for $petal(ws, loc)$ when the number of content peers $c_{ws,loc}$ can no more be managed by the existing directory peers $d^i_{ws,loc}$. This is detected by directory peers upon processing new queries by evaluating the number of their content peers against a predefined limit.

Recall that queries routed over D-ring are initiated by new clients that eventually join the petals. Thus, in PetalUp-CDN, a query targeting $petal(ws, loc)$ scans through the existing directory peers $d^i_{ws,loc}$ in search for an underloaded directory peer that can resolve the query and take in charge the client as a new content peer. If no such directory peer is found, the latest created $d^i_{ws,loc}$ initiates the join of a new directory peer $d^{i+1}_{ws,loc}$. In the following, we describe how a query is routed over the evolving D-ring and then how it is processed in such a way that might result in the creation of a new diectory peer for the petal targeted by the query.

Query Routing. While scanning the directory peers of its target petal, a query may undergo several redirections before being actually served. Thus, in order to limit query response time, we should minimize the number of query redirections required to reach an underloaded directory peer. Moreover, if contacted by every new client of its petal, a directory peer can become overloaded even if its is just redirecting queries to other directory peers. Thus, as directory peers share the management of directory information, they should also share the handling of new queries. Therefore, we believe that to achieve the optimal routing, each client should discover the number of directory peers that have been created so far for its petal and randomly choose one of them to contact it. When no such global discovery scheme is available, we use a safe alternative that is described below.

When routing a query over D-ring, the client uses a key in which the website and locality IDs are set according to the her information (cf. Section 3.3.1). To determine the value of the scalable ID in the routed key, we propose to pick a random value between 0 and its middle value. For instance, if the scalable ID is formed of 2^3 bits, the scalable ID takes a value between 0 and 4. Consider a query with $ID^4_{ws,loc}$. If $d^4_{ws,loc}$ does not exist, the DHT routing protocol delivers the query to the first preceding directory peer (i.e., $d^i_{ws,loc}$ with $0 \leq i < 4$) because the latter has the closest ID to $ID^4_{ws,loc}$. In such a case, the query would have reached the latest created directory peer which can locally process the query or create a new directory peer for $petal(ws, loc)$ if overloaded. If $d^4_{ws,loc}$ does

exist, the query gets to $d^4_{ws,loc}$ which keeps on redirecting the query to further directory peers of $petal(ws, loc)$ until an underloaded directory peer is found or created. This redirection approach shortens the route of the query and distributes load rather evenly accross directory peers.

Query Processing. Whenever the query reaches a directory peer $d^i_{ws,loc}$ of the target petal, it is handled based on Algorithm 7, i.e., **scalable-process(**$query(o_{ws})$**)**. First, d_{ws,loc_i} checks its view size against a limit, $maxDirectory$. $maxDirectory$ is determined a priori according to the average expected peer capacity in terms of bandwidth, processing and storage. If the view size has reached $maxDirectory$, $d^i_{ws,loc}$ verifies if $d^{i+1}_{ws,loc}$ is in D-ring. In case $d^{i+1}_{ws,loc}$ exists (i.e. lines 2-4), $d^i_{ws,loc}$ redirects the query to $d^{i+1}_{ws,loc}$ which in its turn runs **scalable-process(**$query(o_{ws})$**)**. As for $d^i_{ws,loc}$, its task stops here with **break**. In case $d^{i+1}_{ws,loc}$ does not exist (i.e. lines 5-13), $d^i_{ws,loc}$ selects from its view the content peer to join D-ring as $d^{i+1}_{ws,loc}$. The content peer is then removed from the view and directory-index of $d^i_{ws,loc}$ because it will no loger behave as a content peer. The new $d^{i+1}_{ws,loc}$ keeps its cached content until it expires; meanwhile, it might use this content to satisfy relevant queries received from new clients.

Afterwards, in order to avoid waiting for $d^{i+1}_{ws,loc}$ to join, $d^i_{ws,loc}$ processes the query (i.e., line 13), in its stead, based on **process(**$query(o_{ws})$**)** of Algorithm 2 (Chapter 3). Consequently, $d^i_{ws,loc}$ adds the client to its directory-index as a provider of o_{ws} and to its view as a content peer $c_{ws,loc}$. If the view size has not reached $maxDirectory$ yet, $d^i_{ws,loc}$ performs the same steps to resolve the query and add the new client (i.e., line 13). In

Algorithm 7 - scalable-process($query(o_{ws})$**)** at $d^i_{ws,loc}$

```
1: if view.size ≥ maxDirectory then
2:     if d^{i+1}_{ws,loc} exists then
3:         redirect query(o_ws) to d^{i+1}_{ws,loc}
4:         break
5:     else
6:         c_{ws,loc} ← view.select_Neighbor()
7:         ask c_{ws,loc} to join
8:         d^{i+1}_{ws,loc} ← c_{ws,loc}
9:         directory-index.remove(c_{ws,loc}, −)
10:        view.remove(c_{ws,loc})
11:    end if
12: end if
13: process(query(o_ws))
```

consequence of the above, a new client is only added to the view and directory-index of one specific directory peer, which achieves Properties 1 and 2: each directory peer of ws in loc only adds to its directory-index and its view a partial subset of the clients wrt. (ws, loc).

4.2.3.2 D-ring Shrink

A petal's size evolve dynamically, sometimes decreasing as more content peers leave and sometimes increasing as new clients join. This may result in some cases where an overloaded directory peer gets rid of its failed/departed content peers and starts serving new clients since its view size is reduced. Furthermore, a website may loose its popularity with time, having content peers continuously leaving its petals. In such a case, we need to remove the redundant directory peers and eventually end up with one directory peer to manage the small petal. However, we cannot discard directory peers randomly as it has severe implications on the routing and processing of queries.

To handle this issue, we propose a solution that can be illustrated by a simple example. Assume ws was once very popular in loc, which resulted in creating 3 directory peers $d^0_{ws,loc}$, $d^1_{ws,loc}$ and $d^2_{ws,loc}$. Then, $petal(ws, loc)$ starts to shrink by loosing content peers $c_{ws,loc}$, In such a case, the 3 directory peers merge their subsets of content peers; $d^1_{ws,loc}$ and $d^2_{ws,loc}$ withdraw from D-ring, leaving only one directory peer to manage $petal(ws, loc)$.

More precisely, as a petal starts to shrink, its extra directory peers start to resign from their directory peer positions and become again content peers. This progressive resignation involves the latest created directory peers (noted $d^l_{ws,loc}$) to avoid breaking the sequence of $d^i_{ws,loc}$ and disrupting the mechanisms of PetalUp-CDN (see Section 4.2.3). $d^l_{ws,loc}$ can discover that it is the last directory peer of the sequence by checking that its successor on D-ring belongs to a different petal.

To clearly show how a directory peer decides to resign, let us consider Algorithms 8 and 9. Algorithm 8 describes the case where $d^l_{ws,loc}$ has lost a great majority of its content peers, i.e, its view has reached a predefined minimum noted $minDirectory$. $d^l_{ws,loc}$ sends a $requestMerge$ to its preceding neighbor $d^{l-1}_{ws,loc}$ which accepts to merge its view with $view(d^l_{ws,loc})$ only if the resulting view has an acceptable size. In such a case, $d^l_{ws,loc}$ resigns and $d^{l-1}_{ws,loc}$ takes over.

Algorithm 8 - shrink at $d^l_{ws,loc}$

```
1: if view.size ≤ minDirectory then
2:     send requestMerge(view.size) to d^{l-1}_{ws,loc}
3:     receive answerMerge from d^{l-1}_{ws,loc}
4:     if answerMerge==yesMerge then
5:         resign()
6:         d^{l-1}_{ws,loc}.takeOver()
7:     end if
8: end if
```

Algorithms 9 describes the case where $d^i_{ws,loc}$ (i.e, not the latest created directory peer) has lost a great majority of its content peers. $d^i_{ws,loc}$ sends a $requestMerge$ to $d^l_{ws,loc}$ which accepts only if the merged view has an acceptable size. In such a case, $d^l_{ws,loc}$ resigns and $d^i_{ws,loc}$ takes over.

Algorithm 9 - shrink at $d^i_{ws,loc}$

```
1: if view.size ≤ minDirectory then
2:     send requestMerge to d^l_{ws,loc}
3:     receive answerMerge from d^l_{ws,loc}
4:     if answerMerge==yesMerge then
5:         d^l_{ws,loc}.resign()
6:         takeOver()
7:     end if
8: end if
```

Next, we detail the algorithms of **resign()** and **takeOver()**. In Algorithm 10, $d^l_{ws,loc}$ is resigning to let some other existing $d^i_{ws,loc}$ take over by merging their directory information. Since $d^l_{ws,loc}$ will become again a content peer, $d^l_{ws,loc}$ adds a new entry related to itself in its directory-index: the entry contains the address of $d^l_{ws,loc}$, the list of ws' content stored by $d^l_{ws,loc}$ and the age zero. Then, $d^l_{ws,loc}$ transfers its directory-index to $d^i_{ws,loc}$. $d^i_{ws,loc}$ takes over only if the merged view or directory-index does not exceed $maxDirectory$ (cf. Algorithm 7 in Section 4.2.3). As depicted in Algorithm 11, it basically consists of $d^i_{ws,loc}$ receiving *directory-index*($d^l_{ws,loc}$) and merging it with its own directory-index.

Algorithm 10 $d^l_{ws,loc}$.**resign()** for $d^i_{ws,loc}$; $0 \leq i \leq l-1$

directory-index.**add**($d^l_{ws,loc}$, $content\_list$, 0)
transfer *directory-index* to $d^i_{ws,loc}$

Algorithm 11 $d^i_{ws,loc}$.**takeOver**()

receive *directory-index*($d^l_{ws,loc}$)
directory-index.**merge**(*directory-index*($d^l_{ws,loc}$))

In the worst case, the petal ends up with one directory peer, which is guaranteed as long as there are content peers in the petal. These guarantees are provided by the maintenance protocols that are introduced in Section 4.3.2.

4.2.4 Petal Management in PetalUp-CDN

To maintain efficient content search, a petal should not be affected by the multi-directory scheme. Recall that once a client becomes a content peer, it does not use D-ring anymore and relies on its petal to route its queries and search for its desirable content. Moreover, as a petal scales up, its aggregate resources increase. As such, there will be more content of ws available in $petal(ws, loc)$ as the number of $c_{ws,loc}$ increases. Therefore, each $c_{ws,loc}$ should be able to leverage the scale-up of its petal independently of the number of directory peers.

To enable content sharing throughout $petal(ws, loc)$, $c_{ws,loc}$ gossips to any other $c_{ws,loc}$ of its petal. Thus, in Figure 4.2, c_1 can gossip to both c_2 and c_3 and eventually benefit from their stored content to satisfy its queries. But how does c_1 get to know content peers like c_3 that are controlled by other directory peers? In Flower-CDN, a newly joining ($c_{ws,loc}$) initializes its view based on the view of an older content peer of $petal(ws, loc)$ or its own directory peer $d_{ws,loc}$. In PetalUp-CDN, one should provide the first content peers of $d^i_{ws,loc}$ with content peers related to other directory peers of $petal(ws, loc)$. To illustrate the purpose behind this approach, let us consider Figure 4.2. Suppose that c_3 is the first content peer to join via $d^1_{\beta,1}$ and gets an initial view containing c_1 and c_2. Afterwards, c_4 joins and gets a view containing c_3 which can then transmit the two contacts c_1 and c_2 to c_4 via gossip exchanges. This solution is vey simple and practical and can be implemented as follows.

A new $d^{i+1}_{ws,loc}$ uses its view and content summaries maintained while still a content peer of $d^i_{ws,loc}$, until its old view expires (more details in Section 4.3.1) and gets progressively replaced by a new view related to newly arrived clients. When receiving first clients, $d^{i+1}_{ws,loc}$ provides them with a subset of its old view so that they initialize their view of $petal(ws, loc)$. Thereby, these clients that will become content peers get to know content peers of $d^i_{ws,loc}$ and eventually introduce them to other content peers of $d^{i+1}_{ws,loc}$ via gossip.

4.3 Robustness Under Churn

Dealing with the highly dynamic nature of peers is crucial to ensure the robustness of the P2P CDN. In this section, we first focus on the protocols that maintain D-ring and its petals connected despite churn. Then, we discuss the maintenance protocols of D-ring that aims at preserving the architecture originality. As we explain next, these maintenance protocols cover both approaches of Flower-CDN and PetalUp-CDN. In case of Flower-CDN, the notation $d^i_{ws,loc}$ refers to the single directory peer $d_{ws,loc}$.

4.3.1 Maintenance of Connection between D-ring and Petals

Flower-CDN mechanisms are achieved via the connection between D-ring and the petals. However, the failure or departure of a diretcory peer may disconnect (at least partly) its petal from D-ring. Thus, a primary concern is to maintain this connection despite the highly dynamic environment governed by churn.

In Flower-CDN, the maintenance protocol aims at keeping the one directory peer connected with all the content peers of the petal. In PetalUp-CDN, given that several directory peers may coexist within the same petal, one should maintain the connection of each $d^i_{ws,loc}$ to a subset of content peers from its $petal(ws, loc)$, which corresponds to its $view(ws, loc)^i$. To achieve this, each content peer of $petal(ws, loc)$ restricts its communications to the directory peer $d^i_{ws,loc}$ via which it joined the petal.

The maintenance protocol relies on two features: *push & keepalive messages* and exchange of *dir-info*.

Exchange of *dir-info*. Each $c_{ws,loc}$ keeps track of its directory peer $d^i_{ws,loc}$: it maintains a *dir-info* which contains the address and peer ID of $d^i_{ws,loc}$ as well as the *age* field. $c_{ws,loc}$ periodically increments its *dir-info* by 1 and resets it to zero whenever contacting $d^i_{ws,loc}$. Recall that two content peers that gossip to each other also exchange their *dir-info* to discover the current available directory peer. If the exchanged *dir-info* share the same peer ID, then the 2 content peers belong to the same directory peer. In such a case, they both keep the *dir-info* with the smaller age, which refers to more recent information about their directory peer. Thus, whenever a directory peer leaves, some of its content peers that detect it when trying to contact it, gossip the information to the other content peers concerned with this particular directory peer so that they update their *dir-info*.

Push & Keepalive Messages. As discussed in Section 3.4.1.3, the directory peer and the content peers of a petal monitor the liveness of each other mainly via push messages. However, this is not enough because some content peers do not produce frequent changes in their stored content and therefore rarely communicate with their directory peer via push messages. That is why we exploit a feature inherent to P2P systems, *keepalive messages*, which are periodically sent to check links between peers. In consequence, there will be two forms of interaction between a directory peer and its content peers: *push messages* and *keepalive messages*. More precisely, $c_{ws,loc}$ regularly sends *keepalive* messages to $d^i_{ws,loc}$ in addition to push messages. In case of the example shown in Figure 4.2, c_1 which is linked to $d^0_{\beta,1}$ only sends push and keepalive messages to $d^0_{\beta,1}$. At the same time, $d^i_{ws,loc}$ periodically increments the age of its view entries and discards the expired ones as they probably refer to dead content peers. Upon the reception of a push or keepalive message from $c_{ws,loc}$, $d^i_{ws,loc}$ resets to zero the age of $c_{ws,loc}$'s entry in its *directory-index*(ws, loc).

4.3.2 Maintenance of D-ring

Churn has severe implications on D-ring architecture and operation in the absence of appropriate maintenance protocols. If a directory peer fails or leaves, its queries will be redirected to unconcerned directory peers and the clients will not be able to join their target petal. Thus, D-ring should be able to detect and recover from failures and leaves. Furthermore, to support the gradual construction, D-ring should enable directory peers to dynamically join D-ring without disrupting the architecture. In the following, we first discuss the failures and leaves, then the joins and replacements of directory peers. The protocols that handle such events are not affected by whether one or several directory peers exist for the same petal (i.e., Flower-CDN or PetalUp-CDN). More details are given below.

4.3.2.1 Failures and Leaves

A directory peer leaves D-ring when it fails or quits the system. The leave of $d^i_{ws,loc}$ is detected by its content peers, i.e., contained in its $view(ws, loc)^i$, while sending keepalive or push messages. The replacement of $d^i_{ws,loc}$ is performed by a peer that shares the

interest in the same website's content and belong to the same locality, i.e., a content peer from $view(ws, loc)^i$ or a new client. If $d^i_{ws,loc}$ leaves voluntarily, it selects from its view the content peer to replace it. Otherwise, any content peer of $view(ws, loc)^i$ can perform the replacement as soon as it detects the failure.

However, in case of a deliberate resignation of a directory peer $d^i_{ws,loc}$ due to the petal's shrink, the content peers should not confuse it with a failure and replace their resigned directory peer. Since the latter is instantly replaced by its preceding directory peer $d^{i-1}_{ws,loc}$, $d^{i-1}_{ws,loc}$ is the first peer that knows about the resignation. Moreover, any join message targeting the position $d^i_{ws,loc}$ reaches the replacing directory peer $d^{i-1}_{ws,loc}$ which is the numerically closest to $d^i_{ws,loc}$ on D-ring. In such cases, $d^{i-1}_{ws,loc}$ notifies the content peers that are trying to join and replace $d^i_{ws,loc}$ about the resignation. It also informs them that they are now affiliated to $d^{i-1}_{ws,loc}$.

The detection and replacement involve one directory peer and its content peers. Thus these protocols operate similarly on Flower-CDN and PetalUp-CDN.

4.3.2.2 Joins and Replacements

A peer p can try to join D-ring as a directory peer either in case it is initially (1) a content peer or (2) a new client. Case (1) occurs when p is replacing its failed directory peer or when it joins as as $d^{i+1}_{ws,loc}$ due to its petal's growth. Case (2) happens if p has found no directory peer available for ws in loc while routing its query over D-ring, because (i) p is the first/only participant for $petal(ws, loc)$; or (ii) all the previous directory peers of $petal(ws, loc)$ have left D-ring and have not been replaced yet. In all cases, p uses joinDring($ID^i_{ws,loc}$) (Algorithm 12) where $ID^i_{ws,loc}$ is the ID of the directory peer position targeted by p ($i = 0$ in case (2)). However, p does not always succeed in joining because several peers may simultaneously target the same vacant position; the one that first integrates into D-ring, succeeds.

Algorithm 12 - joinDring($ID^i_{ws,loc}$)

```
1: route joinMessage(ID^i_{ws,loc}) over D-ring
2: directoryPeer ← joinMessage(ID^i_{ws,loc}).destination
3: if directoryPeer.ID == ID^i_{ws,loc} then
   // joinMessage reached a directory peer with the same target ID
4:     dir-info.update(directoryPeer)
5:     if new client then
6:         join petal(ws, loc) as c_{ws,loc}
7:     end if
8: else
9:     become d^i_{ws,loc}
10:    construct directory-index
11: end if
```

Similarly to the standard join in DHT-based overlays, p routes a join message with a

key equal to $ID^i_{ws,loc}$ and eventually reaches a directory peer from the overlay referred to by destination (i.e, line 1-2). If the target position is not vacant (i.e., lines 3-8), the join message reaches the current $d^i_{ws,loc}$ and p discovers its current directory peer to update its *dir-info*. Then, if p is a new client, it simply joins $petal(ws, loc)$ as a content peer. If the target position is vacant (i.e., lines 9-12), p becomes $d^i_{ws,loc}$ and gradually contructs its view and directory-index as its content peers discover its join and send it push messages. As introduced in Section 4.3.1, content peers discover the join of p as they try to contact their previous directory peer $d^i_{ws,loc}$ and detect its leave. Then, some of them will try to join, detect that there is already a new directory peer and update their *dir-info*. Subsequently, the information about the new $d^i_{ws,loc}$ spreads rapidly via gossip to content peers related to $d^i_{ws,loc}$.

If the previous $d^i_{ws,loc}$ had voluntarily left, it would have transferred a copy of its view and directory-index to the new directory peer p before its departure. Moreover, in case p was a content peer before joining D-ring, p would hold content summaries and use them to answer its first received queries, while waiting for its new directory-index to be built.

Subsequent to joins and leaves of directory peers, routing tables should be updated to ensure a correct lookup. For this, we rely on the underlying DHT protocols that can normally detect the presence or the absence of a directory peer and propagate the changes.

4.4 Performance Evaluation

In this section, we present a performance evaluation of PetalUp-CDN and our maintenance protocols. We focus on robustness and scalability provided provided by our proposed protocols. In other words, we aim at quantifying the performance in serving queries under dynamic and large-scale participation of peers.

To evaluate the protocol's robustness and scalability, we use the metrics that were previously used and dicussed in Section 3.7: **background traffic**, **hit ratio**, **lookup latency**, and **transfer distance**. In the remainder of this section, we first describe our evaluation methodology, then we discuss the experimental results. The first set of experiments focuses on robustness and thus involves the maintenance protocols and operate on Flower-CDN as a beginning. Then the second set of experiments aims at scalability and introduces PetalUp-CDN while running the maintenance protocols to manage churn.

4.4.1 Evaluation Methodology

We conduct a set of simulation-based experiments in a highly dynamic environment governed by churn. Our evaluation methodology reuses many concepts defined in Section 3.7. In short, our simulation relies on PeerSim [JMJV]. We generate an underlying topology of peers connected with links of variable latencies between 10 and 500 ms. Then, we model $k = 6$ localities, choose Chord [SMK+01] as our DHT-based overlay

Table 4.1: Simulation Parameters.

Parameter	**Values**
Latency	10-500 ms
Nb of localities k	6
Nb of websites $\|W\|$	100
Population P	3000-15000
Underlying network	$P * 1.3$
Mean uptime m	60 min
Nb of objects/website	500
Query rate	1 query/6 min/peer
Summary size	8*500 bits
Push threshold	0.5
V_{gossip}	-
T_{gossip}	1 h
L_{gossip}	-
$maxDirectory$	15; 25; 35

and compare our protocols with DHT-Directory approaches. We set $W = 100$ websites involved in the P2P CDN but restrict query generation to 6 *active* websites of W. Each active website provides 500 objects whose popularity follows Zipf distribution [BCF+99]. Table 4.1 lists the main parameters.

For a realistic simulation environment, we simulate churn based on a study [SR05] where P2P population converges to a desired size, P. For this purpose, the arrival rate of peers must be equal to the mean departure rate, $\frac{P}{m}$, where m denotes the mean uptime of a peer. We model the uptime of a peer as an exponential distribution with $m = 60$ minutes, resulting in a high churn rate. We assume that a peer always fails (i.e., when its lifetime expires) and never leaves normally, to test our P2P CDN in highly unstable scenarios. Moreover, a peer might re-join multiple times during an experiment, each time with a different uptime.

We conduct experiments targeting different population sizes (i.e., $P = 3000, 5000, 9000, 11000, 15000$) in the context of a highly dynamic environment. The underlying network which consists of all peers (online and offline) have a size of $1.3 * P$.

Initially, each peer is randomly assigned a website from $|W|$ to which it has interest throughout the experiment. We start with a population of $k * |W| = 600$ directory peers which have limited uptimes and form the initial D-ring (i.e., one directory peer per couple (website, locality)). After a small warm-up period, the population stabilizes around P as new clients keep on arriving and existing peers on failing. For all *non-active* websites, peers are only involved with churn because it affects D-ring routing. More precisely, a peer with interest for an active website submits queries on a regular basis, as soon as it arrives until it fails. A peer belonging to a non-active website, is simply added to its petal upon its arrival; it is only involved in the failure management of its directory peer.

We do not limit the *view size* of a content peer and allow it to grow with the size of its petal which reaches at most 60 with $P = 15000$ in the current configuration; also, when a peer selects a contact for gossip and finds it unavailable, the peer removes the contact from its view, which naturally bounds the view size. Finally, *gossip/keepalive period* which refers to the periodicity of gossip and keepalive messages sent by a content peer is calibrated at 1 hour.

4.4.2 Robustness to churn

Here, we focus on the robustness of our protocols under high churn. Thus, we conduct for both DHT-Directory and Flower-CDN the same experiment under the same churn and workload conditions. The experiment targets a mean population size of 3000. The obtained results are depicted in Figures 4.3, 4.4a and 4.4b.

First, we analyse the evolution of hit ratio with time (Figure 4.3). At the beginning, DHT-Directory surpasses Flower-CDN wrt. hit ratio. This is because Flower-CDN needs a warm up period to build up and enable its petals to get populated, given that query search space involves specific petals to achieve locality-awareness. In contrast, DHT-Directory searches the whole overlay for queries and its hit ratio increases faster than that of Flower-CDN. However, as the impact of churn becomes more significant, DHT-Directory fails to preserve an increasing hit ratio while Flower-CDN keeps on improving despite failures: the improvement reaches 40% after 24 simulation hours. In fact, in DHT-Directory, the information about previous downloaders which is held in a directory, is abruptly lost with the failure of the directory peer in charge of it. In contrast, Flower-CDN efficiently manages this problem because periodic updates are disseminated throughout a petal via gossip and push. Thus, a new directory peer d can progressively reconstruct its directory-index as it receives updates from content peers. Meanwhile, d can resolve first queries using content summaries previously received during gossip exchanges, given that a failed directory is replaced by a content peer.

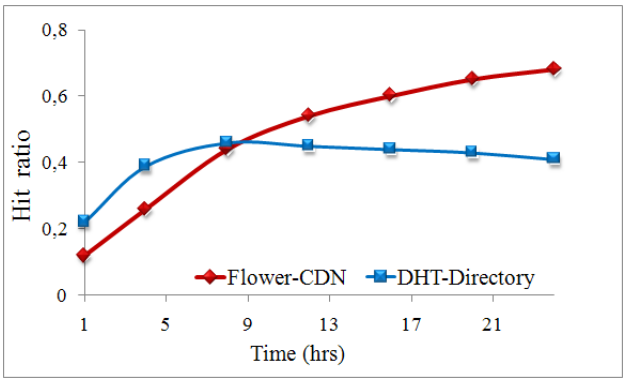
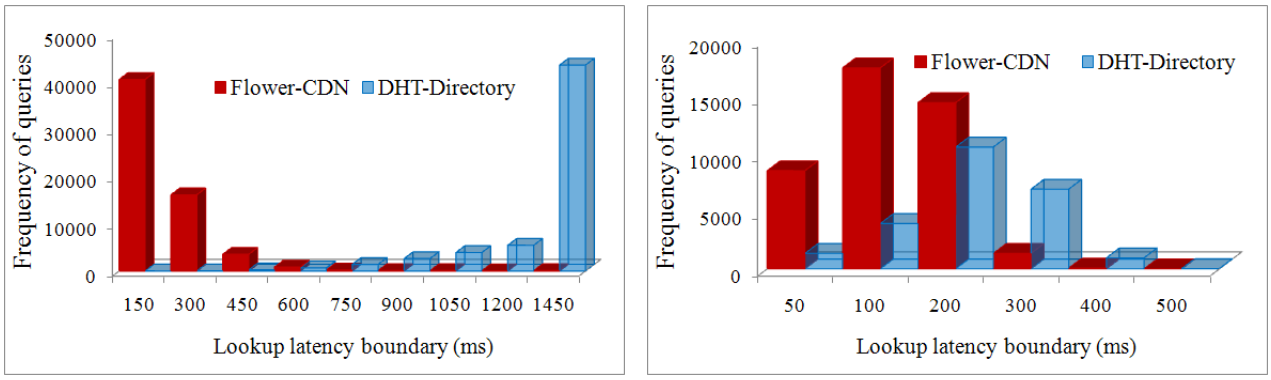


Figure 4.3: Hit ratio evolution in dynamic environment.

Second, we look at the distribution of queries with respect to lookup latency and transfer distance for $P = 3000$. Figure 4.4a shows that 66% of our queries are resolved

within 150 ms while 75% of DHT-Directory's queries take more than 1200 ms. Figure 4.4b shows that the percentage of queries served from a distance within 100 ms is 62% for Flower-CDN and 22% for DHT-Directory. Thus, Flower-CDN preserves its highly significant locality-aware gains under the worst scenarios of failures, given that the directories lost with DHT-Directory can be quickly recovered with Flower-CDN.



(a) Lookup latency.

(b) Transfer distance.

Figure 4.4: Query distribution in dynamic environment.

4.4.3 Scalability

In the following set of experiments, we analyse the scalability of our protocols. First, we examine Flower-CDN under variable rates of participation then we validate PetalUp-CDN. Note that the experiments still simulate high churn.

4.4.3.1 Flower-CDN

We study the behavior of Flower-CDN with respect to scalability and compare it to the behavior of DHT-Directory in a similar scenario. For each approach (Flower-CDN and DHT-Directory), we conduct 5 experiments, each one targeting a different population size (i.e., $P = 3000, 5000, 7000, 9000, 11000$) in the context of a highly dynamic environment. For each experiment, we collect the hit ratio obtained after 24 simulation hours, and the average lookup latency and transfer distance for a query. To avoid over-fitted results, we run each experiment 3 times and compute the average hit ratio, lookup latency and transfer time for this experiment. We also measure for Flower-CDN the average background traffic. The results of the 4 experiments are summarized in Table 4.2.

We can see that the hit ratio of Flower-CDN increases from 0.7 to 0.82 when increasing P from 3000 to 11000. This means that Flower-CDN leverages larger scales to achieve higher gains. Actually, a larger population size enables Flower-CDN to build up and converge to a maximum hit ratio faster. Moreover, the results of hit ratio show that Flower-CDN maintains its improvement over DHT-Directory through variable population sizes.

When comparing the results of lookup latency and tranfer distance between Flower-CDN and DHT-Directory, we observe that the improvement factor increases with scale and can reach 12 for the average lookup latency and 2 for the average transfer distance. Indeed, when a petal has more content peers submitting queries and becoming providers of the requested content, searches in this petal will have larger scopes and thus are more likely to be resolved within this petal. That is why large scales are also advantageous for search speed and localization of close results in Flower-CDN.

Finally, the results of background bandwidth show that a peer experience around 90 *bps* due to its exchanges. This is very low bandwidth that could be sustained even by modem connections, which proves that Flower-CDN incurs very acceptable overhead via its highly effective gossip protocols. Here, we study the behavior of Flower-CDN wrt. scalability and we summarize the results in Table 4.2 for lack of space.

P		HIT RATIO	AVG LOOKUP	AVG TRANSFER
3000	DHT-Directory	0.41	1544 ms	166 ms
	Flower-CDN	0.7	178 ms	107 ms
5000	DHT-Directory	0.52	1596 ms	165 ms
	Flower-CDN	0.72	141 ms	89 ms
7000	DHT-Directory	0.58	1618 ms	167 ms
	Flower-CDN	0.78	160 ms	91 ms
9000	DHT-Directory	0.59	1692 ms	165 ms
	Flower-CDN	0.79	156 ms	87 ms
11000	DHT-Directory	0.62	1743 ms	164 ms
	Flower-CDN	0.83	143 ms	84 ms
			BACKGROUND TRAFFIC	
3000	Flower-CDN		97 bps	
5000	Flower-CDN		89 bps	
7000	Flower-CDN		91 bps	
9000	Flower-CDN		92 bps	
11000	Flower-CDN		94 bps	

Table 4.2: Scalability comparison.

4.4.3.2 PetalUp-CDN

PetalUp-CDN aims at achieving a graceful scale-up of Flower-CDN. In a nutshell, the goal is to maintain the high performance of Flower-CDN and at the same time limit the load on directory peers as the number of participants reaches massive scales.

We evaluate the performance of PetalUp-CDN through a set of 4 experiments targeting a population size of 15000[1]. Each experiment depicts the behavior of PetalUp-CDN for

[1]Due to memory constraints, we could not simulate more than 15000 peers

a specific value of $maxDirectory$, the construction parameter of PetalUp-CDN. Recall that $maxDirectory$ defines the maximum number of content peers that a directory peer should manage to avoid overload situations. Above this number, an additional directory peer is created for the corresponding petal. The current simulation configuration leads to petals that can at most reach 60 content peers. Thus, $maxDirectory$ is consecutively assigned the values (15; 25; 35) in the first three experiments. The fourth experiment corresponds to an unlimited $maxDirectory$, which brings us back to Flower-CDN. The results of the four experiments are synthesized into four curves, each one depicting the time-based evolution of one of the metrics (background traffic, hit ratio, lookup latency, and transfer distance). They are shown in Figures 4.5.

First, let us analyse the results of background traffic (Figure 4.5a). Our aim is to measure the impact of PetalUp-CDN on the amount of load that a directory peer undergoes due to the keepalive and push messages regularly sent by its subset of content peers. We measure the average background traffic of a participant peer because during an experiment a peer can alternatively become a diretcory peer and a content peer. Obviously, the smaller is $maxDirectory$, the smaller is the traffic load on a directory peer. In particular, the load reduction can reach 33% between Flower-CDN and PetalUp-CDN with $maxDirectory = 15$.

When examining hit ratio evolution (Figure 4.5b), we observe that the four approaches achieve similar results. This demonstrates that the partionning of a petal does not affect the performance of our P2P CDN in handling queries. Whether the set of content peers is managed by one directory peer or distributed accross several directory peers, the system succeeds equally well in locating the requested content.

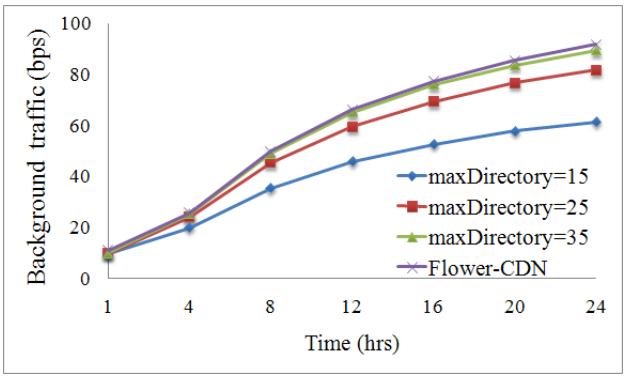
Regarding lookup latency (Figure 4.5c) and transfer distance (Figure 4.5d), the performance is quite the same for all the approaches (with a slight difference of 5 ms in transfer distance). Thus, PetalUp-CDN can achieve the same locality-aware gains as Flower-CDN, independently of the number of directory peers in charge of a petal. In other terms, it can perform fast searches and serve close-by content.

4.4.4 Discussion

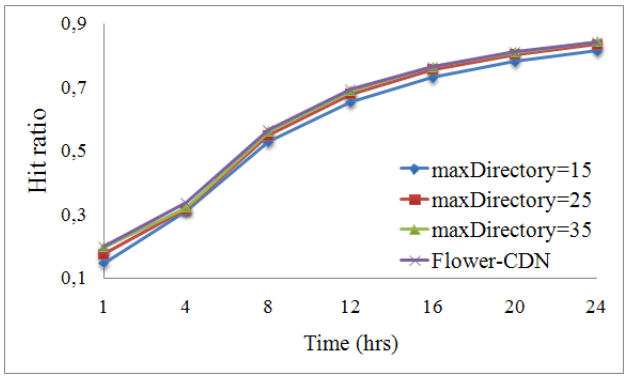
Based on the previous experiments, we conclude that our P2P CDN can maintain an excellent performance under a large-scale and dynamic participation of peers.

With respect to robustness, our maintenance protocols can guarantee a high hit ratio and reduced lookup latency and transfer distance. They provide an efficient detection mechanism for dynamicity via low-cost gossip protocols. Also, they ensure a fast recovery of the P2P CDN that attenuates the loss of directory information and enables a smooth transition. To resume, Flower-CDN and PetalUp-CDN can be extremely robust despite high levels of churn due to the efficient use of gossip.

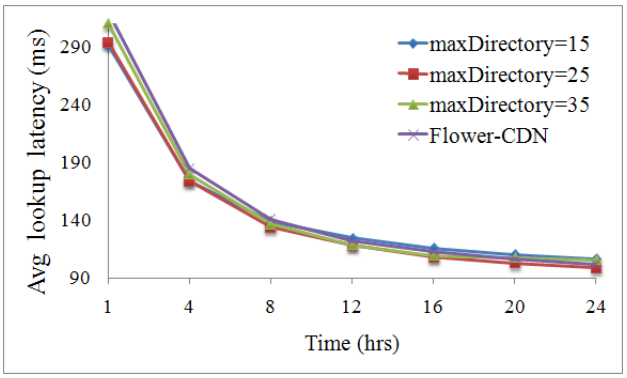
Regarding scalability, Flower-CDN has shown excellent gains despite modest sizes of petals (i.e., a petal size did not exceed 60 peers). We believe that large petals can significantly contribute in increasing the gains. For higher scales, PetalUp-CDN has demonstrated its ability to avoid overload situations without a decline in performance.



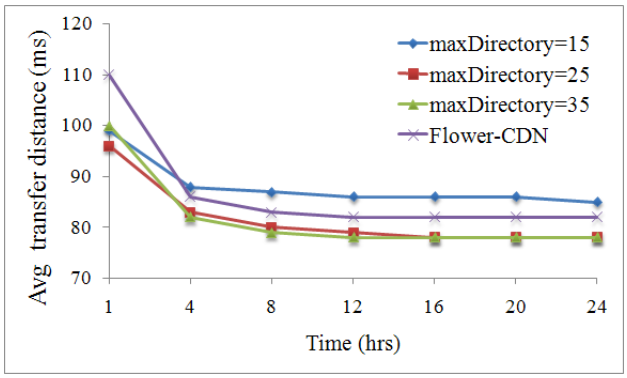
(a) Background traffic.



(b) Hit ratio.



(c) Lookup latency.



(d) Transfer distance.

Figure 4.5: PetalUp-CDN vs. Flower-CDN performance and overhead.

Its multi-directory scheme does not affect hit ratio, transfer distance, and latency lookup when handling queries. The results are extremely promising since they show that our P2P CDN can efficiently support massive scales.

4.5 Conclusion

After demonstrating Flower-CDN's high performance in Chapter 3, this chapter aimed at providing the two other primary requirements of CDN, scalability and reliability/robustness. For scalability purposes, we proposed PetalUp-CDN that extends Flower-CDN to large scales. To avoid overload situations, PetalUp-CDN is designed in a way that allows several directory peers to share the management of a large petal. D-ring is adapted in order to dynamically evolve with respect to the needs of the petals while maintaining its locality- and interest-aware architecture and high performance. The performance evaluation demonstrated that this multi-directory scheme does not affect hit ratio and response times, thus enabling efficient scalability.

Furthermore, we ensured the robustness of both Flower-CDN and PetalUp-CDN via our maintenance protocols. Based on low-cost gossip, these protocols efficiently detect failures and churn, and can recover the P2P CDN smoothly and quickly. Simulation results showed that our approach successfully resists to churn and leverages higher scales to achieve higher improvements. In summary, hit ratio is ameliorated by 40% and response times reduced by a factor of 12, in comparison with an existing P2P CDN.

Chapter 5

Deployment of Flower-CDN

Abstract. In this chapter, we present the main guidelines to deploy Flower-CDN for public use. For this, we propose a browser extension that integrates Flower-CDN functionality into the browsing activity of a user. Flower-CDN extension provides a highly flexible and safe interface through which users can configure their interests with respect to several different websites.

5.1 Introduction

Flower-CDN is deployed over clients that are interested in some particular website and that are willing to participate in order to enjoy a better access for the content of their interest. A website ws is supported by Flower-CDN as long as there are a sufficient number of clients on behalf of ws. More precisely, the more popular a website ws is, the more participants are attracted to Flower-CDN to populate the petals of ws and to occupy its directory peer positions. As for an unpopular website, its petals tend to be empty and its directory peer positions vacant.

A user accesses the Web through its web browser which handles her HTTP requests and accordingly allows her to search and view web content. In order to use and contribute to Flower-CDN transparently, a user should incorporate Flower-CDN functionality into her browser and let it run over HTTP.

We propose a Flower-CDN browser extension that enables P2P content distribution in a transparent and flexible manner. Additionally, it provides configuration management through which users can dynamically update their interests and enforce privacy

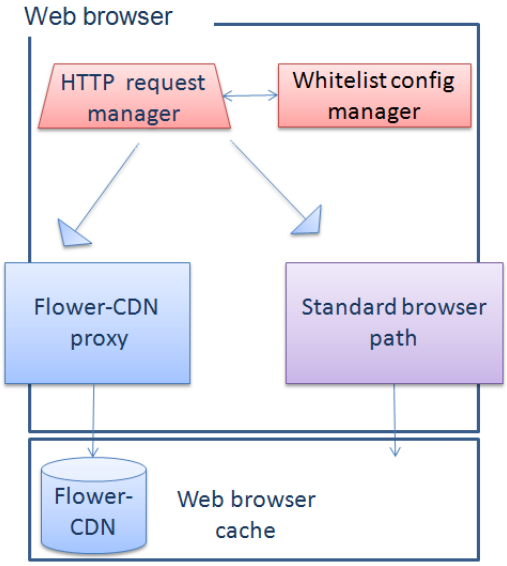


Figure 5.1: Flower-CDN extension within the web browser.

preferences by specifying which content they will share. Finally, we adopt a simple security model that guarantees content integrity even in the face of untrusted peers.

Roadmap. In the following, we first introduce how Flower-CDN can be integrated into the user's web browser in Section 5.2. Then, we deepen our study by describing the implementation architecture of Flower-CDN in Section 5.3. Finally, we conclude in Section 5.4.

5.2 Flower-CDN Browser Extension

Flower-CDN functionality can be implemented as a browser extension. Figure 5.1 illustrates the changes affecting a browser that installs Flower-CDN extension. Flower-CDN operates via three main components: a *whitelist config manager*, an *HTTP request manager* and a *Flower-CDN proxy*.

As shown in Figure 5.1, the content that the user shares in Flower-CDN is stored in a delimited section of the browser cache (i.e., the disk storage allocated for the web browser). This ensures the privacy of the user, because it allows to isolate the web content that the user wants to share from the private content. The amount of disk space allocated to Flower-CDN section grows dynamically as more content is cached, bounded by the available disk space of the browser cache. The cache replacement and expiration policies adopted by the browser cache are used to manage Flower-CDN content (recall

that the content mainly consists of web pages and their embedded objects). Further, the view and directory informations of a peer are also stored in this Flower-CDN section and managed according to their own expiration policies (i.e., the view via gossip exchanges cf. Section 3.4.1.2 and the directory information via push and keepalive cf. Section 4.3.1).

The *whitelist config manager* mantains a list configured by the user and called *Flower-CDN whitelist* that specifies a set of domains referring to websites on behalf of which the user participates to Flower-CDN. The web browsing process begins when the user inputs a URL into the browser and initiates an HTTP request. This request is first handled by the *HTTP request manager*. It checks the URL against the *Flower-CDN whitelist* and forwards the request to the local *Flower-CDN proxy* if the URL matches the whitelist. Otherwise, the HTTP request follows the browser's standard processing path. Upon receiving the request, the Flower-CDN proxy tries first to locally resolve it and then resorts to the Flower-CDN network. The user is connected to the Flower-CDN network as a content or directory peer, via its local proxy which communicates with other Flower-CDN proxies at remote users. Thus, a Flower-CDN proxy handles requests coming from remote users in addition to the local user's requests.

Below, we first give more details on how a Flower-CDN extension is configured wrt. the user's interests and locality. Then, we give more explanation on how a user is connected to the Flower-CDN network (i.e., D-ring or petals).

5.2.1 Configuration

A user may have interest in several websites for which she wants to use Flower-CDN. In Flower-CDN, peers that are related to different websites are involved in different petals and thus have uncorrelated behaviors. Therefore, the user can participate in Flower-CDN as n different peers. She specifies, via the whitelist config manager, the names of the n websites of her interest and the cache section of her Flower-CDN proxy contains n subsections of dynamic sizes.

Figure 5.2 illustrates how a user Suzan is integrated in a Flower-CDN network. Suzan who is in locality 2 is interested in 2 websites α and β. Thus, she is represented in Flower-CDN network as 2 different content peers $c_{\alpha,2}$ and $c_{\beta,2}$. Technically speaking, the Flower-CDN proxy that operates within Susan's browser, manages two different cache subsections, one for each content peer. For instance, the first subsection contains the view and the content maintained by $c_{\alpha,2}$.

Upon the reception of an HTTP request, the Flower-CDN proxy detects the website ws targeted by the request based on its URL. If Suzan has a query for α, her Flower-CDN proxy accesses the Flower-CDN network as $c_{\alpha,2}$ and deals with the local cache subsection of $c_{\alpha,2}$.

Upon its installation by the user, a Flower-CDN browser extension is provided with the number k of localities involved in the system as well as the technique used to detect one's locality. For instance, if we use the landmark-based technique [RHKS02], the user will know the IP adresses of a set of well-known landmarks spread across the network. Thus, she can measure its RTT to the landmarks and orders them by increasing latency. Given

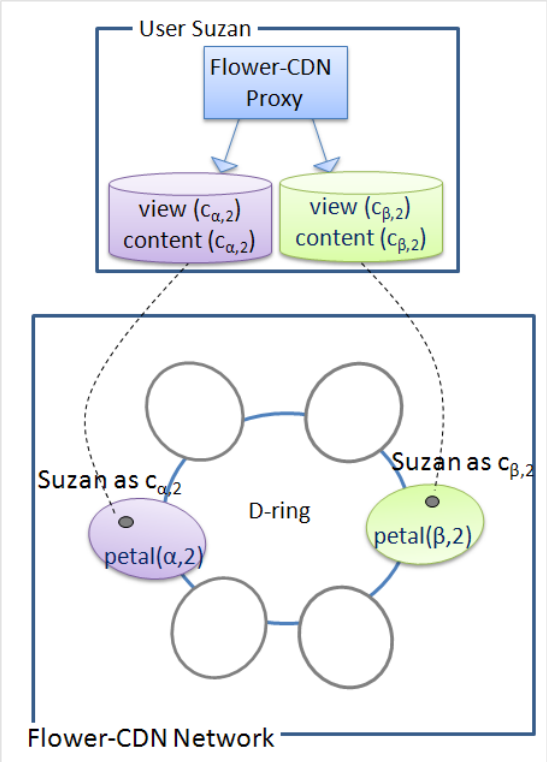


Figure 5.2: A user in Flower-CDN as two content peers related to two different websites.

that each possible landmark ordering identifies a locality, the user detects her locality *loc* based on her ordering.

5.2.2 Connection with Flower-CDN network

Recall that a new client uses D-ring to enter Flower-CDN. Thus, a newly installed Flower-CDN browser extension has a list of IP addresses referring to random directory peers for bootstraping. When the Flower-CDN proxy wants to access Flower-CDN network for the first time, it uses a random bootstrap peer to route its first message over D-ring.

Upon receiving a query, the Flower-CDN proxy detects the target website ws and acts as the corresponding peer p. If this is the first query for ws, p needs to access D-ring. Thus, it computes the key reflecting the website targeted by the query and the locality of p and picks a random bootstrap peer which invokes the DHT routing procedure to forward the query to the target directory peer. If p has already submitted queries for ws, p acts as a directory or content peer of ws according to its acquired role, and uses its view to connect to peers from its petal hosted by remote users via their Flower-CDN local proxies.

A user may reconnect after a temporary disconnection or failure. In such a case, each one of her peers p does not necessarily have to take all the way via D-ring as if it is a new client. p can act as a content peer and try to renew contacts with other content peers of its petal using its previously built view which is stored within the user browser cache. More precisely, p searches for a contact from its view that is still available to gossip with in order for p to reintegrate its petal. However, if p's view contains no available contact, p cannot reintegrate its petal and thus has to rejoin Flower-CDN as a new client.

5.3 Flower-CDN Implementation

We now go inside the Flower-CDN proxy of each user and describe its implementation architecture. We first introduce the global architecture with its different layers. Then, we focus on Flower-CDN implementation details.

5.3.1 Global Architecture

Flower-CDN relies on a DHT-structured overlay via its D-ring. Let us look at a standard DHT-based application. Then, we discuss Flower-CDN layering and identify the resulting changes.

5.3.1.1 DHT-based Applications

DHT systems provide an infrastructure for distributed storage and search. This is enbaled via two interfaces:

- $put(key, data)$ stores a key and its associated $data$ object in the DHT.
- $get(key)$ retrieves the $data$ object associated with key from the DHT.

Figure 5.3 shows the global architecture of a DHT-based application. On top of the Internet network there is the P2P overlay network that is structured as a DHT. The overlay layer ensures key-baesd routing and location by implementing the lookup method: $lookup(key)$ returns the IP address of the DHT peer in charge of key. In order to guarantee the correctness of lookup, the overlay layer manages peer failures and churn by regularly updating routing tables [RGRK04]. On top of this layer, the distributed storage layer ensures key-based data distribution and search by implementing the put and get methods.

5.3.1.2 Flower-CDN

Figure 5.4 illustrates a global architecture implementing Flower-CDN functionality.

The application layer now integrates two additional services, the HTTP request management and the whitelist config management.

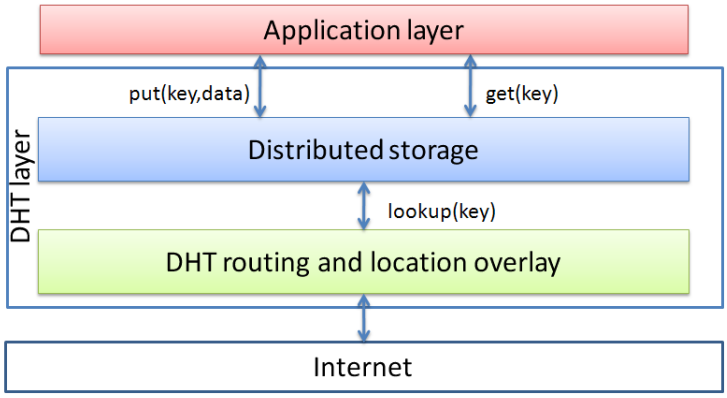


Figure 5.3: DHT-based application global architetcure.

The Flower-CDN proxy layer comprises two adjacent sublayers, the D-ring layer and the petal layer. The petal layer is represented by two main components that depict the behavior of a content peer, *content protocol* and *gossip protocol* (as discussed in Section 3.4). The D-ring layer consists of the key management service supporting the P2P directory service (described in Section 3.3.3). It comes directly over the DHT routing and location overlay. Flower-CDN does not use the distributed storage functionality of the DHT. Thus, the interfaces put and get are deactivated at each peer implementing Flower-CDN.

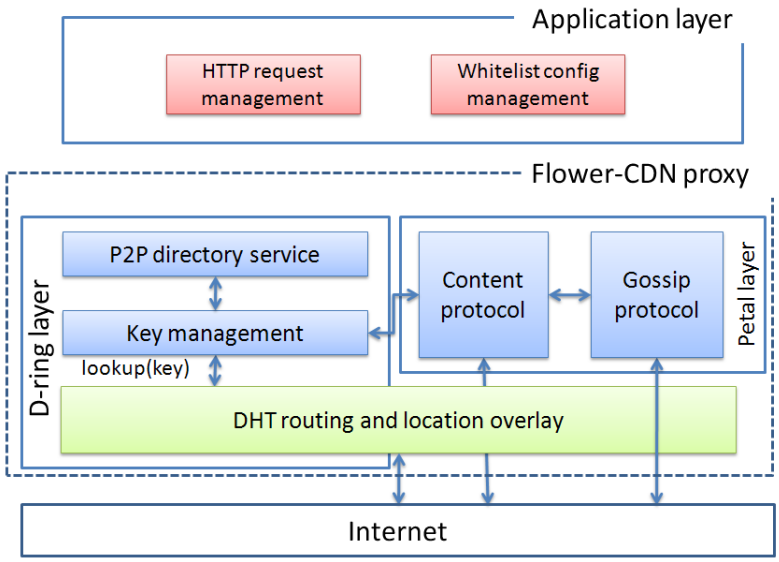


Figure 5.4: Flower-CDN global architecure

Flower-CDN only uses the routing and location overlay of the DHT to build the P2P directory service of D-ring. The DHT-based overlay provides the key-based lookup to route queries. Additionally, it is combined with stabilization protocols that keep routing tables up-to-date despite the arrivals and departures of peers [SMK+01,RD01a]. Basically, a peer periodically checks the liveness of its neighbors in the routing table. It also exchanges routing information with its neighbors to discover newly joining peers and accordingly update its routing table. Therefore, in Flower-CDN, the joins and leaves of directory peers are automatically detected by the DHT-based overlay.

5.3.2 Implementation Architecture

The implementation architecture of Flower-CDN is illustrated in Figure 5.5. It shows seven components connected to links that refer to method invocations. Most of the links have two parts. The component that is connected to the part ending with a circle provides the method. The component that is connected to the part ending with an arc invokes the method. For instance, the component *content protocol* provides the method *processQuery*(q) that can be invoked by the component *interest manager*. In addition, some links have only one part and are only connected via the red circle to one component. This means that the component provides a method to be invoked by remote instances of the same component (e.g., via Java RMI). As an example, *content protocol* provides the method *processQuery*(q) for remote content protocols.

As introduced in Section 5.2.1, the Flower-CDN cache section is subdivided according to the interests of the user. In Figure 5.5, we omit this interest-based subdivision for simplicity. Further, we clearly separate the different types of data (i.e., view, content objects, directory-index, etc.) and include them in their appropriate component.

In this section, we first introduce the architecture components and then present in more detail their sequential interactions.

5.3.2.1 Components

Flower-CDN implementation architecture is organized under seven components: *locality manager*, *key manager*, *interest manager*, *directory protocol*, *content protocol*, *security manager* and *gossip protocol*.

Locality Manager. Its role consists in computing the own locality of the user. It offers one method:

- **getMyLocality**(): returns the locality of the current user.

Key Manager. It provides the key management service of D-ring, as described in Section 3.3.1. This component is used by the local content protocol and directory potocol when they need to access D-ring via the DHT layer. For this, it offers one method:

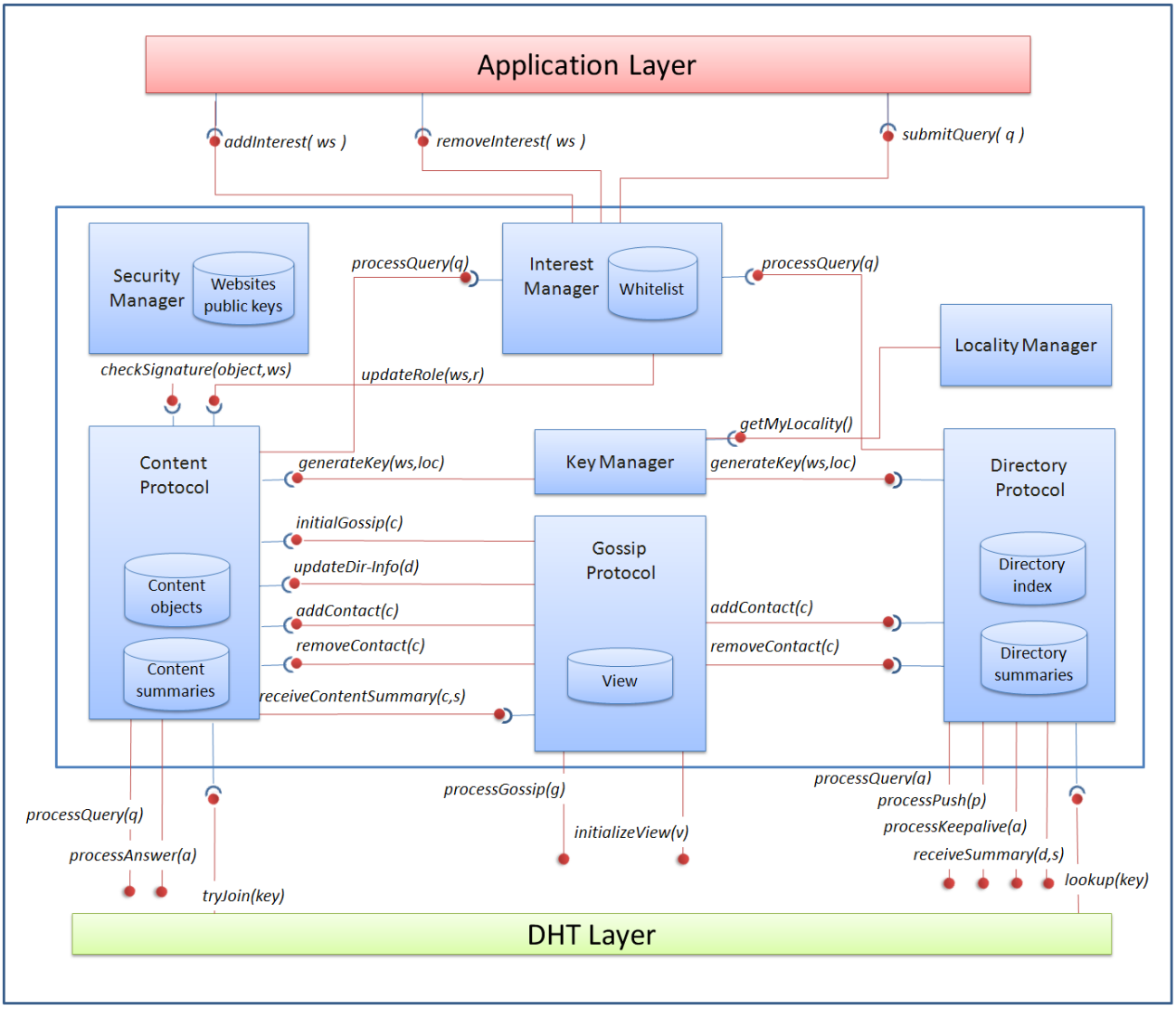


Figure 5.5: Flower-CDN implementation architecture.

- **generateKey**(**ws**, **loc**): generates a key when provided with the url of the target website *ws* and the identfier of the target locality *loc*.

Interest Manager. It holds the whitelist specified by the user (i.e., the user interests in terms of websites for which she wants to contribute in Flower-CDN). It provides two methods for the user interface to update the whitelist according to the current user interests:

- **addInterest**(**ws**): adds the website *ws* to the whitelist.

- **removeInterest**(**ws**): removes the website ws from the whitelist.

In addition, the *interest manager* is responsible of receiving the user queries via:

- **submitQuery**(**q**): enables the user to submit a query q related to a website from the whitelist. It detects the target website ws and redirects the query to the component *directory protocol* if the corresponding peer is a directory peer or to the component *content protocol* if the corresponding peer is a content peer or a new client (see Section 5.2.1). For this, it calls the method $processQuery(q)$ provided by both components.

Recall that for each interest, the user is represented by a peer that has a specific role among directory peer, content peer and client. Upon adding an interest ws, the default role associated with ws is client. Eventually, the role changes with time according to the behavior of the peer. To update the role associated to each website of the whitelist, the folllowing method is provided:

- **updateRole**(**ws**, **r**): sets the role of the local peer related to ws ($r = directory, content, client$).

Directory Protocol. It depicts the behavior of each directory peer $d_{ws,loc}$ hosted by the user. For this, it maintains the data related to each $d_{ws,loc}$ which consists of the *directory-index*(ws, loc) and the *directory-summaries* received from directory peers $d_{ws,loc'}$ of other users (same website as the local directory peer but for different localities). For each method, the *directory protocol* detects the target website and therefore uses the corresponding data. For simplicity, we consider one directory peer $d_{ws,loc}$ hosted by the user (i.e, the user is only interested in website ws).

The *directory protocol* provides the following methods:

- **processQuery**(**q**): handles a query q using Algorithm 2 (Chapter 3). q refers to a query targeting $d_{ws,loc}$. Therefore, this method can be invoked by the *interest manager* for the user's own queries as a directory peer. In addition, it can be invoked by remote directory protocols that have used D-ring to route the query towards $d_{ws,loc}$.

- **processPush**(**p**): processes a push message p received by $d_{ws,loc}$ from one of its content peers $c_{ws,loc}$. This method is invoked by the remote content protocol corresponding to $c_{ws,loc}$.

- **processKeepalive**(**k**): processes a keepalive message k received by $d_{ws,loc}$ from one of its content peers $c_{ws,loc}$. This method is invoked by the remote content protocol corresponding to $c_{ws,loc}$.

- **receiveSummary**(**d**, **s**): receives and stores a directory-summary s from a directory peer d (i.e., a neighbor of $d_{ws,loc'}$ on the D-ring, $d_{ws,loc'}$). This method is invoked by a remote directory protocol corresponding to $d_{ws,loc'}$.

The *directory protocol* can invoke a method of the DHT layer:

- **lookup**(**key**): searches for the directory peer responsible of *key*. This is used to route a query over D-ring towards its target directory peer. Such a query is originated and sent by a new client (hosted by a remote user) that is using the curent directory peer $d_{ws,loc}$ as a bootstrap (cf. Section 5.2.2). The method *lookup*(*key*) connects the local directory protocol to the target directory protocol so that it can invoke *processQuery*(q) on the target component.

Content Protocol. It depicts the behavior of a content peer or a new client related to query processing (cf. Section 3.4.2). It manages the data related to each content peer $c_{ws,loc}$ hosted by the user, which consists of the content objects locally cached by $c_{ws,loc}$ and the *content-summaries* received from remote content peers $c'_{ws,loc}$. Upon running a method, the *content protocol* can detect the target website and thus can manipulate the corresponding data. For simplicity, we consider one content peer $c_{ws,loc}$ hosted by the user (i.e, the user is only interested in website ws).

The *content protocol* provides the method:

- **processQuery**(**q**): processes the query q according to Algorithm 6 (Chapter 3). This method can be invoked by the *interest manager* for the user's own queries as a new client or a content peer. It can also be invoked by remote content and directory protocols related to the same petal as $c_{ws,loc}$ as pointed out in Section 3.4.2.

- **processAnswer**(**a**): processes an answer a that is received for a previously submitted query of the user.

The *content protocol* can invoke a method of the DHT layer:

- **tryJoin**(**key**): tries to join D-ring as a directory peer according to Algorithm 12 and using *key* related to its locality *loc* and website ws.

Security Manager. It ensures the integrity of the content that is transferred to the local user. In an open P2P environment, some peers may be malicious and corrupt the shared content. This problem can be easily solved if web-servers provide digitally signed certificates along with their content [Tay04]. The security manager only needs the website public key to verify the digital signature of an object related to this website and received by the user from some content peer. This solution is indifferent to peer dynamicity and copes well with a loosely-trusted environment. To achieve this solution, it provides one function:

- **checkSignature**(**object**, **ws**): invoked by *content protocol* upon downloading new objects.

Gossip Protocol. It is responsible of the gossip behavior of a peer. It manages the view of every content or directory peer hosted by the local user. For simplicity, we consider one local peer hosted by the user.

The *gossip protocol* provides a method to implement a gossip exchange between two content peers as depicted in Algorithm 3 (Chapter 3:

- **initializeView**(**v**): sets v as an intial view of the local peer (new content peer). This is invoked by a remote gossip protocol when the local peer has recently joined its petal as a content peer.

- **processGossip**(**g**): processes a gossip message g. It is invoked by a remote gossip protocol to initiate a gossip exchange with a local content peer.

The *gossip protocol* also interacts with the local components *directory protocol* and *content protocol* to update the view of the associated directory or content protocol. The associated methods are as follows:

- **addContact**(**c**): adds a new contact c to the view of the local peer.

- **removeContact**(**c**): removes the contact c from the view of the local peer.

- **initialGossip**(**c**): initiates a new client c for gossip exchanges. It consists in sending to c a subset of the local view so that c can initialize its view and become a content peer. This is invoked by a directory or content protocol when they get in touch with a new client (cf. Section 3.4.1.1) .

- **updateDir** − **info**(**d**): updates the *dir-info* which refers to the view entry of the local content peer referring to its current directory peer (cf. Section 3.4.1.3).

- **receiveContentSummary**(**c**, **s**): receives the content-summary s from another content peer c that shares the same petal as the local content peer.

5.3.2.2 Components at Work

We have introduced the Flower-CDN components individually. We now present how they work together by discussing two typical scenarios.

Scenario 1. In Figure 5.6, we consider the case where the component *content protocol* representing a content peer $c_{ws,loc}$ is invoked via $processQuery(q)$. Four users are involved in this scenario: the local user that hosts $c_{ws,loc}$, the remote user 1 that hosts the directory peer $d_{ws,loc}$ of the $petal(ws, loc)$, the remote user 2 that hosts another content peer $c'_{ws,loc}$ of the petal, and the remote user 3 that hosts the originator of the query q.

$c_{ws,loc}$ handles the query according to Algorithm 6 (described in Chapter 3). In case the object o_{ws} is locally cached, $c_{ws,loc}$ answers the query and serves o_{ws} by calling $processAnswer(a)$ of the originator's content protocol. If the latter is a new client, $c_{ws,loc}$ invokes the method $initialGossip(c)$ of the local gossip protocol that also represents $c_{ws,loc}$.

Eventually, the gossip protocol invokes $initializeView(subset)$ on the gossip protocol of the originator in order to initialize the originator's view with a subset of $view(c_{ws,loc})$.

Now, let us look at all the other cases where the object o_{ws} is not in the local cache. In case $c_{ws,loc}$ needs to query the content-summaries for the requested object o_{ws} and has not found any, it redirects the query to $d_{ws,loc}$ by invoking $processQuery(q)$ on the directory protocol of the remote user 1.

In case $c_{ws,loc}$ has found a content-summary related to $c'_{ws,loc}$ showing that the latter might have a copy of o_{ws}, $c_{ws,loc}$ redirects the query to $c'_{ws,loc}$ by invoking $processQuery(q)$ on the content protocol of the remote user 2.

However, if $c'_{ws,loc}$ is not alive (e.g., remote user 2 has disconnected), $c_{ws,loc}$ removes it from is view via its gossip protocol and redirects the query to the original web-server ws.

Scenario 2. In Figure 5.7, we consider the case where the component *content protocol* representing a content peer $c_{ws,loc}$ is invoked via $processAnswer(a)$. Two users are involved in this scenario: the local user that hosts $c_{ws,loc}$ and the remote user 1 that hosts the directory peer of the $petal(ws, loc)$, i.e., $d_{ws,loc}$.

Once a requested object o_{ws} is downloaded, $c_{ws,loc}$ checks its integrity by invoking $checkSignature(o_{ws}, ws)$ of the local interest manager. If validated, it caches the object. Then, it performs the push behavior as depicted in Algorithm 4 of Chapter 3. It checks if the number of changes in the local cache (i.e., content objects) has reached the threshold to notify the directory peer $d_{ws,loc}$. In such a case, $c_{ws,loc}$ sends a push message p to $d_{ws,loc}$ by invoking $processPush(p)$ on the directory protocol of the remote user 1. Upon contacting its $d_{ws,loc}$, $c_{ws,loc}$ updates its directory address information via $updateDir - info()$.

However, if $d_{ws,loc}$ is not alive (e.g., remote user 1 has disconnected), $c_{ws,loc}$ invokes the method $tryJoin(key)$ of the DHT layer in order to replace its directory peer. To obtain the appropriate key with respect to its locality loc and website ws, it invokes $generateKey(ws, loc)$ of the local key manager. Then, it is redirected over D-ring towards the target position. If $c_{ws,loc}$ succeeds in replacing its directory peer, it informs the interest manager about it via $updateRole(ws, directory)$. Otherwise, i.e., if $d_{ws,loc}$ has been already replaced by another peer, $c_{ws,loc}$ invokes $updateDir - info()$ to store the address information about the new $d_{ws,loc}$.

5.4 Conclusion

In this chapter, we presented how Flower-CDN can be implemented and used in practice. The distinctive feature of Flower-CDN functionality is that it can be integrated into the user web browser, allowing the user to transparently contribute and exploit the benefits of Flower-CDN.

Furthermore, we showed that Flower-CDN functionality provides a highly flexible configuration for the user; it can easily manage the different interests of the user and their dynamic changes. We also sillustrated how the content authenticity can be ensured

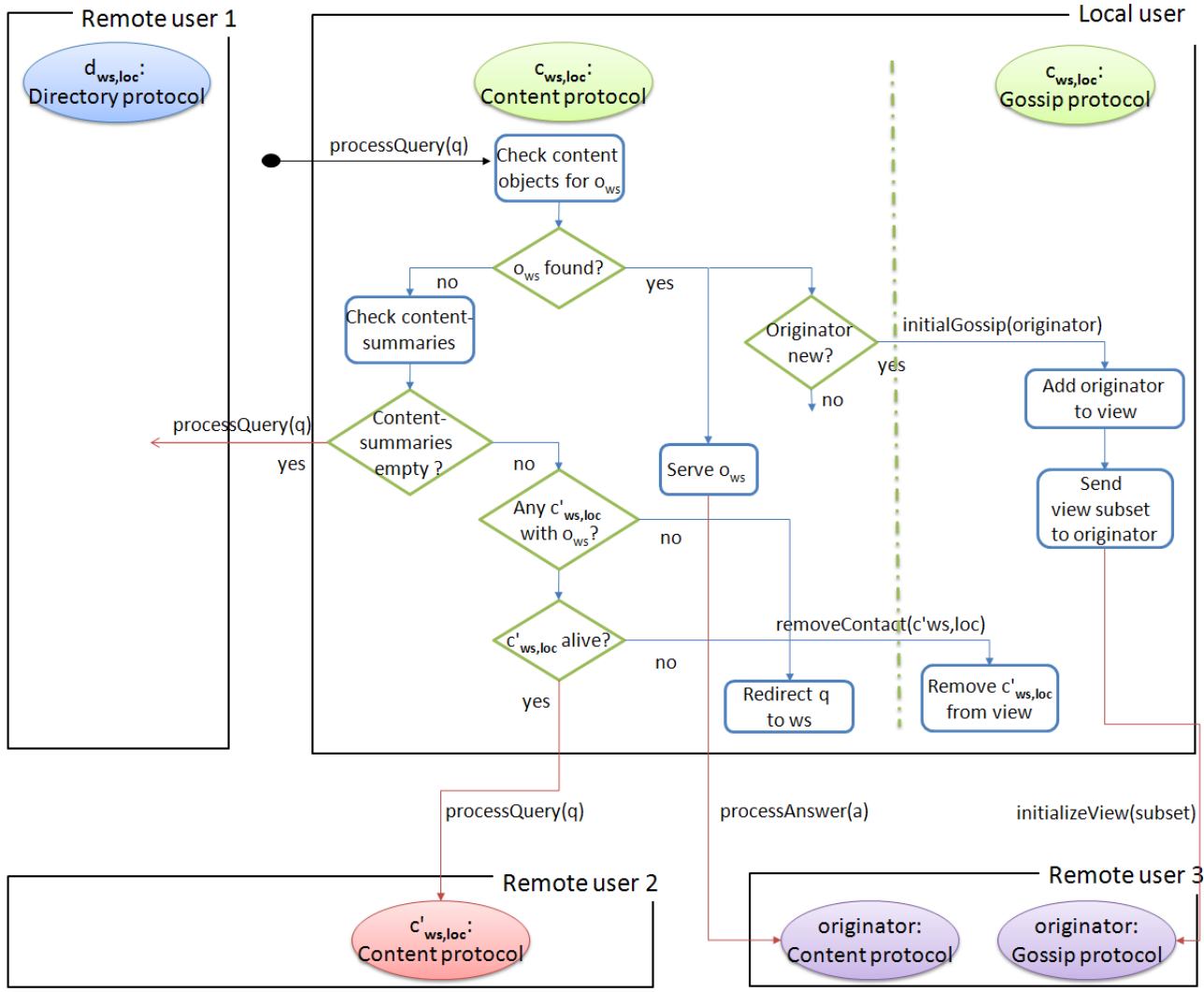


Figure 5.6: Scenario 1: *processQuery*(q) at content protocol

via a straigthforward solution. Finally, we designed the implementation architecture of Flower-CDN, which constitues an important step towards the implementation of Flower-CDN extension and its release for public use. Then each interested user can simply download the extension and start using Flower-CDN.

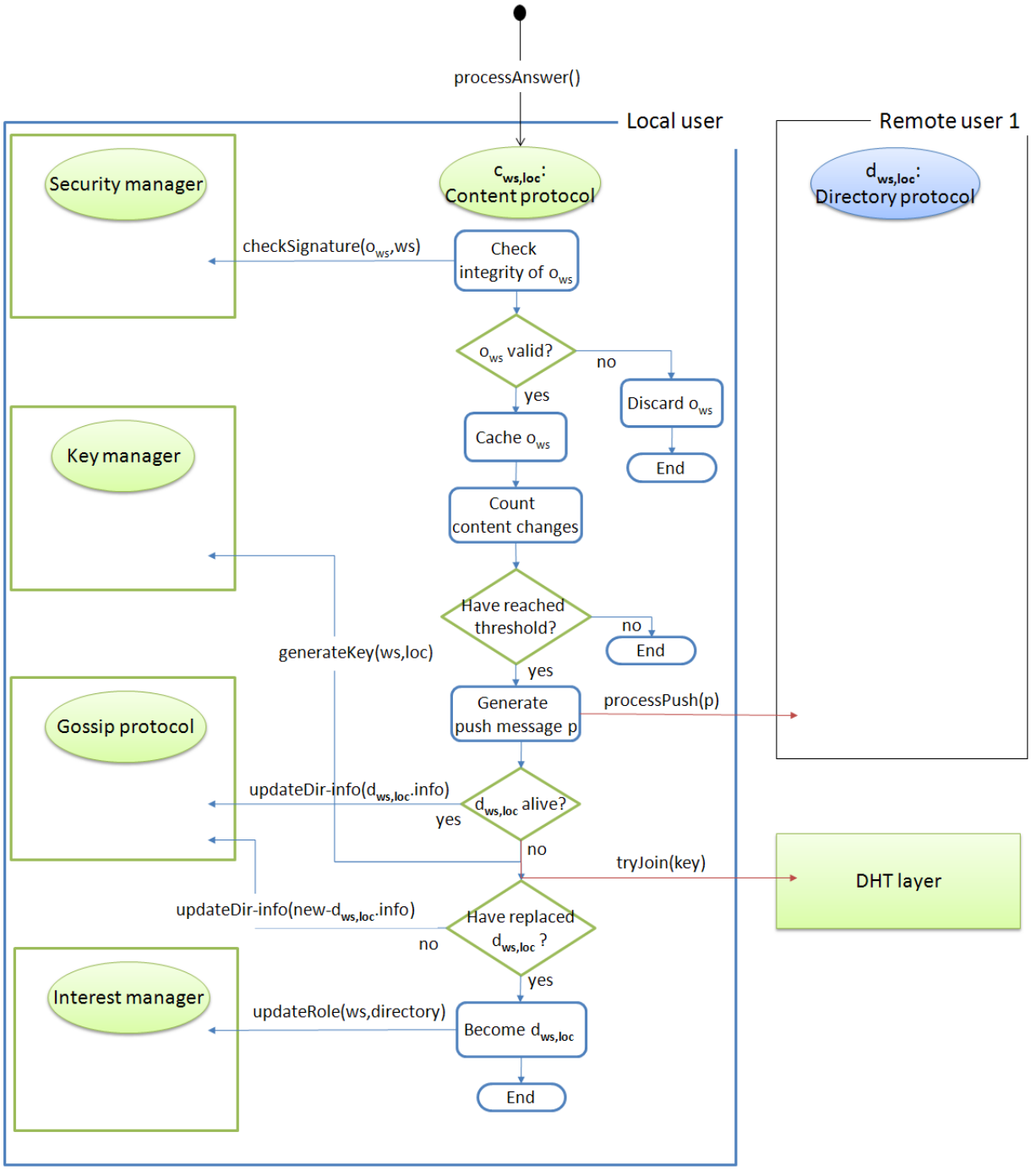


Figure 5.7: Scenario 2: *processAnswer*(q) at content protocol

Conclusion

In this chapter, we summarize our main contributions. Then we give an overview of how we plan to pursue our thesis work.

Summary of Contributions

This thesis has addressed content distribution in P2P systems. It has been motivated by the explosive growth of the Web and its urgent needs in terms of scalable, efficient and low-cost infrastructures for content distribution. At the same time, the rise of Web 2.0 which encourages participation has shed light on P2P collaborations. Thus the aim of this thesis has been to build a P2P infrastructure for content distribution. It has been accomplished in four steps which are summarized below.

1. **State-of-the-Art of P2P Content Distribution:** Our first contribution aimed at a comprehensive survey of P2P content distribution, in order to motivate our next contributions and highlight the shortcomings that we should address. It helped us identify several important observations that can be summarized as follows. First CDNs have stringent requirements which are performance, scalability and reliability. Performance refers to high hit ratio and short response times which can be achieved via locality-aware routing. Scalability ensures that no bottlenecks or decline in performance can result from an increase of the number of clients and queries. Reliability implies that the CDN is robust and does not present any single-point-of failures. Scalabilty is still an open issue as it requires additional investment and prohibitive costs.

   The second observation is that P2P systems are the perfect match to build cheap and scalable CDN. They introduce fundamental requirements like autonomy, expressiveness, efficiency, quality of service, and robustness. Moreover, there are recent trends that refine the P2P overlay network for more performance and efficiency in content distribution. Among these trends, we focused on locality-aware and interest-aware overlay matching, gossip usage and overlay combination. The challenges are to keep the solutions simple, avoid centralized management and large overheads, operate fast and adapt to changes.

Our third observation refers to P2P file sharing, one of the popular applications of P2P content distribution. Such applications tolerate loose garantees on performance as they adopt light-weight and best-effort approaches, and yet are accepted by the user population. Nonetheless, these systems should rigorously aim at keeping the network load at bay to enable a deployment over large-scales. A top priority is to exploit locality-awareness in order to serve files from close-by locations. However, most existing works do not address this issue.

Our final observation concerns P2P CDNs which should strictly meet the requirements of P2P and CDN and cope with their correlations. Existing P2P CDNs compromise one requirement for the other and most importantly, do not address scalability.

2. **Locaware:** The second contribution focused on P2P file sharing, as a first effort to build a basic infrastructure for content distribution with loose requirements. The infrastructure relied on unstructured overlays as they are widely deployed in the context of file sharing because of their flexibility. We addressed the problem of bandwidth consumption, from two angles: search inefficiency and long-distance file transfers. Our solution, Locaware, leverages inherent properties of P2P-file sharing environments which consist of natural replication, temporal and network localities. Locaware peforms index caching that selectively caches query responses in the form of file indexes with locality-aware information. Moreover, Locaware adapts its index caching scheme for keyword search and leverages the scheme for query routing. These achievements are realized with a perfectly acceptable overhead in terms of storage and bandwidth requirements. Through simulation, we showed that Locaware significantly improves the success rate of selective indexing caching solutions and reduces the traffic of flooding solutions. Most importantly, the results demonstrated that Locaware can limit wasted bandwidth and reduce network resource usage.

   This contribution was published in [DPV07,DP09]. Our initial efforts on distributed algorithms for P2P collaboration which have pioneered our work on locality-awareness were subject to publication in [DMP07, MPDJP08].

3. **Flower-CDN:** The third contribution aimed towards a more sophisticated infrastructure for content distribution. It targets the requirement of performance through locality-awareness, the requirement of autonomy through interest-awareness and the requirement of efficiency through overlay combination. Flower-CDN enables any website to distribute its content, by strictly relying on the community interested in its content. Flower-CDN builds a query routing infrastructure that intelligently combines DHT efficiency with gossip robustness. Furthermore, it exploits peer interests and localities for efficient collaboration and content distribution. The P2P directory service, D-ring, relies on a novel DHT mechanism that can be easily integrated into existing structured overlays, whereas the petals are constructed and managed via cheap gossip protocols. We analytically and empirically analysed our

gossip protocols to efficiently tune their parameters and control their overhead. Through extensive simulations, Flower-CDN showed high performance because it performs fast searches and finding close-by results. Furthermore, gossip incured acceptable overhead in terms of bandwidth consumption, which could be adapted to the available network resources and hit ratio requirements. Our results demonstrate that the design choices of our hybrid architecture are perfectly adapted to the context.

The initial proposal of Flower-CDN was published in [DPK09a] and an improved proposal appeared in [DPK09d].

4. **Robustness and Scalability for Flower-CDN:** The fourth contribution focused on the two requirements of CDN, scalability and reliability/robustness. For scalability purposes, we proposed PetalUp-CDN that extends Flower-CDN to large scales. To avoid overload situations, D-ring dynamically evolves with respect to the needs of the petals while maintaining its locality- and interest-aware architecture and high performance. Furthermore, we ensured the robustness of our approach via maintenance protocols that are based on low-cost gossip. These protocols efficiently detect failures and churn, and can recover the P2P CDN smoothly and quickly. Simulation results showed that our approach successfully resists to churn and leverages higher scales to achieve higher improvements. In summary, hit ratio is ameliorated by 40% and response times reduced by a factor of 12, in comparison with an existing P2P CDN.

   An overview of this contribution first appeared in [DPK09c], then a more detailed version was published in [DPK09b].

5. **Flower-CDN Deployment:** The final contribution provided the first guidelines to make Flower-CDN available for public use. We proposed to implement Flower-CDN functionality as an extension for the user's web browser. As such, the user enjoys a transparent, flexible and highly configurable experience with Flower-CDN. We designed an implementation architecture that covers security and privacy issues in a simple and practical manner.

Future Work

Our future work focuses on providing Flower-CDN with more advanced features and extending it to other contexts. We present in the following a non-exhaustive list of work that we plan to carry out.

1. **Trace-Driven Experimentation:** We are currently refining our simulation studies by injecting real Web traces. Traces can accurately depict the user browsing behavior and its variants. This can make the simulations more realistic.

2. **Flower-CDN Browser Extension:** Another on-going work is to finalize and test the implementation of Flower-CDN browser extension. We hope that interested users would be able to download the browser extension and start using Flower-CDN.

3. **Semantic-Based Interests and Search:** We intend to go one step further and empower users to express their interests in more granular and complex ways. An interest should reflect semantic preferences via ontologies, combination of topics, etc. Accordingly, participants should be able to conduct semantic searches where queries can be expressed with keywords or other semantic information rather than strict URLs. This would help users navigate through available content and discover objects that would interest them.

4. **Social Content Sites:** Web 2.0 is fostering the integration of content information with the social information (profiles, connections and activities) of users, giving rise to social content sites. Sites that started as pure content oriented (e.g., Youtube, Yahoo Travel) or pure social networking (e.g., Facebook, MySpace) are rapidly evolving towards such an integration. However, these sites are supported by a centralized architecture, which incurs high cost and suffers from availability, privacy, and scalability problems. To handle this, we envision the deployment of such sites over the decentralized infrastructure of Flower-CDN. Obviously, such a proposal requires a complete study of the changes that should target Flower-CDN like the concept of interests, the search techniques, etc.

5. **Extending IP Routing with Data Semantics for CDN:** Finally, we plan to deepen our investigation into locality-awareness and further refine the routing in Flower-CDN. The IP routing protocol has proven to be highly scalable, supporting millions of nodes, and allowing dynamic behavior of nodes which can be added or removed autonomously. Thus, a promising research direction is to extend IP routing with data semantics in order to better support CDN like queries in Flower-CDN. The extensions should consider distributed data management techniques. This work should be done in two steps. First, we need to better understand how P2P routing is mapped into IP routing and to quantify the performance limitations resulting from routing protocol translation. Second, based on this understanding, the goal is to propose extensions to IP routing with data semantics to support CDN like queries.

Bibliography

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Communicaions of the ACM*, 45(11):56–61, 2002.

[ATS04] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, 2004.

[BBK02] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 205–217, 2002.

[BCF+99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: evidence and implications. In *Proceedings of the 18th IEEE International Conference on Computer Communications (INFOCOM)*, pages 126–134, 1999.

[Bir07] Ken Birman. The promise, and limitations, of gossip protocols. *ACM SIGOPS Operating Systems Review*, 41(5):8–13, 2007.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BPV08] Rajkumar Buyya, Mukaddim Pathan, and Athena Vakali. *Content Delivery Networks*. LNEE. Springer, 2008.

[CCR04] Miguel Castro, Manuel Costa, and Antony I. T. Rowstron. Should we build Gnutella on a structured overlay? *ACM SIGCOMM Computer Communication Review*, 34(1):131–136, 2004.

[CDF+98] Ramón Cáceres, Fred Douglis, Anja Feldmann, Gideon Glass, and Michael Rabinovich. Web proxy caching: the devil is in the details. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):11–15, 1998.

[CG01] Bengt Carlsson and Rune Gustavsson. The rise and fall of napster - an evolutionary approach. In *Proceedings of the 6th International Computer Science Conference on Active Media Technology (AMT)*, volume 2252 of *LNCS*, pages 347–354. Springer, 2001.

[CGM02] Arturo Crespo and Hector Garcia-Molina. Routing indices for peer-to-peer systems. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 23–, 2002.

[CGM04] Arturo Crespo and Hector Garcia-Molina. Semantic overlay networks for P2P systems. In *Proceedings of the 3rd International Workshop on Agents and Peer-to-Peer Computing (AP2PC)*, volume 3601 of *LNCS*, pages 1–13. Springer, 2004.

[CJ06] An-Hsun Cheng and Yuh-Jzer Joung. Probabilistic file indexing and searching in unstructured peer-to-peer networks. *Computer Networks*, 50(1):106–127, 2006.

[CMH+02] Ian Clarke, Scott G. Miller, Theodore W. Hong, Oskar Sandberg, and Brandon Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[CMT01] Ian Cooper, Ingrid Melve, and Gary Tomlinson. Internet Web replication and caching taxonomy. Internet Engineering Task Force RFC 3040, 2001. `www.ietf.org/rfc/rfc3040.txt`.

[CRB+03] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making Gnutella-like P2P systems scalable. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 407–418, 2003.

[CS02] Edith Cohen and Scott Shenker. Replication strategies in unstructured P2P networks. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures,and Protocols for Computer Communications (SIGCOMM)*, pages 177–190, 2002.

[CW04] Hailong Cai and Jun Wang. Foreseer: A novel, locality-aware peer-to-peer system architecture for keyword searches. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pages 38–58, 2004.

[DGH+87] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–12, 1987.

[DGMY02] Neil Daswani, Hector Garcia-Molina, and Beverly Yang. Open problems in data-sharing peer-to-peer systems. In *Proceedings of the 9th International Conference on Database Theory (ICDT)*, volume 2572 of *LNCS*, pages 1–15. Springer, 2002.

[DHA03] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 76–, 2003.

[DKK+01] Frank Dabek, M. Frans Kaashoek, David R. Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–215, 2001.

[DMP07] Manal El Dick, Vidal Martins, and Esther Pacitti. A topology-aware approach for distributed data reconciliation in P2P networks. In *Proceedings of the 13th International Conference on Parallel and Distributed Computing (Euro-Par)*, volume 4641 of *LNCS*, pages 318–327. Springer, 2007.

[DP09] Manal El Dick and Esther Pacitti. Locaware: Index caching in unstructured P2P-file sharing systems. In *Proceedings of the 2nd ACM International Workshop on Data Management in Peer-to-peer systems (DAMAP)*, page 3, 2009.

[DPK09a] Manal El Dick, Esther Pacitti, and Bettina Kemme. Flower-cdn: a hybrid P2P overlay for efficient query processing in CDN. In *Proceedings of the 12th ACM International Conference on Extending Database Technology (EDBT)*, pages 427–438, 2009.

[DPK09b] Manal El Dick, Esther Pacitti, and Bettina Kemme. A highly robust P2P-CDN under large-scale and dynamic participation. In *Proceedings of the 1st International Conference on Advances in P2P Systems (AP2PS)*, pages 180–185, 2009.

[DPK09c] Manal El Dick, Esther Pacitti, and Bettina Kemme. Leveraging P2P overlays for large-scale and highly robust content distribution and search. In *Proceedings of the VLDB PhD Workshop*, 2009.

[DPK09d] Manal El Dick, Esther Pacitti, and Bettina Kemme. Un réseau pair-à-pair de distribution de contenu exploitant les intérêts et les localités des pairs. In *Actes desÂ 23èmes Journées Bases de Données Avancées, (BDA) (Informal Proceedings)*, pages 407–388, 2009.

[DPV07] Manal El Dick, Esther Pacitti, and Patrick Valduriez. Location-aware index caching and searching for P2P systems. In *Proceedings of the 5th International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, 2007.

[EGKM04] Patrick T. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulieacute. Epidemic information dissemination in distributed systems. *IEEE Computer*, 37(5):60–67, 2004.

[FCAB98] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 291–293, 1998.

[FFM04] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 239–252, 2004.

[FHKM04] Fabrice Le Fessant, Sidath B. Handurukande, Anne-Marie Kermarrec, and Laurent Massoulié. Clustering in P2P file sharing workloads. In *Proceedings of the 3rd International Workshop on P2P Systems (IPTPS)*, volume 3279 of *LNCS*, pages 217–226. Springer, 2004.

[GDS+03] P. Krishna 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 *Proceedings of the 19th ACM Symposium on Operating systems principles (SOSP)*, pages 314–329, 2003.

[GGG+03] P. Krishna Gummadi, Ramakrishna Gummadi, Steven D. Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures,and Protocols for Computer Communications (SIGCOMM)*, pages 381–394, 2003.

[Gnu05] Gnutella protocol specification. `http://wiki.limewire.org/index.php?title=GDF`, 2005.

[HHH+02] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. Complex queries in DHT-based peer-to-peer networks. In *Revised Papers from the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, LNCS, pages 242–259. Springer, 2002.

[HHL+03] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB)*, pages 321–332, 2003.

[HKFM04] Sidath B. Handurukande, Anne-Marie Kermarrec, Fabrice Le Fessant, and Laurent Massoulié. Exploiting semantic clustering in the eDonkey P2P network. In *Proceedings of the 11th ACM SIGOPS European Workshop*, page 20, 2004.

[iR08] AccuStream iMedia Research. CDN market growth 2006 - 2009, 2008. `www.researchandmarkets.com`.

[IRD02] Sitaram Iyer, Antony I. T. Rowstron, and Peter Druschel. Squirrel: a decentralized P2P web cache. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 213–222, 2002.

[ITU07] ITU World Telecommunication/ICT Indicators Database. International Telecommunication Union - Development Sector, 2007. `http://www.itu.int/ITU-D/ict/statistics/ict/index.html`.

[JAB01] Mihajlo A. Jovanovic, Fred S. Annexstein, and Kenneth A. Berman. Scalability issues in large peer-to-peer networks: a case study of Gnutella. Technical report, ECECS Department, University of Cincinnati, 2001.

[JB05] Márk Jelasity and Özalp Babaoglu. T-Man: Gossip-based overlay topology management. In *Proceedings of the 3rd International Workshop on Engineering Self-Organising Systems (ESOA)*, volume 3910 of *LNCS*, pages 1–15. Springer, 2005.

[JGKvS04] Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, volume 3231 of *LNCS*, pages 79–98. Springer, 2004.

[JMJV] Márk Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. The PeerSim simulator. `http://peersim.sf.net`.

[Jov00] Mihajlo A. Jovanovic. Modelling large-scale peer-to-peer networks and a case study of Gnutella. Master's thesis, ECECS Department, University of Cincinnati, 2000.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven E. Czerwinski, Patrick R. Eaton, Dennis Geels, Ramakrishna Gummadi, Sean C. Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Y. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–201, 2000.

[KGZY02] Vana Kalogeraki, Dimitrios Gunopulos, and Demetrios Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. In *Proceedings of the 11th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 300–307, 2002.

[KLL+97] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th ACM Symposium on the Theory of Computing (STOC)*, pages 654–663, 1997.

[KM02] Magnus Karlsson and Mallik Mahalingam. Do we need replica placement algorithms in content delivery networks? In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution (WCW)*, pages 117–128, 2002.

[KRP05] Thomas Karagiannis, Pablo Rodriguez, and Konstantina Papagiannaki. Should internet service providers fear peer-assisted content distribution? In *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC)*, pages 63–76, 2005.

[KS94] Michael L. Katz and Carl Shapiro. Systems competition and network effects. *Journal of Economic Perspectives*, 8(2):93–105, 1994.

[KvS07] Anne-Marie Kermarrec and Maarten van Steen. Gossiping in distributed systems. *ACM SIGOPS Operating Systems Review*, 41(5):2–7, 2007.

[KWX01] Balachander Krishnamurthy, Jia Wang, and Yinglian Xie. Early measurements of a cluster-based architecture for P2P systems. In *Proceedings of the 1st ACM SIGCOMM Internet Measurement Workshop*, pages 105–109, 2001.

[LBBsS02] Nathaniel Leibowitz, Aviv Bergman, Roy Ben-shaul, and Aviv Shavit. Are file swapping networks cacheable? characterizing P2P traffic. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution (WCW)*, 2002.

[LCC+02] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th ACM International Conference on Supercomputing (ICS)*, pages 84–95, 2002.

[LGB03] Prakash Linga, Indranil Gupta, and Ken Birman. A churn-resistant P2P web caching system. In *Proceedings of the ACM Workshop on Survivable and Self-Regenerative Systems (SSRS)*, pages 1–10, 2003.

[LKR06] Jian Lianga, Rakesh Kumarb, and Keith W. Ross. The FastTrack overlay: A measurement study. *Computer Networks Journal*, 50(6):842–858, 2006.

[LOZB96] Julia Porter Liebeskind, Amalya Lumerman Oliver, Lynne Zucker, and Marilynn Brewer. Social networks, learning, and flexibility: Sourcing scientific knowledge in new biotechnology firms. *Organization Science*, 7(4):428–443, 1996.

[LXL+05] Yunhao Liu, Li Xiao, Xiaomei Liu, Lionel M. Ni, and Xiaodong Zhang. Location awareness in unstructured P2P systems. *IEEE Transaction on Parallel and Distributed Systems*, 16(2):163–174, 2005.

[Mar02] Evangelos P. Markatos. Tracing a large-scale peer-to-peer system: An hour in the life of Gnutella. In *Proceedings of 2nd the IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 65–74, 2002.

[MBK07] Balasubramaneyam Maniymaran, Marin Bertier, and Anne-Marie Kermarrec. Build one, get one free: Leveraging the coexistence of multiple P2P overlay networks. In *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems (ICDCS)*, page 33, 2007.

[MLMB02] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. BRITE topology generator, 2002. `http://www.cs.bu.edu/brite/`.

[Moh01] C. Mohan. Caching technologies for Web applications. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, page 726, 2001.

[MPDJP08] Vidal Martins, Esther Pacitti, Manal El Dick, and Ricardo Jiménez-Peris. Scalable and topology-aware reconciliation on P2P networks. *Journal of Distributed and Parallel Databases*, 24(1-3):1–43, 2008.

[NT04] Nikos Ntarmos and Peter Triantafillou. Aesop: altruism-endowed self-organizing peers. In *Proceedings of the 2nd International Workshop on Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P)*, volume 3367/2005 of *LNCS*, pages 151–165. Springer, 2004.

[NWQ+02] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. Edutella: a P2P networking infrastructure based on rdf. In *Proceedings of the 11th ACM International Conference on World Wide Web (WWW)*, pages 604–615, 2002.

[O'R05] Tim O'Reilly. What is Web 2.0. O'Reilly Media, 2005. `http://oreilly.com/pub/a/web2/archive/what-is-web-20.html?page=2`.

[Pen03] Gang Peng. CDN: Content distribution network. Technical report TR-125, Experimental Computer Systems Lab, Department of Computer Science, State University of New York, 2003.

[PGES05] Johan A. Pouwelse, Pawel Garbacki, Dick H. J. Epema, and Henk J. Sips. The Bittorrent P2P file-sharing system: Measurements and analysis. In *Proceedings of the 4th International Workshop on P2P Systems (IPTPS)*, volume 3640 of *LNCS*, pages 205–216. Springer, 2005.

[PH03] Sunil Patro and Y. Charlie Hu. Transparent query caching in peer-to-peer overlay networks. In *Proceedings of the 17th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 32a, 2003.

[PS02] Venkata N. Padmanabhan and Kunwadee Sripanidkulchai. The case for cooperative networking. In *Proceedings of the 1st International Workshop on P2P Systems (IPTPS)*, volume 2429 of *LNCS*, pages 178–190. Springer, 2002.

[PV06] George Pallis and Athena Vakali. Insight and perspectives for content delivery networks. *Communications of the ACM*, 49(1):101–106, 2006.

[PWP+04] Vivek S. Pai, Limin Wang, KyoungSoo Park, Ruoming Pang, and Larry Peterson. The dark side of the Web: an open proxy's view. *ACM SIGCOMM Computer Communication Review*, 34(1):57–62, 2004.

[QB06] Yi Qiao and Fabián E. Bustamante. Structured and unstructured overlays under the microscope: a measurement-based view of two P2P systems that people use. In *Proceedings of the USENIX Annual Technical Conference (ATEC)*, pages 341–355, 2006.

[RD01a] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale P2P systems. In *Proceedings of the 2nd ACM/IFIP International Conference on Middleware*, volume 2218 of *LNCS*, pages 329–350. Springer, 2001.

[RD01b] Antony I. T. Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 188–201, 2001.

[RFB07] Weixiong Rao, Lei Chen 0002, Ada Wai-Chee Fu, and Yingyi Bu. Optimal proactive caching in P2P network: analysis and application. In *Proceedings of the 6th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 663–672, 2007.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 161–172, 2001.

[RFI02] Matei Ripeanu, Ian T. Foster, and Adriana Iamnitchi. Mapping the Gnutella network: Properties of large-scale P2P systems and implications for system design. *IEEE Internet Computing*, 6(1):50–57, 2002.

[RGRK04] Sean C. Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a dht. In *Proceedings of the USENIX Annual Technical Conference, General Track*, pages 127–140, 2004.

[RHKS02] Sylvia Ratnasamy, Mark Handley, Richard M. Karp, and Scott Shenker. Topologically-aware overlay construction and server selection. In *Proceedings of the 21st IEEE International Conference on Computer Communications (INFOCOM)*, pages 1190–1199, 2002.

[Rit01] Jordan Ritter. Why Gnutella can't scale, no, really. `http://www.darkridge.com/jpr5/doc/gnutella.html`, 2001.

[RKCD01] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Proceedings of the 3rd International Workshop on Networked Group Communication (NGC)*, volume 2233 of *LNCS*, pages 30–43. Springer, 2001.

[RMH98] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. Technical report TR98-1687, Cornell University, 1998.

[RSS02] Sylvia Ratnasamy, Ion Stoica, and Scott Shenker. Routing algorithms for DHTs: Some open questions. In *Revised Papers from the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *LNCS*, pages 45–52. Springer, 2002.

[RY05] Young-Suk Ryu and Sung-Bong Yang. An effective P2P web caching system under dynamic participation of peers. *IEICE Transactions*, 88-B(4):1476–1483, 2005.

[SGD+02] Stefan Saroiu, P. Krishna Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy. An analysis of Internet content delivery systems. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 315–327, 2002.

[SMB02] Tyron Stading, Petros Maniatis, and Mary Baker. P2P caching schemes to address flash crowds. In *Proceedings of the 1st International Workshop on P2P Systems (IPTPS)*, volume 2429 of *LNCS*, pages 203–213. Springer, 2002.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: a scalable P2P lookup service for Internet applications. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 149–160, 2001.

[SMZ03] Kunwadee Sripanidkulchai, Bruce M. Maggs, and Hui Zhang. Efficient content location using interest-based locality in P2P systems. In *Proceedings of the 22nd IEEE International Conference on Computer Communications (INFOCOM)*, pages 2166–2176, 2003.

[SPV06] Konstantinos Stamos, George Pallis, and Athena Vakali. Integrating caching techniques on a content distribution network. In *Proceedings of the 10th East European Conference Advances in Databases and Information Systems (ADBIS)*, volume 4152 of *LNCS*, pages 200–215. Springer, 2006.

[SR02] Anurag Singla and Christopher Rohrs. Ultrapeers: Another step towards Gnutella scalability. Gnutella Developers Forum, 2002.

[SR05] Daniel Stutzbach and Reza Rejaie. Characterizing churn in P2P networks. Technical report CIS-TR-2005-03, University of Oregon, 2005.

[SR06] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement (IMC)*, pages 189–202, 2006.

[Sri01] Kunwadee Sripanidkulchai. The popularity of Gnutella queries and its implication on scaling. `http://www.cs.cmu.edu/~kunwadee/research/p2p/gnutella.html`, 2001.

[SRS02] Angelos Stavrou, Dan Rubenstein, and Sambit Sahu. A lightweight, robust P2P system to handle flash crowds. In *Proceedings of the 10th IEEE International Conference on Network Protocols (ICNP)*, page 226, 2002.

[SRS05] Yee Jiun Song, Venugopalan Ramasubramanian, and Emin Gun Sirer. Optimal resource utilization in content distribution networks. Technical report TR2005-2004, Cornell University, 2005.

[SW04] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. *IEEE/ACM Transactions Networking*, 12(2):219–232, 2004.

[SX08] Haiying Shen and Cheng-Zhong Xu. Hash-based proximity clustering for efficient load balancing in heterogeneous DHT networks. *Journal of Parallel and Distributed Computing*, 68(5):686–702, 2008.

[Tay04] Ian J. Taylor. *From P2P to Web services and Grids: peers in a client/server world*, chapter Security. Springer, 2004.

[Tec99] Akamai Technologies. Fast Internet content delivery with FreeFlow. White paper, 1999. `http://www.cs.purdue.edu/homes/cs536/lecture/freeflow.pdf`.

[Tec04] Akamai Technologies. Akamai - the business Internet - a predictable platform for profitable e-business. White paper, 2004. `http://www.akamai.com/dl/Whitepapers/Akamai_Business_Internet_Whitepaper.pdf`.

[VGS05] Spyros Voulgaris, Daniela Gavidia, and Maarten Steen. Cyclon: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.

[VvS05] Spyros Voulgaris and Maarten van Steen. Epidemic-style management of semantic overlays for content-based searching. In *Proceedings of the 11th International Conference on Parallel and Distributed Computing (Euro-Par)*, volume 3648 of *LNCS*, pages 1143–1152. Springer, 2005.

[Wan99] Jia Wang. A survey of Web caching schemes for the Internet. *ACM SIGCOMM Computer Communication Review*, 29(5):36–46, 1999.

[WNO+02] Xiaoyu Wang, Wee Siong Ng, Beng Chin Ooi, Kian-Lee Tan, and Aoying Zhou. Buddyweb: A P2P-based collaborative web caching system. In *Revised papers from the NETWORKING Workshops on Web Engineering and Peer-to-Peer Computing*, volume 2376 of *LNCS*, pages 247–251. Springer, 2002.

[WXLZ06] Chen Wang, Li Xiao, Yunhao Liu, and Pei Zheng. DiCAS: An efficient distributed caching mechanism for P2P systems. *IEEE Transactions Parallel Distribed Systems*, 17(10):1097–1109, 2006.

[YGM02] Beverly Yang and Hector Garcia-Molina. Improving search in peer-to-peer networks. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 5–14, 2002.

[ZHS+04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

[ZZJ+01] Shelley Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 11–20, 2001.

A

Résumé Étendu

A.1 Introduction

Tributaire de certaines technologies, le web 2.0 est avant tout un changement de paradigme. L'usager n'est plus un simple consommateur d'informations (i.e., contenu); il peut désormais devenir un acteur du réseau et un producteur de contenu. Il y a une "architecture de participation" implicite, une éthique de coopération incorporée dans laquelle les usagers peuvent interagir entre eux, créer et partager du contenu. Les applications du web 2.0 fournissent un service qui s'améliore automatiquement quand plus de gens l'utilisent. Quelques exemples d'applications web 2.0 sont l'encyclopédie en ligne Wikipedia, qui permet aux particuliers de créer et modifier des articles (contenu), les sites de réseaux sociaux comme Facebook, les sites de partage de photos et vidéos comme YouTube et Flickr, ainsi que les wikis et les blogs.

À l'heure où l'Internet connaît une croissance foudroyante, le web 2.0 a favorisé l'émergence de la technologie P2P comme un nouveau modèle de communication. Le modèle P2P s'oppose au modèle traditionnel client-serveur, chaque pairétant à la fois client et serveur. Ainsi, les réseaux P2P reposent sur un principe d'égalité et de partage de ressources entre les pairs, en s'appuyant sur une organisation la plus décentralisée possible. Par conséquent, ils garantissent le passage à l'échelle de l'Internet. Deux grandes classes des réseaux P2P existent : non-structurés et structurés. Dans les réseaux non-structurés chaque pair est complètement autonome et la propagation des requêtes se fait

par inondation. Les mécanismes sont simples et flexibles mais posent des problémes importants de performances et de passage à l'échelle. Les réseaux structurés organisent les pairs ainsi que la répartition du contenu sur les pairs selon une structure stricte et efficace, notamment une table de hachage distribuée (Distributed Hash Tables ou DHT). Ils gagnent à la fois l'efficacité et la garantie de recherche. Le prix à payer est la perte d'autonomie des pairs et le coût de maintenance élevé.

L'application la plus répandue du P2P est le partage de fichiers. Parmi les applications les plus populaires, on peut distinguer BitTorrent [PGES05] et Gnutella [Gnu05]. Les communautés de partage de fichiers favorisent les réseaux non structurés pour leur grande flexibilité en termes de placement de pairs et de contenu. De plus, la recherche de fichiers par les techniques d'inondation est simple, fiable et surtout flexible dans la manière d'exprimer une requête (plutôt que d'exiger strictement le nom exact du fichier). Cependant, cette inondation est coûteuse en bande passante en raison d'une recherche aveugle et d'une redondance de messages, ce qui menace dangereusement le passage à l'échelle. Pour palier à cela, la technique de index caching permet de créer et distribuer des index de fichiers, de façon simple et pratique. L'idée clé est que les pairs conservent en cache les réponses de requête transitant par leur biais. Les techniques existantes [PH03, Sri01, WXLZ06] limitent la flexibilité de la recherche ou sont inefficaces en termes de stockage et de maintenance d'index. En outre, ces techniques ne prennent pas en compte la proximité physique entre le demandeur et le fournisseur du fichier, alors que les fichiers populaires sont naturellement répliqués en différentes localités. Les conséquences pourraient être désastreuses, notamment en surchargeant le réseau et dégradant les temps de réponses [RFI02].

Aujourd'hui, la collaboration P2P va bien au-delà du simple partage de fichiers. Alors que les usagers du web 2.0 deviennent de plus en plus impliqués, les réseaux P2P ont permis la création de communautés à grandeéchelle pour le partage et la gestion de contenu. Le succés de Wikipedia atteste que la collaboration P2P peut même aboutir à une efficacité qui dépasse de loin celle des systèmes fermés.

Dans le cadre du web 1.0, le contenu des serveurs web est distribué au grand public via les *réseaux de distribution de contenu (CDN)* [BPV08]. Le mécanisme de base consiste à répliquer le contenu populaire sur des serveurs fiables et stratégiquemnt bien placés. Ce mécanisme absorbe la surcharge des web serveurs, limitent les coûts de bande passante et optimisent les temps de réponse. Cependant, un CDN requiert un investissement coûteux qui augmente d'autant plus que le nombre de clients et de requêtes est important.étant donné la forte croissance de la quantité de contenu web et du nombre des clients, les réseaux P2P semblent la solution idéale pour construire des infrastructures peu coûteuses pour la distribution de contenu. C'est parce qu'ils peuvent offrir plusieurs avantages comme la décentralisation, l'auto-organisation, la tolérance aux pannes et le passage à l'échelle. Dans un système P2P, les usagers peuvent servir les requêtes des autres en partageant le contenu précédemment demandé.

Toutefois, le caractère décentralisé et ouvert des réseaux P2P complique l'exploitation des avantages P2P. De nombreux défis doivent être surmontés lors de la construction d'une infrastructure P2P qui est aussi *scalable*, *robuste* et *performante* qu'un CDN.

Un enjeu majeur est de remédier au décalage entre le réseau P2P et le réseau physique sous-jacent dont résultent deux problèmes. Le premier est la surexploitation des ressouces réseau qui limite le passage à l'echelle [RFI02]. Le deuxième est la détérioration de la performance en augmentant les temps de réponse. Pour une collaboration efficace et une bonne qualité de service, les utilisateurs doivent accéder au contenu proche physiquement et communiquer avec des pairsà proximité. Pour ce faire, le réseau P2P a besoin d'intégrer une *locality-awareness* qui se réfèreà des informations sur l'emplacement physique de pairs et de contenu.
Une autre problématique réside dans le fait que les pairs sont autonomes, dynamiques et volatiles [SR06]. Dans ces conditions, il est difficile de garantir la fiabilité du sysème et la disponibilité du contenu. Par ailleurs, le passage à l'échelle requiert l'équilibrage de charge entre les pairs qui prend en compte leurs propres intérêts. Dans la littérature P2P, plusieurs approches comme [IRD02, WNO+02, RY05, FFM04] proposent un P2P CDN. Elles sont généralement confrontés à des compromis entre autonomie et fiabilité, entre qualité de service et coût de maintenance [DGMY02]. En outre, elles n'abordent pas le problème de passage à l'échelle.

L'ojectif de nos travaux de thèse est de construire une infrastructure P2P pour la distribution de contenu. Notre infrastructure ne doit dépendre d'aucune administration centralisée ni de serveurs dédiés. Elle doit exploiter les ressources des pairs et mettre en valeurs leurs intérêts.

Le résumé étendu de cette thèse est organisé en respectant l'organisation en chapitres de la thèse. Ainsi, chaque section du résumé correspond à un chapitre majeur de la thèse. Dans la Section 2, nous décrivons brièvement notre solution pour les systèmes P2P de partage de fichiers. La Section 3 résume Flower-CDN, notre infrastructure de P2P CDN. Dans la Section 4, nous expliquons notre approche pour assurer la robustesse et le passage à l'échelle du P2P CDN. La Section 5 décrit comment nous comptons déployer Flower-CDN pour le public. Enfin dans la Section 6, nous présentons un résumé de nos contributions principales, et proposons quelques directions de recherche futures.

A.2 Partage de fichiers avec Locaware

Dans cette section, nous présentons *Locaware*, la solution d'indexation et de routage pour le partage de fichiers en P2P. Locaware fournit un support efficace pour les requêtes par mots-clés par l'intermédiare des filtres de Bloom. D'autre part, il exploite les localités physiques dans sa technique d'indexage.

A.2.1 Problématique étudiée

Le contexte de notre travail est les réseaux P2P non structurés comme Gnutella [Gnu05]. Les pairs sont dynamiques et autonomes, pouvant entrer ou quitter le réseau à tout moment. Ils partagent des fichiers de tout type spécifié par l'application. Pour rechercher

un fichier, un utilisateur exprime sa requête par une chaîne de mots-clés extraits du nom du fichier cible. La requête est ensuite inondée sur le réseau P2P. Les pairs répondent à la requête, avec des fichiers dont le nom comporte tous les mots-clés de la requête. Une réponse à la requête contient le nom et l'adresse IP d'un pair fournissant le fichier demandé. Les réponses aux requêtes suivent le chemin inverse des requête. Le pair ayant généré la requête télécharge le fichier via une connexion directe avec le pair fournisseur et finit par devenir un fournisseur du fichier en question.

Notre objectif est d'exploiter pleinement les avantages de la mise en cache d'index afin de limiter la consommation de bande passante. Les critères qui ont orienté nos choix sont les suivants.

- **Flexibilité :** la technique d'indexation devrait être couplée à une technique de routage qui dirige les requêtes efficacement vers les index pertinents. Cette technique devrait permettre les recherches par mots-clés qui sont très courantes dans les systèmes de partage de fichiers.
- **Locality-awareness :** l'approche doit intégrer les localités afin d'optimiser le transfert de fichiers. Les index ne doivent pas dirigent les requêtes de façon aléatoire alors que les fichiers concernés sont disponibles dans des pairs à proximité.
- **Disponibilité :** l'appoche devrait tirer profit de la réplication naturelle des fichiers afin de fournir des réponses aux requêtes avec de plus fortes garanties sur la disponibilité du fichier.

A.2.2 Support pour les requêtes mots-clés

Chaque pair maintient un filtre de Bloom qui représente l'ensemble des mots clés de tous les noms de fichiers en cache. Chaque fois que le pair met en cache l'index d'un fichier, il insère tous les mot-clé du nom du fichier dans son filtre de Bloom. Le pair réplique son filtre de Bloom et envoie une copie à chacun de ses voisins directs. Ainsi, les pairs peuvent interroger les filtres de Bloom reçu de leurs voisins pour faire le routage d'une requête.

A.2.3 Indexage basé sur les localités

L'espace des pairs est divisé en plusieurs groupes logiques, chacun ayant un identifiant noté *Gid*. Chaque pair est attribué un groupe et donc un Gid aléatoire. Un pair met en cache toute réponse de fichier dont le nom valide le Gid du pair via une fonction de hachage.

L'espace des pairs est divisé en plusieurs localités physiques, chacune ayant un identifiant noté *locId*. Chaque pair détecte sa localité via des mesures de RTT. Un pair peut détenir pour un fichier donné les adresses de plusieurs pairs fournisseurs et leurs locId. Pour cela, une réponse à la requête doit contenir à la fois l'adresse et le locId du fournisseur. En outre, elle contient l'adresse et le locId de l'initiateur de la requête qui sera

aussitôt considéré comme un nouveau fournisseur du fichier. Sur le chemin de retour de la réponse, les pair mettent en cache la réponse si le nom du fichier valide leur Gid. Dans ce cas, ils associent à l'index de ce fichier toutes les adresses de fournisseurs contenues dans la réponse, y compris celle de l'initiateur de la requête. Ceci est illustré dans la figure A.1 où les pairs appartiennent à trois groupes G1, G2 and G3. P2 demande le fichier F dont le nom valide G1; sa requête atteint un pair de G1 ayant un index de F qui référence le fournisseur P1. Ce dernier génère une réponse de la forme de $< F, P1, locId = 1 >$ et met en cache un nouvel index pour F qui référence le fournisseur éventuel P2. Ensuite les pairs de G1 qui transmettent la réponse stockent deux index pour F, un de P1 and un autre de P2.

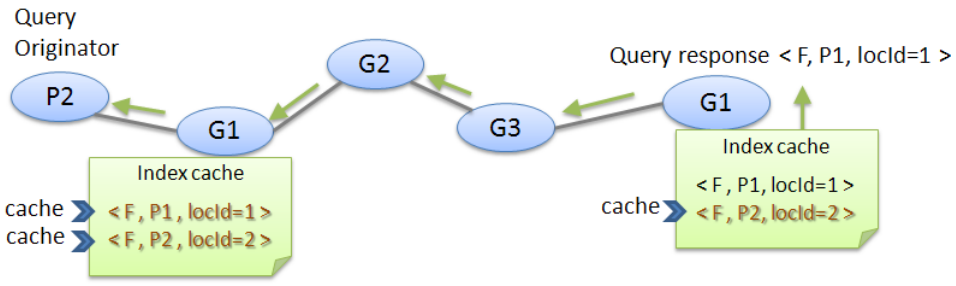


Figure A.1: Locaware : Mise en cache d'index du fichier F dont le nom valide G1.

A.2.4 Routage de requêtes

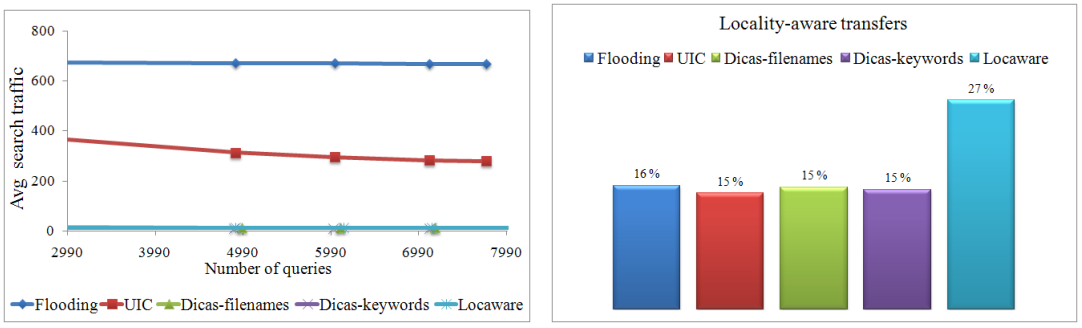
Locaware adapte le mécanisme de routage de requêtes afin de bénéficier pleinement des index mis en cache. La requête est propagée avec les informations d'adresse de l'initiateur de la requête q noté $p_(q)$. Tout pair qui reçoit la requête recherche d'abord dans son cache l'index d'un fichier F qui peut satisfaire q; il collecte l'ensemble d'index liés à F (noté $I(F)$) et en sélectionne ceux qui font référence à des fournisseurs de même $locId$ que l'initiateur de requête. En conséquence, la réponse de la requête peut contenir deux ensembles d'index liés à F, celui qui est conforme au $locId$ de l'initiateur de requête et un autre qui se compose de $locId$s aléatoires pour des garanties de disponibilité. Au cas où le pair n'a trouvé aucun indice qui puisse satify la requête ($I(F)$ est vide), la requête est transmise à certains de ses voisins directs. Il vérifie, pour chaque voisin, si la requête valide son filtre de Bloom et redirige en conséquence la requête.

A.2.5 Évaluation de performances

Nous avons réalisé des simulations avec PeerSim, pour comparer les performances de Locaware avec deux autres algorithmes de base. Nous avons mesuré l'efficacité de la

recherche de fichiers de deux points de vue : satisfaction de l'utilisateur et coût de la recherche.

L'expérience illustrée par la figure A.2a, évalue le coût de recherche en terme de trafic de messages générés. Locaware reduit le trafic de recherche par 98% par rapport aux techniques d'inondation. Donc, Locaware présèrve le principal objectif de mise en cache d'index sélective en limitant drastiquement les surcharges de recherche. Cette réalisation est vitale pour le passage à l'échelle des systèmes P2P de partage de fichiers. L'expérience qui met en relief l'aspect locality-awareness de Locaware mesure la distance



(a) Évolution du trafic de recherche.

(b) Répartition des transferts de fichiers locality-aware.

Figure A.2: Évaluation des performances de Locaware.

de transfert. C'est la disance réseau en terme de latence qui sépare le pair demandeur du pair fournisseur et qui sera traversée par le fichier lors du transfert. La figure A.2b montre le pourcentage de requêtes qui sont servies de la même localité des demandeurs ou le pourcentage de trsnferts de fichiers qui sont locality-aware. Locaware réalise 27% de tranferts locality-aware, une amélioration de 40% par rapport aux autres approches.

La distance de transfert a un impact considérable sur la performance et le passage à l'échelle, surtout que nous parlons de transferts de fichiers volumineux. Des distances courtes limitent le nombre de liens et noeuds physiques intermédiaires qui portent le fardeau de données volumineux. Cela peut réduire considérablement les surcoûts du réseau ainsi que le temps de réponse perçu par l'utilisateur.

Du point de vue satisfaction de l'utilisateur, nous avons comparé le taux de hit de Flower-CDN par rapport à d'autres solutions de mise en cache séléctive. Les résultats montrent que Locaware améliore le taux de hit de manière significative .

A.3 Infrastructure P2P CDN avec Flower-CDN

Le modèle P2P semble être la solution idéale pour construire un CDN efficace et scalable à faibles coûts. Cependant, cette tâche se révèle très difficile à cause du caractère autonome des pairs participants. Dans cette section, nous décrivons *Flower-CDN*, un P2P CDN qui exploite et prend en considération les localités et les intérêts des pairs.

A.3.1 Survol de Flower-CDN

Flower-CDN a pour objectif de distribuer le contenu d'un ensemble W de sites web ws par l'intermédiare des clients intéressés par le contenu de ws. Nous supposons l'existence de k localités physiques : $1 \leq loc \leq k$. La figure A.3 montre un exemple d'architecture de Flower-CDN.

Les participants appartenant à la même localité loc et intéressés par le même site ws construisent ensemble un overlay non structuré noté $petal(ws, loc)$, en utilisant les protocoles gossip. Ces pairs, appelés content peers et notés $c_{ws,loc}$ fournissent du contenus de ws, permettant ainsi d'alléger le serveur ws. Flower-CDN charge un pair de chaque $petal(ws, loc)$, le rôle d'un directory peer (noté $d_{ws,loc}$) : $d_{ws,loc}$ connaît l'ensemble des pairs $c_{ws,loc}$ etindexe leur contenu stocké. Les directory peers font aussi partie de D-ring, un overlay structuré basé sur une DHT, afin de traiter les requêtes des nouveaux clients. Donc Flower-CDN repose sur une architecture hybride composée d'un ensemble de pétales indépendantes reliées par D-ring.

Au lieu d'interroger le serveur ws, un nouveau client situé dans loc, soumet sa requête à D-ring et est dirigé vers le directory peer en charge de ws dans loc , soit $d_{ws,loc}$. Ensuite, $d_{ws,loc}$ tente de résoudre la requête en se basant sur sa pétale ou certains pétales voisines liées à ws. La requête est donc dirigée vers un content peer $c_{ws,loc}$ qui détient l'objet demandé; $c_{ws,loc}$ sert la requête en transfèrant l'objet au client. Ensuite, le client peut intégrer $petal(ws, loc)$ comme un content peer $c_{ws,loc}$. Pour les requêtes suivantes, $c_{ws,loc}$ recherche directement dans sa pétale au lieu de D-ring.

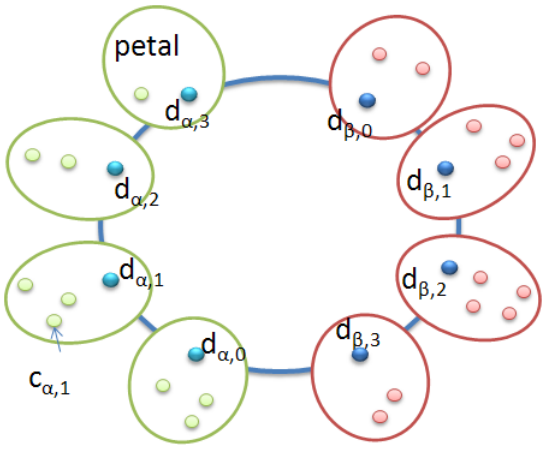


Figure A.3: Architecture de Flower-CDN avec 2 websites α & β et 4 localities.

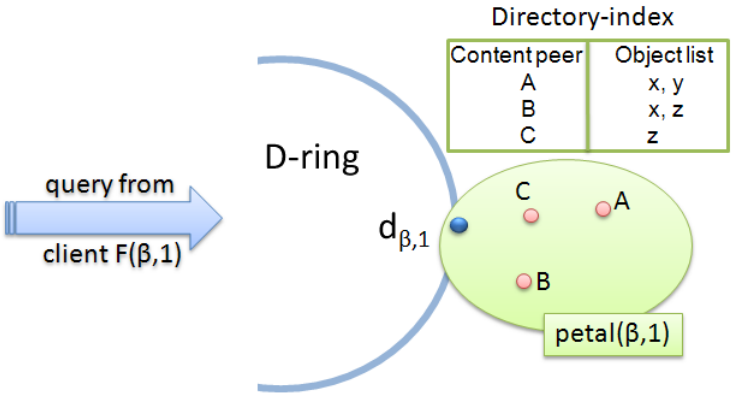


Figure A.4: Une requête pour F soumise par un nouveau client de β de localité $loc = 1$.

A.3.2 D-ring

D-ring est un overlay structuré qui repose sur une DHT, tout en tenant compte des intérêts et des localités des pairs pour construire l'overlay et traiter les requêtes. Pour chaque site ws, D-ring alloue k directory peers, k étant le nombre de localités. Chaque directory peer $d_{ws,loc}$ est attribué un ID en focntion du le site et la localité qu'il représente. Basé sur le service de routage de la DHT, D-ring livre toute requête visant le site ws et la localité loc à $d_{ws,loc}$.

D-ring agit comme un service d'annuaire P2P pour les clients souhaitant utiliser et contribuer à Flower-CDN. Principalement, il offre deux fonctionnalités. Premièrement, il soutient les premières requêtes en provenance de nouveaux clients et les gère au lieu des serveurs web d'origine. Deuxièmement, D-ring permet un accès fiable à Flower-CDN pour les nouveaux participants : en routant sa première requête de D-Ring, un client est guidé vers la pétale liée à sa localité loc et son site d'intérêt ws. Ainsi, il peut faire partie de Flower-CDN en tant que content peer ou directory peer.

La figure A.4 montre une partie de D-ring avec le directory peer $d_{\beta,1}$ et trois content peers pour $(\beta, 1)$, A, B et C. $d_{\beta,1}$ maintient un *directory-index*$(\beta, 1)$ qui liste pour chaque pair dans $petal(\beta, 1)$, leurs objets (A stocke les objets x et y qui sont fournis par le site β). De plus, $d_{\beta,1}$ stocke des résumés des directory-index de ses voisins directs, $d_{\beta,0}$ and $d_{\beta,2}$.

Le nouveau client F du site β arrive avec une requête q pour l'objet x. En supposant que F se trouve dans la localité $loc = 1$, q est transmise à $d_{\beta,1}$ qui recherche x dans son directory-index. Ensuite, $d_{\beta,1}$ dirige q au content peer A ou C, qui ont une copie de x et peuvent donc satisfaire la requête. Si F demande un objet tl que x' ne se trouvant pas dans $petal(\beta, 1)$, $d_{\beta,1}$ vérifie ses résumés de directory-index pour $(\beta, 0)$ et $(\beta, 2)$ et en conséquence $d_{\beta,1}$ envoie q à $d_{\beta,0}$ ou $d_{\beta,2}$. En dernier recours, $d_{\beta,1}$ redirige q au site β.

A.3.3 Les Pétales

Une pétale $petal(ws, loc)$ comporte le directory peer $d_{ws,loc}$ et plusieurs content peers $c_{ws,loc}$. Elle se développe au fur et à mesure que des clients de ws en loc rejoignent Flower-CDN.

Chaque $petal(ws, loc)$ fournit une infrastructure de recherche pour les requêtes des content peers $c_{ws,loc}$. Dès qu'un client devient $c_{ws,loc}$, il résoud ses requêtes en se basant sur sa pétale au lieu de D-ring. A cet effet, dans la pétale, les content peers découvrent le contenu stocké par d'autres pairs $c_{ws,loc}$ en échangeant des résumés de leurs contenus. Ainsi, $c_{ws,loc}$ peut fouiller dans les résumés pour voir où une copie de son objet demandé pourrait être stockée.

Les protocoles gossips sont utilisés pour diffuser les résumés et leurs mises à jour de façon épidémique. Les pairs peuvent également découvrir de nouveaux membres dans leur pétale et détecter les pannes. L'algorithme gossip de base est comme suit. Un pair p connaît un groupe de pairs ou *contacts* qui sont maintenus dans une liste appelée *vue*. Périodiquement, p sélectionne un contact q, lui envoie ses informations, et reçoit en contrepartie des informations de q.

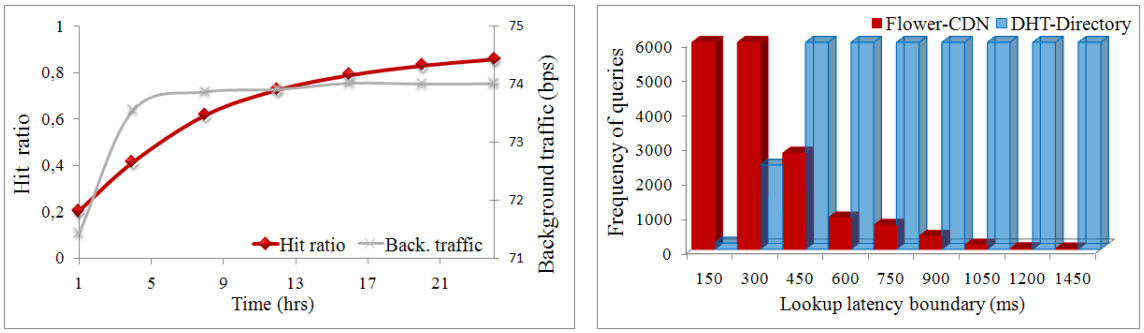
Pour maintenir le directory-index à jour, chaque $c_{ws,loc}$ envoie à $d_{ws,loc}$ des mises à jour sur son contenu en cache via des *push messages*. $c_{ws,loc}$ surveille les changements (les objets nouvellement stockés ou effacés) et chaque fois que le pourcentage de nouveaux changements atteint un seuil prédéfini, $c_{ws,loc}$ envoie la liste des changements à $d_{ws,loc}$.

A.3.4 Évaluation de performances

Nous avons mené une évaluation de performances détaillée via des simulations basées sur PeerSim. Nous comparons Flower-CDN à un autre P2P CDN DHT-Directory. Ci-dessous, nous résumons nos observations les plus significatives.

Premièrement, l'utilisation de gossip engendre des surfrais assez acceptables en termes de bande passante : un algorithme de gossip opère au sein d'une pétale, et donc implique des communications entre des pairs qui sont physiquement proches. En outre, les paramètres de gossip (comme la périodicité) peuvent être ajustés afin d'adapter les surfrais en fonction des ressource disponibles et des exigences en termes de hit ratio. La figure A.5a montre l'évolution du taux de hit en même temps que le trafic de gossip. Le taux de hit ne cesse d'augmenter avec le temps puisque le contenu demandé est répliqué au fur et à mesure que les requêtes sont générées et satisfaites. En contrepartie, le trafic de gossip se stabilise à 74 bps après 5 heures.

Deuxièmement, l'infrastructure de routage de Flower-CDN s'avère très efficace en termes de recherches rapides et pertinentes. Elle repose sur l'architecture hybride qui combine entre overlays structuré et basé sur gossip avec des considérations de localités physiques. D'une part, grâce à la DHT, D-ring permet un lookup efficace et fiable qui garantit que les nouveaux clients trouvent toujours leurs pétale en même temps que leur première requête. Cependant, la taille de D-ring est limitée à un nombre restreint de directory peers, minimisant ainsi le nombre de hops d'une requête. Ceci diminue la charge



(a) Taux de hit et surcharge de bande passante. (b) Latence de lookup.

Figure A.5: Évaluation des performances de Flower-CDN.

de routage sur les pairs de la DHT ainsi que le temps de réponse de la requête. D'autre part, une grande partie des requêtes est routée et résolue au sein des pétales qui sont à base de localités. Ceci implique des recherches plus rapides que dans le cas d'une DHT. En effet, la figure A.5a montre que 87% des requêtes de Flower-CDN sont satisfaites en moins de 150 ms alors que 61% des requêtes de DHT-Directory mettent plus que 1050 ms.

Ces résultats démontrent que l'architecture hybride de Flower-CDN peut pleinement satisfaire les exigences de CDN en termes de performances tout en respectant le caractère autonome des pairs.

A.4 Robustesse et passage à l'échelle de Flower-CDN

Une préoccupation majeure d'un P2P CDN est de gérer la participation dynamique et à large échelle des pairs. Flower-CDN doit être robuste aux pannes et aux effets de churn : ces événements fréquents ne doivent pas perturber son architecture et son fonctionnement. En outre, Flower-CDN doit être en mesure de soutenir de grands nombres de participants sans détérioration de performances ni goulets d'étranglement.

Dans cette section, nous présentons *PetalUp-CDN* pour assurer le passage à l'échelle de Flower-CDN. Pour garantir la robustesse, nous proposons des protocoles de maintenance qui gèrent les connexions et déconnexions des pairs via les protocoles de gossip.

A.4.1 PetalUp-CDN

PetalUp-CDN est une version scalable de Flower-CDN qui s'adapte dynamiquement à des taux de participation de pairs variables. PetalUp-CDN est conçu d'une manière qui permet à plusieurs directory peers de partager la gestion d'une même pétale. Pour

maintenir les aspects de localité et d'intérêt de l'architecture ainsi que ses performances, PetalUp-CDN vise à accomplir certains objectifs :

- adapter l'architecture de D-ring, afin de soutenir plusieurs directory peers par pétale.
- implémenter l'évolution de D-ring de façon dynamique qui n'affecte pas les performances du service d'annuaire P2P.
- adapter la gestion d'une pétale aux changements afin de préserver l'efficacité de la recherche de contenu dans un pétale.

La structure actuelle de D-ring ne permet qu'un seul directory peer par couple (ws, loc). Puisque le problème réside dans le service de gestion de clés de D-ring, PetalUp-CDN adapte ce service afin que D-ring passe à l'échelle. Plusieurs directory peers par couple (ws, loc) vont intégrer D-ring consécutivement. Le nombre de directory peers en charge de $petal(ws, loc)$ augmente progressivement avec le le nombre de clients pour ws dans loc.

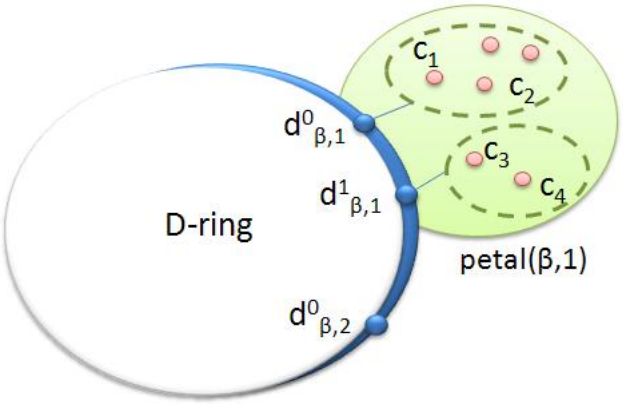


Figure A.6: Exemple de $petal(\beta, 1)$ dans PetalUp-CDN.

En ayant différents directory peers en charge d'une pétale, la défaillance d'un ou plusieurs de ces pairs ne causera pas une perte complète des informations d'indexation, et permettra au système de continuer à fonctionner normalement. Par ailleurs, ces pairs ne maintiennent pas des informations redondantes puisque chacun est responsable des informations d'une partie de la pétale. Un exemple de PetalUp-CDN est illustré dans la figure A.6 qui met en relief $petal(\beta, 1)$. Deux directory peers $d^0_{\beta,1}$ and $d^1_{\beta,1}$ se partagent la gestion de $petal(\beta, 1)$. Ainsi ils gèrent chacun un sous-ensemble de content peers $c_{\beta,1}$.

Les directory peers de $petal(ws, loc)$ sont créés séquentiellement à partir de $d^0_{ws,loc}$. Un nouveau directory peer est créé pour $petal(ws, loc)$ dès que le nombre de content peers $c_{ws,loc}$ n'est plus gérable par les directory peers $d^i_{ws,loc}$ existants. Ceci est détecté par

ces directory peers au fur et à mesure qu'ils traitent de nouvelles requêtes et évaluent le nombre de leurs content peers par rapport à une limite prédéfinie.

La requête d'un nouveau client qui vise $petal(ws, loc)$ parcourt les directory peers existants $d^i_{ws,loc}$ à la recherche d'un directory peer non chargé. S'il n'existe pas encore, le dernier créé $d^i_{ws,loc}$ initie l'arrivée d'un nouveau directory peer $d^{i+1}_{ws,loc}$.

La taille d'une pétale évolue de façon dynamique : elle diminue lorsque les content peers existants se déconnectent et augmente lorsque de nouveaux clients se connectent. Par conséquent, un directory peer surchargé peut se retrouver avec moins de content peers et se met donc à accepter de nouveaux clients. En outre, un site ws peut perdre sa popularité avec le temps, les content peers désertant en permanence ses pétales. Dans ce cas, les directory peers redondants sont éliminés progressivement pour éventuellement se retrouver avec un seul directory peer pour gérer la pétale de taille réduite.

A.4.2 Protocoles de maintenance

Nos protocoles de maintenance font face au comportement dynamique des pairs afin d'éviter toute dégradation de performances ou faille dans le fonctionnement de Flower-CDN.

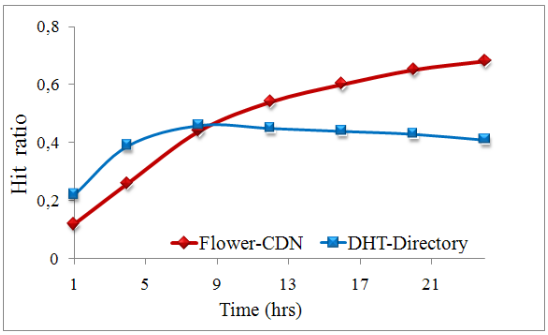
Les mécanismes de Flower-CDN dépendent largement du lien entre D-ring et les pétales. Toutefois, le départ d'un directory peer peut déconnecter sa pétale de D-ring. Pour maintenir le lien, nous nous appuyons sur deux éléments : les **messages push** et *keepalive* d'une part, et l'échange de $dir - info$ d'une autre part. Les content peers d'une pétale peuvent détecter la vivacité de leur directory peer au moyen des messages push et vice versa. Toutefois, ceci n'est pas suffisant parce que certains content peers n'ont pas des changements fréquents dans leurs contenus et donc communiquent rarement avec leurs directory peers au moyen des messages push. Pour cela nous exploitons une caractéristique inhérente aux systèmes P2P, les messages keepalive qui sont échangés périodiquement pour vérifier les liens entre les pairs. Ainsi, $d_{ws,loc}$ supprime de son directory-index les entrées qui ont déjà expiré.

Les effets de churn affectent aussi l'architecture et le fonctionnement de D-ring en l'absence des protocoles de maintenance appropriés. Si un directory peer se déconnecte, ses requêtes seront redirigées vers de faux directory peers et les clients ne seront pas en mesure de rejoindre leurs pétales cibles. En se basant sur gossip, nos protocoles détectent rapidement les directory peers déconnectés et les remplacent efficacement par des content peers de la même pétale. En outre, pour soutenir la construction progressive, D-ring permet aux pairs d'intégrer dynamiquement D-ring sans perturber l'architecture.

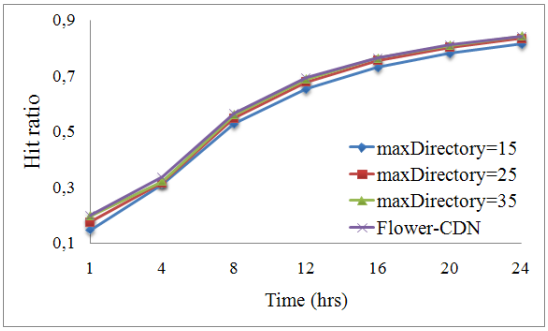
A.4.3 Évaluation de performances

Pour valider nos contributions, nous avons implémenté nos protocoles et simulé un environnement P2P avec un taux de churn très élevé. Nos protocoles de maintenance peuvent garantir un taux de hit très élevé et des temps de réponse réduits. En résumé, le taux de hit est amélioré de 40% (Figure A.7a) et les temps de réponses sont réduits

d'un facteur de 12. De plus, le mécanisme de détection par gossip est assez efficace et ne génère pas des surcoûts significatifs. En outre, le mécanisme de reprise atténue la perte d'informations et permet une transition en douceur. Donc, Flower-CDN et PetalUp peuvent être extrêmement robustes malgré des taux de churn élevés.



(a) Taux de hit entre Flower-CDN et DHT-Directory.



(b) Évolution du taux de hit.

Figure A.7: Évaluation des performances dans un environnement dynamique.

En ce qui concerne le passage à l'échelle, Flower-CDN montre de très bons gains malgré la taille modeste des pétales (maximum 60 pairs). Nous croyons que des pétales plus larges peuvent contribuer à accroître les gains. Pour les échelles supérieures, PetalUp-CDN démontre sa capacité à éviter les situations de surcharge sans entraîner une baisse de performances. Son approche qui tend à partager la gestion d'une pétale n'affecte pas le taux de hit (Figure A.7b) ni les temps de réponse (Figure A.8) lors de la manipulation des requêtes. Les résultats sont très prometteurs, prouvant que notre P2P CDN peut passer à l'échelle.

En examinant l'évolution du taux de hit (Figure A.7b), nous observons que les quatre approches parviennent à des résultats similaires. Ceci démontre que le "partitionnement" d'une pétale n'affecte pas les performances de notre P2P CDN dans le traitement des requêtes. Que l'ensemble des content peers soit géré par un seul directory peer ou réparti sur plusieurs directory peers, le système réussit aussi bien à localiser le contenu demandé et satisfaire les requêtes.

A.5 Déploiment de Flower-CDN

Le déploiment de Flower-CDN est assuré par les clients qui sont intéressés à un site particulier et qui sont disposés à participer afin de profiter d'un meilleur accès au contenu de leur intérêt. Un site web ws est soutenu par Flower-CDN tant qu'il y a un nombre suffisant de clients pour le compte de ws. Plus précisément, plus un site ws est populaire, plus les participants sont attirés par Flower-CDN pour peupler les pétales de ws et occuper ses positions de directory peers. Pour un site non populaire, ses pétales ont tendance à être vides et ses positions de directory peers non occupées.

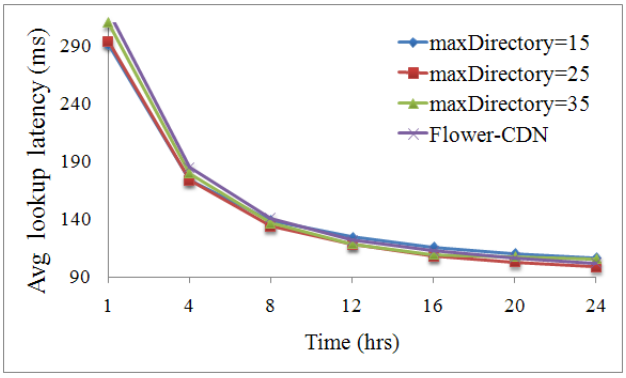


Figure A.8: Latence de lookup de PetalUp-CDN.

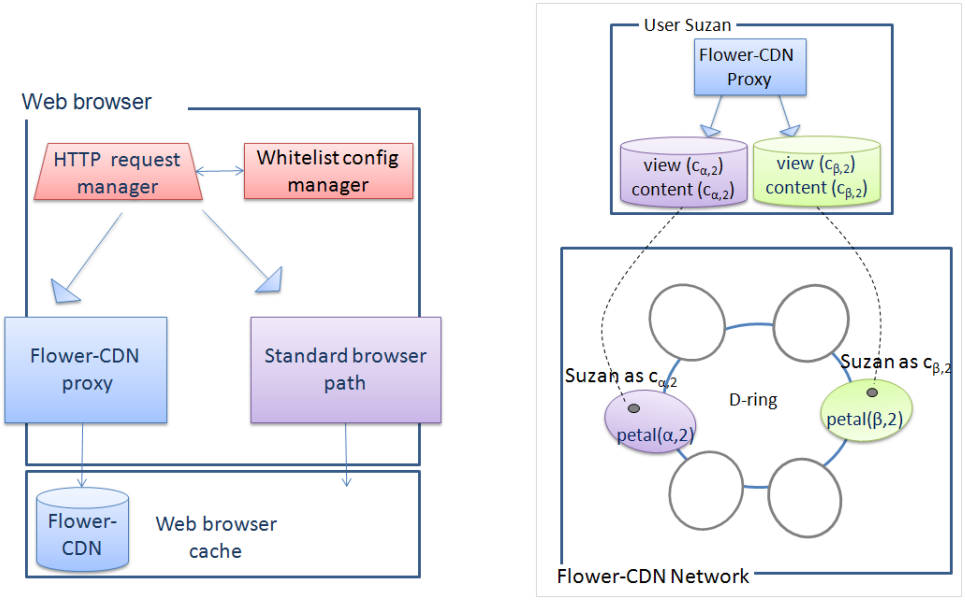
Un utilisateur accède au web via son navigateur web qui prend en charge ses demandes HTTP et permet en conséquence la recherche et la visualisation du contenu web. Afin d'utiliser et de contribuer à Flower-CDN d'une manière transparente, l'utilisateur doit incorporer la fonctionnalité de Flower-CDN dans son navigateur.

Nous proposons une extension de navigateur pour Flower-CDN qui permet la distribution de contenu en P2P en toute transparence et flexiblilité. En outre, la configuration de cette extension permet aux utilisateurs de gérer dynamiquement leurs intérêts et leurs préférences en matière de confidentialité, en précisant le contenu qu'ils veulent partager. Enfin, nous adoptons un modèle de sécurité simple qui garantit l'intégrité du contenu, même face à des pairs non fiables.

Dans cette section, nous décrivons l'extension proposée pour intégrer la fonctionalité de Flower-CDN à un navigateur web. Puis, nous discutons brièvement de la configuration de l'extension et concluons avec les questions de sécurité.

A.5.1 Extension pour le navigateur

La figure A.9a montre que Flower-CDN opère dans un navigateur via trois composants principaux : un *whitelist config manager*, un *HTTP request manager* et un *Flower-CDN proxy*. Le *whitelist config manager* maintient une liste configurée par l'utilisateur, la *whitelist* de Flower-CDN, qui spécifie un ensemble de domaines se référant à des sites web pour lesquels l'utilisateur participe à Flower-CDN. Le processus de navigation sur le web est déclenché lorsque l'utilisateur saisit une URL dans le navigateur et lance une requête HTTP. Cette requête est d'abord traitée par le *HTTP request manager* qui la transmet au *Flower-CDN proxy* si l'URL correspond à la whitelist. Sinon, la requête HTTP suit le processus de traitement standard du navigateur. À la réception de la requête, le *Flower-CDN proxy* essaie d'abord de la résoudre localement avant d'avoir recours au réseau Flower-CDN. L'utilisateur est connecté à ce réseau en tant que content peer ou directory peer, via son proxy local qui communique avec d'autres Flower-CDN proxies hébergés par des utilisateurs distants. Ainsi, un Flower-CDN proxy gère les demandes en provenance des utilisateurs distants, en plus des demandes de l'utilisateur local.



(a) Intégration dans un navigateur Web. (b) Un utilisateur en deux content peers.

Figure A.9: L'extension Flower-CDN pour un navigateur web.

A.5.2 Configuration

Le contenu que l'utilisateur partage en Flower-CDN est stocké dans une section délimitée du cache du navigateur. Cela garantit la confidentialité de l'utilisateur, car il permet d'isoler le contenu privé de l'utilisateur du contenu web qu'il veut partager. Limitée par l'espace disque disponible pour le cache du navigateur, la quantité d'espace disque alloué à Flower-CDN croît dynamiquement avec la quantité de contenu mis en cache. Les politiques d'expiration adoptées par le cache du navigateur sont également utilisées pour gérer le contenu de Flower-CDN. En outre, toutes les informations de vue et de directory-index d'un pair sont stockées dans cette section de Flower-CDN et gérées en fonction des échanges gossip et keepalive.

Un utilisateur peut s'intéresser à n sites différents pour lequels il veut utiliser Flower-CDN. Or, dans Flower-CDN, les pairs qui sont liés à des sites différents sont impliqués dans des pétales différentes et donc leurs comportements sont aucunement corrélés. Par conséquent, l'utilisateur peut participer à Flower-CDN en tant n pairs différents. Il précise, par l'intermédiaire du *whitelist config manager*, le nom des n sites de son intérêt et la section de cache réservée à Flower-CDN est partitionnée en n sous-sections de taille dynamique.

La figure A.9b montre comment l'utilisatrice Suzan est représentée dans le réseau Flower-CDN. Suzan est dans la localité 2 et intéressée par deux sites α and β. Donc, elle est représentée par deux content peers $c_{\alpha,2}$ et $c_{\beta,2}$. Le Flower-CDN proxy qui opère

au sein du navigateur de Susan, gère deux sous-sections de cache différéntes, une pour chaque content peer.

A.5.3 Sécurité

Dans un environnement P2P ouvert, des pairs peuvent être malveillants et tenter de corrompre le contenu partagé. Ce problème peut être facilement résolu si les serveurs web fournissent des certificats signés numériquement avec leur contenu. Le Flower-CDN proxy local de l'utilisateur peut, à l'aide de la clé publique du site web, vérifier la signature numérique d'un objet lié à ce site et en provenance d'un certain content peer distant. Cette solution n'est nullement affectée par la dynamicité des pairs et s'accommode très bien avec un environnement non fiable.

A.6 Conclusion

Dans cette section, nous résumons nos principales contributions. Ensuite, nous donnons un aperçu de la façon dont nous envisageons poursuivre ce travail de thèse.

A.6.1 Contributions principales

Cette thèse a abordé la distribution de contenu dans les systèmes P2P. Le travail aété réalisé en quatreétapes.

D'abord, nous avons entrepris une étude globale et approfondie des systèmes P2P et de la distribution de contenu, afin de motiver nos prochaines contributions et de souligner les lacunes que nous devons aborder. Les principales observations que nous avons faites peuvent être résumés comme suit. Premièrement, les CDNs ont des exigences très strictes en matière de performances, passage à l'échelle et fiabilité. Pour garantir de hautes performance à large échelle, il faut inévitablement plus d'investissement en matière de serveurs dédiés et donc coûteux. La seconde observation est que les systèmes P2P introduisent des exigences fondamentales telles que l'autonomie, l'expressivité, l'efficacité, la qualité de service et la robustesse. De plus, nous observons l'émergence de nouvelles tendances qui visent à raffiner le réseau P2P pour davantage de performance et d'efficacité. Parmi ces tendances, nous nous sommes intéressés aux shémas basés sur la proximité physique ou sémantique, à l'emploi des protocoles de gossip et la combinaison de réseaux P2P. Les défis sont de garder les solutions simples, d'éviter une gestion centralisée ou coûteuse, d'opérer rapidement et de s'adapter aux changements dynamiques. Notre troisième observation se rapporte au partage de fichiers P2P. Néanmoins, ces systèmes devraient obligatoirement limiter la charge réseau. La priorité est d'exploiter les localités physiques pour un transfet de fichier optimisé. Toutefois, la majorité des travaux existant négligent cette priorité. Notre dernière observation concerne les P2P CDNs. Ils doivent respecter les exigences P2P et CDN mais en réalité sacrifient l'une pour l'autre. Surtout, ils n'abordent pas le passage à l'échelle.

La deuxième contribution s'est focalisée sur le partage de fichiers en P2P, en vue d'une infrastructure basique pour la distribution de contenu [DPV07, DP09]. L'infrastructure repose sur un réseau overlay non structuré puisque ce type d'overlay est largement déployé dans le cadre de partage de fichiers en raison de sa flexibilité. Nous avons abordé le problème de consommation de bande passante sous deux angles : l'inefficacité de la recherche et les transferts de fichiers longue distance. Notre solution, Locaware, tire profit des propriétés intrinsèques du partage de fichiers P2P: la réplication naturelle des fichiers, les localités temporelle et physique des requêtes. Locaware consiste à mettre en cache des index de fichiers avec des informations sur leurs localités. Elle fournit également un support efficace pour les requêtes par mots clés qui sont courantes dans ce genre d'applications.

La troisième contribution a visé une infrastructure élaborée qui puisse remplacer les CDNs commerciaux de par ses garanties en termes de performance et d'efficacité. Flower-CDN permet à tout site populaire et sous-provisionné de distribuer son contenu, par l'intermédiaire de sa communauté d'utilisateurs. Pour un routage efficace, l'infrastructure Flower-CDN combine intelligemment différents types d'overlays avec des protocoles épidémiques tout en exploitant les intérêts et les localités des pairs. Elle fournit un accès fiable aux nouveaux participants par l'intermediaire de D-ring et leur permet de s'organiser en pétales en fonction de leur localités et leurs intérêts. Les pétales assurent des recherches rapides qui permettent au demandeur de localiser le contenu disponible à proximité pour un transfert efficace.

La quatrième contribution avait pour objectif d'assurer le passage à l'échelle de Flower-CDN. Pour cela, nous avons proposé PetalUp-CDN qui permet à Flower-CDN de supporter des taux de participation considérables, et même variables. Pour éviter les situations de surcharge, D-ring évolue de manière dynamique par rapport aux besoins des pétales, tout en préservant l'aspect orienté localité et intérêt de l'architecture et en maintenant les performances. Nous nous sommes également intéressés à la fiabilité de Flower-CDN et avons conçu des protocoles de maintenance qui gèrent la dynamicité des pairs et offrent un P2P CDN robuste.

Finalement, nous avons adressé le déploiement de Flower-CDN pour permettre à tout client de participer au nom des web sites qui l'intéressent. Pour une participation flexible et transparente, nous avons choisi d'implémenter la fonctionalité de Flower-CDN via une extension qui peut être intégrée au navigateur web du client.

A.6.2 Travaux futurs

Nos travaux futurs visent à enrichir Flower-CDN avec des fonctionnalités plus avancées. Nous présentons ci-suit une liste non exhaustive des travaux que nous envisageons de réaliser.

1. **Expérimentation avec traces réelles :** Nous sommes entrain de raffiner nos

expérimentations par l'injection de traces web réelles. Ainsi les simulations décriront le comportement de l'utilisateur et ses variantes d'une manière plus fiable et réaliste.

2. **Exension Flower-CDN pour les navigateurs :** Un autre travail en cours est de finaliser la mise en oeuvre de l'extension Flower-CDN pour un navigateur. Nous espérons permettre aux utilisateurs intéressés de télécharger l'extension et d'utiliser Flower-CDN.

3. **Intérêts et recherche sémantiques :** Nous avons l'intention de permettre aux utilisateurs d'exprimer leurs intérêts d'une manière plus complexe. Un intérêt refléterait les préférences sémantiques par le biais d'ontologies, de combinaison de thèmes, etc. En conséquence, les participants seraient en mesure d'effectuer des recherches sémantiques où les requêtes pourraient être exprimées avec des mots clés ou autres informations sémantiques plutôt que des stricts URL. Ceci permetterait aux utilisateurs de naviguer le contenu disponible et découvrir les objets qui pourraient les intéresser.

4. **Les sites sociaux de contenu :** Le Web 2.0 a favorisé l'intégration de contenu avec les informations d'ordre social des utilisateurs, ce qui a donné lieu aux sites sociaux de contenu. Des sites qui étaient initialement orientés vers le contenu (par exemple, Youtube, Yahoo Voyage) ou vers les réseaux sociaux (par exemple, Facebook, MySpace) évoluent rapidement vers une telle intégration. Toutefois, ces sites reposent sur des architectures centralisées ou dédiées, et donc peuvent souffrir de problèmes de disponibilité, de confidentialité, et de passage à l'échelle. Pour pallier ces problèmes, nous envisageons le déploiement de ces sites sur l'infrastructure décentralisée de Flower-CDN. Évidemment, une telle perspective requiert une étude approfondie des changements que devrait subir Flower-CDN, comme la notion d'intérêt, les techniques de recherche, etc.

5. **Extension du routage IP :** Enfin, nous envisageons approfondir notre étude sur les localités physiques et raffiner le routage dans Flower-CDN. Le protocole de routage IP s'est avéré extrêmement scalable, soutenant des millions de noeuds et gérant leur volatilité. Ainsi, une direction de recherche prometteuse consisteà étendre le routage IP avec la sémantique des données, en vue d'un meilleur soutien EUR comme des requêtesà Flower-CDN. Les extensions devraient envisager distribués données techniques de gestion.