Building a Peer-to-Peer Content Distribution Network with High Performance, Scalability and Robustness

Manal El Dick, Esther Pacitti, Reza Akbarinia, Bettina Kemme

Building a Peer-to-Peer Content Distribution Network with High Performance, Scalability and Robustness

Manal El Dick[a,*], Esther Pacitti[b], Reza Akbarinia[c], Bettina Kemme[d]

[a] *INRIA & LINA, University of Nantes*
[b] *INRIA & LIRMM, University of Montpellier 2*
[c] *INRIA & LINA, Nantes*
[d] *McGill University, Montreal, Canada*

Abstract

Content Distribution Networks (CDN) are fundamental, yet expensive technologies for distributing the content of web-servers to large audiences. The P2P model is a perfect match to build a low-cost and scalable CDN infrastructure for popular websites by exploiting the underutilized resources of their user communities. However, building a P2P-based CDN is not a straightforward endeavor. In contrast to traditional CDNs, peers are autonomous and volunteer participants with their own heterogeneous interests that should be taken into account in the design of the P2P system. Moreover, churn rate is much higher than in dedicated CDN infrastructures, which can easily destabilize the system and severely degrade the performance. Finally and foremostly, while many P2P systems abstract any topological information about the underlying network, a top priority of a CDN is to incorporate locality-awareness in query routing in order to locate close-by content. This paper aims at building a P2P CDN with high performance, scalability and robustness. Our proposed protocols combine DHT efficiency with gossip robustness and take into account the interests and localities of peers. In short, *Flower-CDN* provides a hybrid and locality-aware routing infrastructure for user queries. *PetalUp-CDN* is a highly scalable version of Flower-CDN that dynamically adapts to variable rates of participation and prevent overload situations. In addition, we ensure the robustness of our P2P CDN via low-cost maintenance protocols that can detect and recover from churn and dynamicity. Our extensive performance evaluation shows that our protocols yield high performance gains under both static and highly dynamic environments. Furthermore, they incur acceptable and tunable overhead. Finally we provide main guidelines to deploy Flower-CDN for the public use.

Keywords: P2P, CDN, locality-awareness, interest-awareness, robustness, scalability

1. Introduction

Content Distribution Networks (CDN) such as Akamai 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 without the large costs of traditional CDNs.

In this paper, we propose such a scalable and cheap CDN based on the principles of Peer-to-Peer (P2P) technology. The last decade has witnessed a paradigm shift in the design of internet-scale distributed systems, with widespread proliferation of the P2P model for a wide range of applications. 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 naturally offers scalability: as more peers join the system, they contribute to the aggregate resources of the P2P network. We believe that the P2P model is a perfect match to build a CDN infrastructure 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

*Corresponding author. Tel: +33670433179; Fax:.
Email addresses: manal.el-dick@univ-nantes.fr (Manal El Dick), esther.pacitti@univ-nantes.fr (Esther Pacitti), reza.akbarinia@univ-nantes.fr (Reza Akbarinia), kemme@cs.mcgill.ca (Bettina Kemme)

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 quire different to what is expected from a file-sharing system. Any CDN has to focus on two performance metrics: *response time* and *hit ratio*. A traditional CDN replicates 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 CDNs 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. In contrast to traditional CDNs, peers are not dedicated servers but autonomous and volunteer participants with their own heterogenous interests. Thus, they should not be forced to store web-pages they are not interested in but should only serve content they are willing to. Additionally, churn rate is much higher than in dedicated CDN infrastructures. In fact, the participation of peers is highly dynamic, implying thousands of continuous joins and leaves, which creates the effect of churn. This may destabilize the system and severely degrade the performance in the absence of efficient detection and recovery protocols. Furthermore, while many P2P systems abstract any topological information about the underlying network, we have to make locality-awareness a top priority in order to achieve short query response times.

Our solutions exhibit several unique characteristics that enable us to overcome all of the above mentioned challenges.

- **Flower-CDN** is a P2P CDN that enables any website to efficiently distribute its content, with the help of the non-profit community interested in its content. 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 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 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 Flower-CDN and PetalUp-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.

- We present both an analytical and an extensive simulation-based performance evaluation. It shows that Flower-CDN yields high performance gains under both static and highly dynamic environments. Furthermore, Flower-CDN incurs acceptable overhead, which can be tuned according to hit ratio requirements and bandwidth availability.

- We provide guidelines to deploy Flower-CDN for public use. We propose to implement Flower-CDN functionality as an extension of the user's web browser and cover security and privacy issues in a simple and practical manner. As such, the user enjoys a transparent, flexible and highly configurable experience with Flower-CDN.

In our previous work [1, 2], we introduced Flower-CDN and its scalable version PetalUp-CDN. This paper further refines and develops our initial proposals. In particular, we elaborate here on the algorithms of PetalUp-CDN, mainly with respect to the expansion of D-ring and its shrink. Further, we generalize and extend the maintenance protocols to cover all possible scenarios and dynamically adapt D-ring and the petals. This paper also deepens the performance analysis under churn and investigates the signalling overhead both analytically and empirically. Moreover it provides the first performance evaluation of PetalUp-CDN through extensive simulation and discussion. The final contribution of this paper addresses the architectural changes that are required to deploy Flower-CDN for public use.

Roadmap:. Section 2 provides a detailed presentation of Flower-CDN. PetalUp-CDN is described in Section 3. Section 4 discusses the maintenance protocols that ensure the robustness of Flower-CDN and PetalUP-CDN under churn. A cost analysis that focuses on the gossip overhead of our approach is given in Section 5. Simulation methodology and results are presented in Section 6. Section 7 gives some guidelines about the deployment of Flower-CDN for public use. Section 7 describes some related work before concluding in Section 9.

2. Flower-CDN

Flower-CDN is a P2P CDN that strictly relies on interested clients rather than dedicated and expensive servers. In this section, we first give an overview of Flower-CDN, then we explore the models of D-ring and the petals.

2.1. 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 webpages 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 to the content of ws.

We implement locality-awareness in Flower-CDN via the binning technique [3]. 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 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.

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.

2.2. 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 D-ring which consists of a P2P directory service.

2.2.1. 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 [4], Pastry [5]). For each website $ws \in W$, the directory overlay enables k participant peers from P_{ws}, where k is

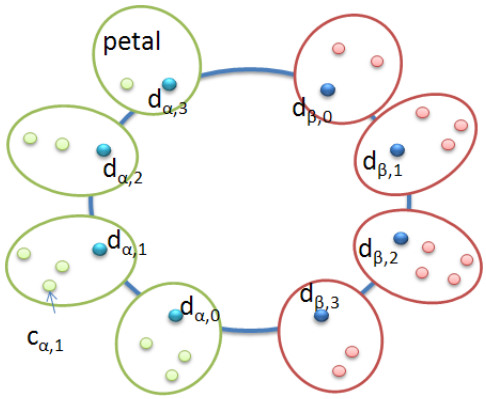

Figure 1: Flower-CDN architecture with websites α and β and four localities.

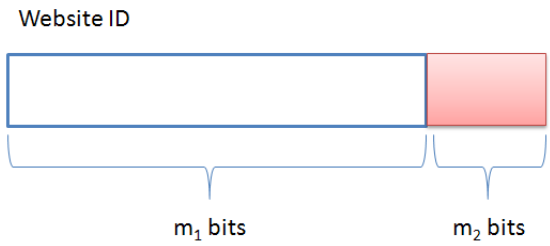

Figure 2: Peer ID structure in D-ring.

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 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 DHT-based systems, peer identifiers (noted ID) are chosen from an identifier space $S = [1 \cdot \cdot 2^m - 1]$; 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 2, the m bits of a peer ID are split into 2 segments, a *website ID* and a *locality ID*:

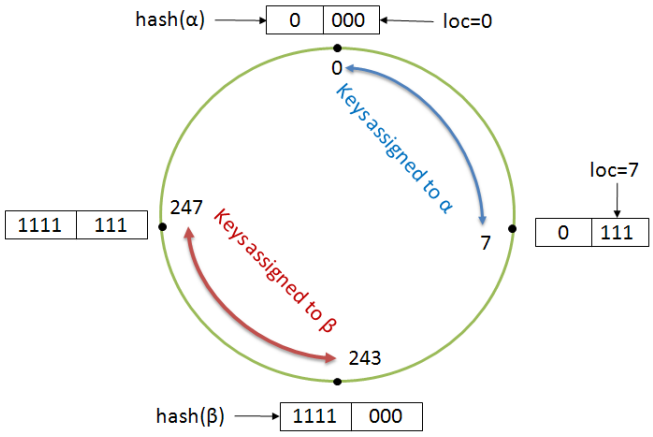

Figure 3: D-ring distribution of keys given that $k = 8$ and $W = \{\alpha, \beta\}$.

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

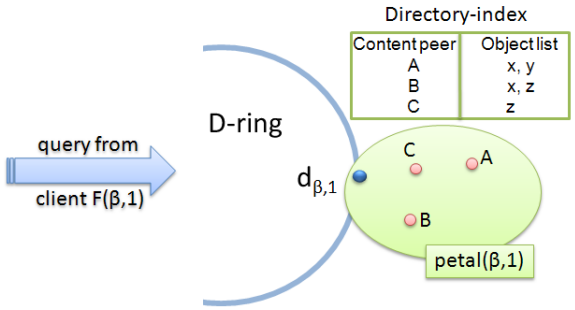

Figure 4: Query submitted by F, a new client of β in locality $loc = 1$.

2.2.2. 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 Section 2.3.1)
   - 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 [6], using the identifiers of the objects listed in *directory-index*(ws, loc_j).

Figure 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 $d_{\beta,2}$.

2.2.3. 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 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 $d_{ws,loc}$ (cf. Section 2.2.1), 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 Section 4 for a detailed 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.

2.2.4. Query Processing

Consider $query(o_{ws})$, a query that is submitted by a new client in locality *i* 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 1. 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. Then, c_{ws,loc_i} serves the object o_{ws} to the client. 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, d_{ws,loc_i} forwards $query(o_{ws})$ to d_{ws,loc_j} which proceeds with **process(***query*(o_{ws})**)**. When no satisfying directory or content peer is found, the client redirects its $query(o_{ws})$ to the website *ws*.

Figure 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 hold 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, $d_{\beta,1}$ forwards *q* accordingly to either $d_{\beta,0}$ or $d_{\beta,2}$. Otherwise, the client *F* redirects *q* to the website β.

2.2.5. 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 1, 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.

Algorithm 1 - 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
```

2.3. 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 Section 2.3.1). 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.

2.3.1. 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 three 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 [7]. Second, it eases information dissemination, such that peers discover new content and new peers providing some content [8]. 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 [7].

2.3.2. 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.
   - $\Delta list(c_{ws,loc})$: 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 three 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 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*(β, 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.

2.3.3. Gossip Behavior

The gossip behavior of each content peer $c_{ws,loc}$ is illustrated in Algorithm 2: 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*.

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

Algorithm 2 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
```

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 two 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 2, $c_{ws,loc}$ 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.

2.3.4. Push Behavior

Recall that the first access to *petal*(ws, loc) is provided by D-ring via its directory peer $d_{ws,loc}$ that 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.

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 Section 4.2).

Algorithm 3 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
```

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 3, $c_{ws,loc}$ 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.

Algorithm 4 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
```

As shown in Algorithm 4, $d_{ws,loc}$ 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.

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 [6] 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.

2.3.5. 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 5. 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 case 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 for o_{ws} to check if some $c'_{ws,loc}$ might have the requested object. In case $c'_{ws,loc}$ is available and alive, *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.

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 accross the set of content peers $c_{ws,loc}$ holding copies of o_{ws}.

2.4. 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 [4] requires $O(log2N)$ messages to update the P2P overlay when a single new peer joins. In a network of 30000 peers, we obtain 220 update messages. This message overhead does not

Algorithm 5 - process($query(o_{ws})$**)** at $c_{ws,loc}$

```
if content-list(c_ws,loc).contain(o_ws) then
  serve query(o_ws)
  if originator is new then
    view.add_Contact(originator, 0, null)
    send viewSubset to originator
  end if
  break
else
  if content-summaries is empty then
    redirect query(o_ws) to d_ws,loc
  else
    c'_ws,loc ← content-summaries.lookup(o_ws)
    if c'_ws,loc != null and c'_ws,loc is alive then
      redirect query(o_ws) to c'_ws,loc
    else
      redirect query(o_ws) to ws
    end if
  end if
end if
```

only increase 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 provides 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 on 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 gossip 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 into a single protocol to limit the overhead under churn and dynamicity. This issue is fully addressed in the next sections.

3. PetalUp-CDN

PetalUp-CDN is a scalable version of Flower-CDN that dynamically adapts to variable rates of participation. In the following, we first 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.

3.1. 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. Third, 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 in 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 4.2).

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, this 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, 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.

3.2. D-ring Architecture in PetalUp-CDN

The current structure of D-ring cannot support more than one directory peer for each pair (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 pair (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.

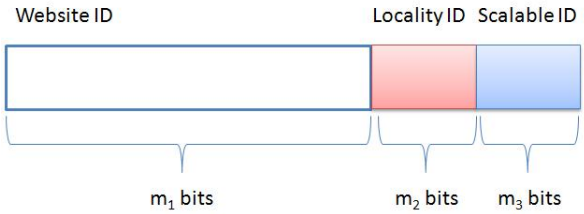

Figure 5: Peer ID structure in D-ring of PetalUp-CDN.

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 5. We thereby obtain 2^{m_3} consecutive peer IDs for each pair (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 2.2.2). 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 directory-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 6 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 $c_{\beta,1}$.

3.3. 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

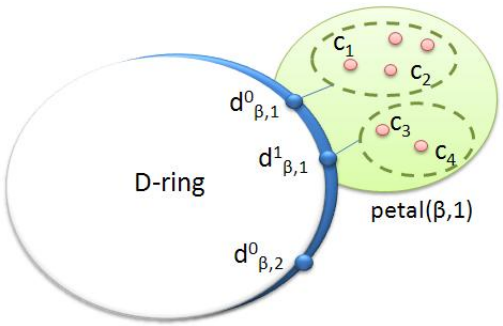

Figure 6: Example of $petal(\beta, 1)$ in PetalUp-CDN.

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.

3.3.1. 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 when they process new queries and finds out that the number of their content peers is at 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 information described in Section 2.2.1. 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 6, i.e., **scalable-process(**$query(o_{ws})$**)**. First, $d^i_{ws,loc}$ 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 a 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. Afterwards, in order to avoid waiting for $d^{i+1}_{ws,loc}$ to join, $d^i_{ws,loc}$ processes the query, in its stead, based on **process(**$query(o_{ws})$**)** of Algorithm 1. 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 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).

Algorithm 6 - 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))
```

3.3.2. 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 three 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 3.3). $d^l_{ws,loc}$ 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 7 and 8. Algorithm 7 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 7 - 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 8 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 8 - 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 9, $d^l_{ws,loc}$ 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 6 in Section 3.3). As depicted in Algorithm 10, it basically consists of $d^i_{ws,loc}$ receiving *directory-index*($d^l_{ws,loc}$) and merging it with its own directory-index.

Algorithm 9 $d^l_{ws,loc}$.**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 10 $d^i_{ws,loc}$.**takeOver**()

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

Upon the resignation of $d^l_{ws,loc}$, $d^{l-1}_{ws,loc}$ eventually detects that it is now the last directory peer of $petal(ws, loc)$ by discovering that its successor on D-ring belongs to a different petal.

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 4.2.

3.4. 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 6, 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 6. 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 4.1) 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. 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 $d_{ws,loc}$.

4.1. 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* on the one hand and exchange of *dir-info* on the other hand.

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 two 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 2.3.4, 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 6, 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).

4.2. 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.

4.2.1. 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. Any join message targeting the position $d^i_{ws,loc}$ reaches the 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 then 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.

4.2.2. 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 $d^{i+1}_{ws,loc}$ 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 11) 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 11 - 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
4:     { joinMessage reached a directory peer with the same
       target ID}
5:     dir-info.update(directoryPeer)
6:     if new client then
7:        join petal(ws, loc) as c_{ws,loc}
8:     end if
9:  else
10:    become d^i_{ws,loc}
11:    construct directory-index
12: 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 4.1, 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.

5. 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.

The rumor propagation is initiated by the author of S upon its first gossip round following the rumor creation. Since a content peer includes its own summary information in every gossip message, the author sends S in each gossip round. A content peer that receives the rumor is called *aware*. Once aware, a peer may participate in the rumor propagation at the rhythm of its gossip behavior. As done in most gossip studies (e.g. [9]), we assume that the rumor propagation can be broken into synchronous rounds during which every aware peer initiates a gossip exchange with one of its contacts.

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. [9], 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:

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

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.

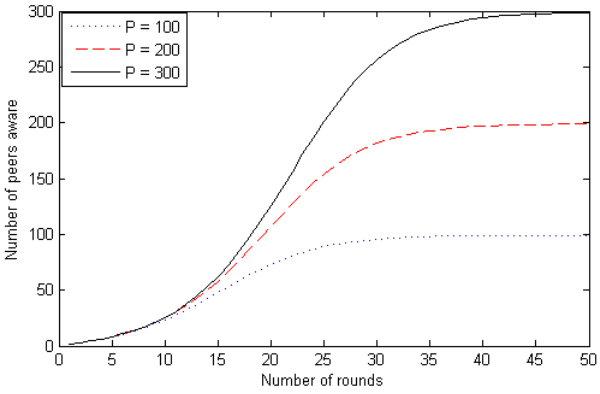

Figure 7: Impact of petal size P on the number of rounds required to spread the rumor in the petal

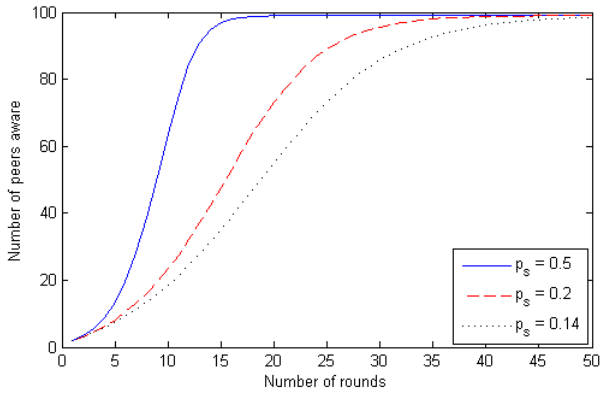

Figure 8: Impact of probability p_S on the number of rounds required to spread the rumor in a petal of size $P = 100$

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 1, we obtain:

$$R(f) = R(f\text{-}1) + 1 * p_{unaware/author}(f\text{-}1) + (R(f\text{-}1) - 1) * p_S) * p_{unaware}(f\text{-}1) \quad (2)$$

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

$$R(f) = \alpha\beta R^2(f\text{-}1) + (1 - \beta + \alpha + 2\alpha\beta)R(f\text{-}1) - \alpha\beta - \alpha + 2\beta + 1 \quad (3)$$

Equation 3 can be illustrated by a curve for some given values of p_S and P. In Figure 7, 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 8, 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 $(x+1)$

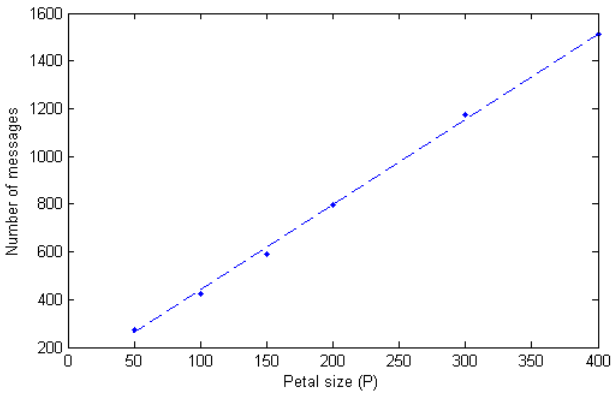

Figure 9: Impact of petal size P on the number of messages required to spread the rumor in the petal

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

$$M(f) = \sum_{x=1}^{f} msg(x) = \sum_{x=1}^{f} [1 + (R(x) - 1) * p_S] \quad (4)$$

In Figures 9 and 10, we illustrate the variation of $M(f)$ with p_S and P, respectively.

In Figure 9, 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. In Figure 10, 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

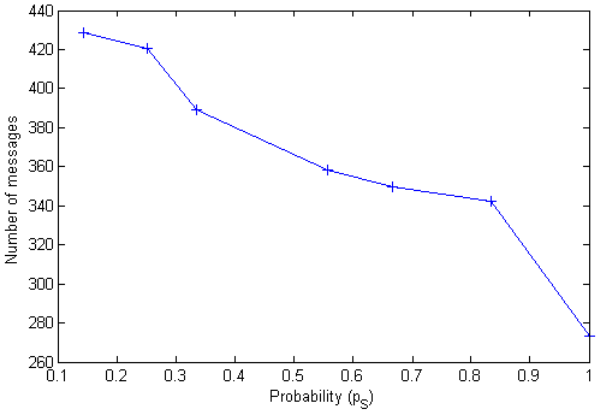

Figure 10: Impact of probability p_S on the number of messages required to spread the rumor in a petal of size $P = 100$

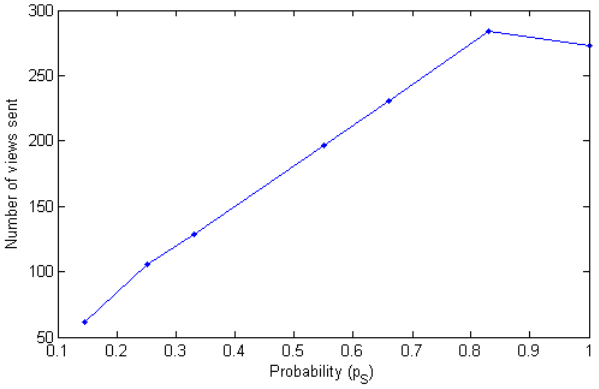

Figure 11: Impact of probability p_S on the total number of views exchanged to spread the rumor in a petal of size $P = 100$

achieved within fewer rounds, the number of redundant messages is significantly reduced.

Figure 11 shows the total number of views exchanged during the f rounds with increasing p_S. We derived this figure from Figure 10, 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.

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 9). 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 10 and 11. 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 10). 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 11).

6. Performance Evaluation

To extensively evaluate the performance of our protocols, we perform two simulation-based analyses under different environmental contexts and experimental setups. In the section, we first describe our evaluation methodology and then we present and discuss each analysis.

6.1. Evaluation Methodology

We conduct simulation-based experiments using PeerSim [10], 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 [11] assigns latencies between 10 and 500 ms. Localities are modeled using the binning technique [3]. We use $k = 6$ localities that are non-uniformly populated.

Given that D-ring relies on a DHT-structured overlay, we choose Chord overlay [4] for its simplicity; we simulate its routing and churn stabilization protocols and adapt its key management service as explained in Section 2.2.1, to be able to simulate the D-ring protocol. To construct D-ring overlay, we assume that Flower-CDN/PetalUp-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 [12, 20, 21]. 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 with respect to 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 nb-ob=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 [13]. 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 targeted 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.

Our performance evaluation covers two analyses in both static and dynamic environments. The main simulation parameters used in the analyses are summarized in Table 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 objects provided by its website, thus we set *summary size* according to the analysis in [6], 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. Section 2.3.4). V_{gossip} refers to the view size and T_{gossip} to the gossip period as described in Section 2.3.1 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.

In our experiments, we measure the following performance metrics:

- **Background traffic**: the average traffic in 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.

Table 1: Simulation Parameters

Parameter	**Static setup**	**Dynamic setup**
Latency	10-500 ms	10-500 ms
Nb of localities k	6	6
Nb of websites $\|W\|$	100	100
Population P	4200	3000-15000
Underlying network	5000	$P * 1.3$
Mean uptime m	-	60 min
Nb of objects/website	500	500
Query rate	6 queries/sec	1 query/6 min/peer
Summary size	8*500 bits	8*500 bits
Push threshold	0.1; 0.5; 0.7	0.5
V_{gossip}	20; 50; 70	≤ 30
T_{gossip}	1 min; 30 min; 1 h	1 h
L_{gossip}	5; 10; 20	≤ 10

6.2. Performance in Static Environment

The first set of experiments is conducted in a static environment where peers do not leave the system after joining it. Here, we focus on quantifying the gains in Flower-CDN due to locality-awareness. Furthermore, we aim at evaluating the price to be paid for achieving these gains, by examining the trade-off between hit ratio and gossip bandwidth consumption.

6.2.1. Static Setup

Experiments start with a stable D-ring: for each pair (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.

6.2.2. 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

[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 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)

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 three 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 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 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 2. Yet, the increase in hit ratio is not substancial.

Table 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.

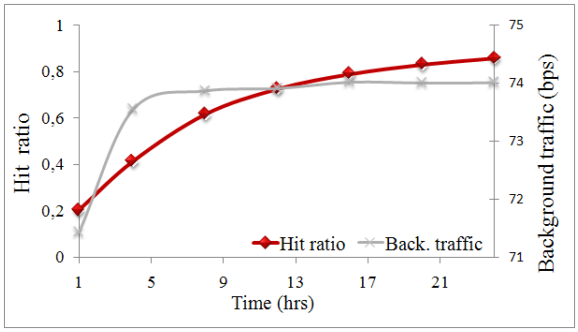

Figure 12: Trade off between hit ratio and bandwidth in Flower-CDN

Table 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 12 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.

6.2.3. Hit ratio

The following results compare DHT-Directory and Flower-CDN wrt. hit ratio. Figure 13 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 targeted *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. 2.2.3), 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}.

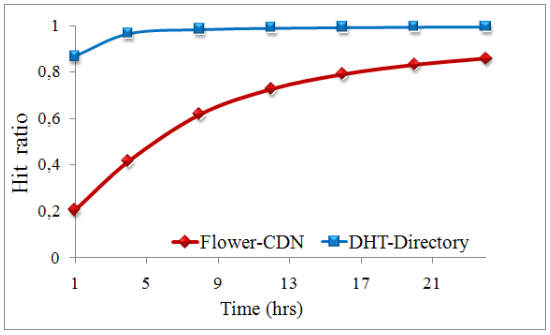

Figure 13: Hit ratio evolution in static environment

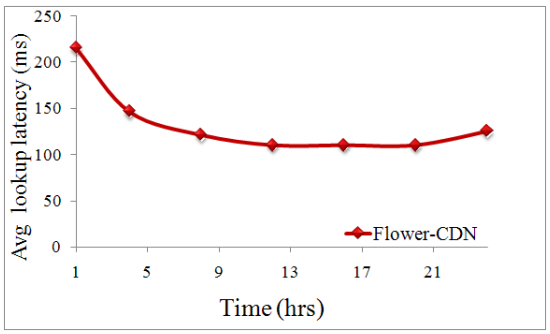

Figure 14: Lookup latency evolution in static environment

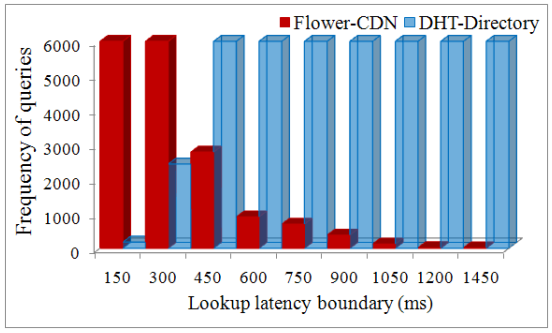

Figure 15: Lookup latency distribution in static environment

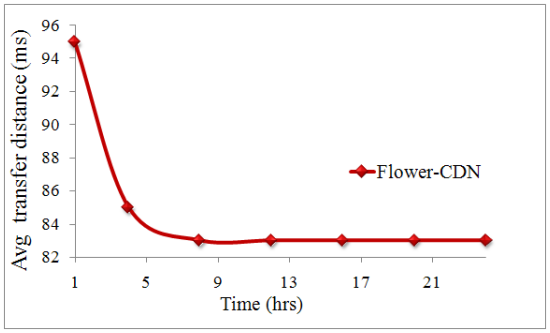

Figure 16: Transfer distance evolution in static environment

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.

6.2.4. *Locality-awareness*

We evaluate the gains due to locality-awareness in Flower-CDN, by measuring lookup latency and transfer distance.

The first experiment measures the lookup latency. Figure 14 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 15 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.

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 16 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 17 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.

6.2.5. *Discussion*

We learnt two main lessons through our first 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

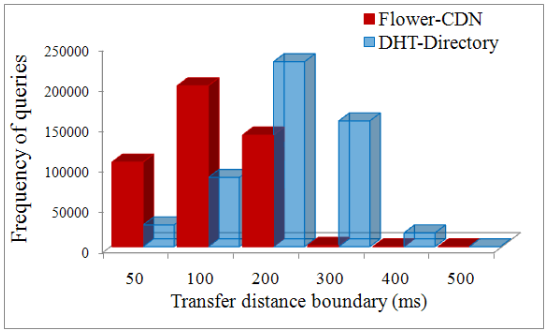

Figure 17: Transfer distance distribution in static environment

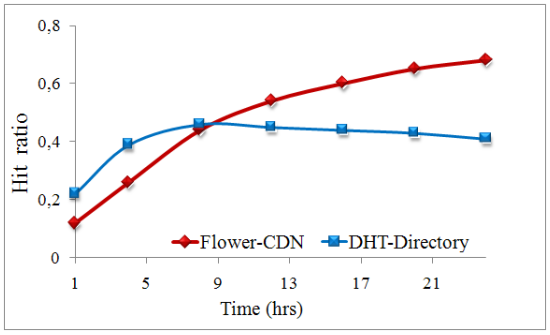

Figure 18: Hit ratio evolution in dynamic environment

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.

6.3. Performance in Dynamic Environment

The second set of experiments is conducted in a dynamic environment where peers connet and disconnect. We want to assess the robustness of Flower-CDN under churn and evaluate its scalability wrt. the population size.

6.3.1. Dynamic Setup

For a realistic environment, we simulate churn based on a study [14] 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 simulate 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) has 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 pair (website, locality)). After a small warm-up period, the population stabilizes around P as new clients keep on arriving and existing peers fail. 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 automatically with the size of its petal which reaches at most 30 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.

6.3.2. 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 18, 19 and 20.

First, we analyse the evolution of hit ratio with time (Figure 18). 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 the 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.

Second, we look at the distribution of queries with respect to lookup latency and transfer distance for $P = 3000$. Figure 19 shows that 66% of our queries are resolved within 150 ms

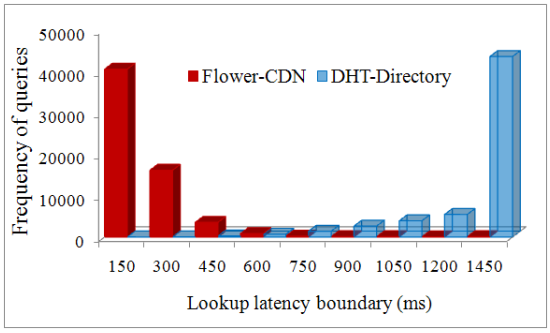

Figure 19: Lookup latency distribution in dynamic environment

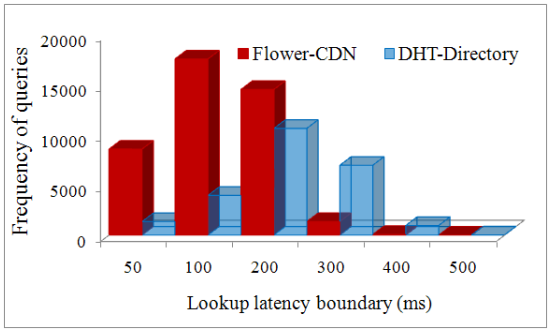

Figure 20: Transfer distance distribution in dynamic environment

while 75% of DHT-Directory's queries take more than 1200 ms. Figure 20 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.

6.3.3. 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.

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 five 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 three 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 3.

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 experiences 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.

Table 3: Scalability comparison.

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

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 four experiments targeting a population size of 15000[2]. Each experiment depicts the behavior of PetalUp-CDN for 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

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

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).

First, let us analyse the results of background traffic (Figure 6.3.3). 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.

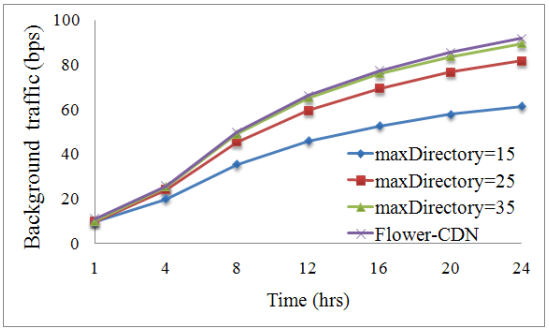

Figure 21: Overhead in PetalUp-CDN.

When examining hit ratio evolution (Figure 6.3.3), 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.

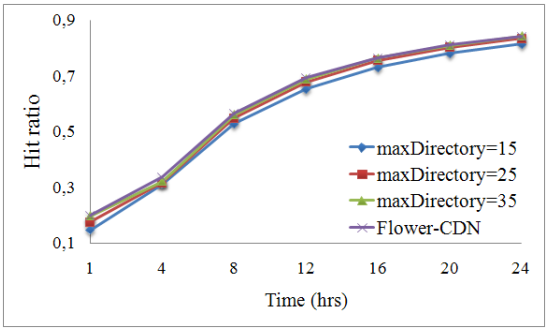

Figure 22: Hit ratio in PetalUp-CDN.

Regarding lookup latency (Figure 6.3.3) and transfer distance (Figure 6.3.3), 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.

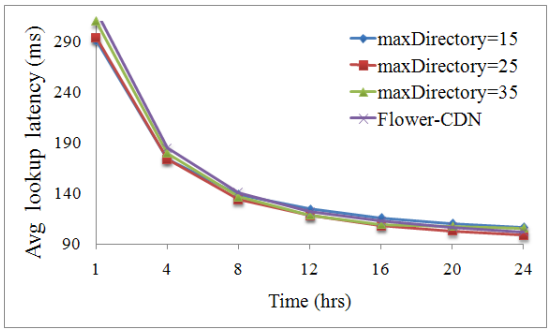

Figure 23: Lookup latency in PetalUp-CDN.

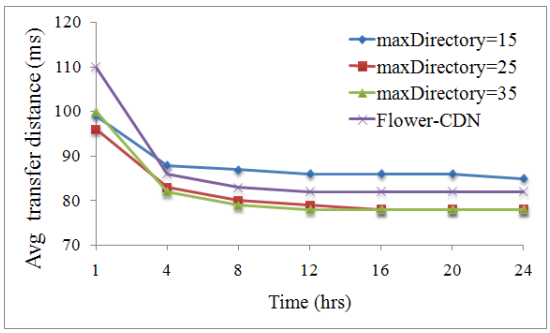

Figure 24: Transfer distance in PetalUp-CDN.

6.4. 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. 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.

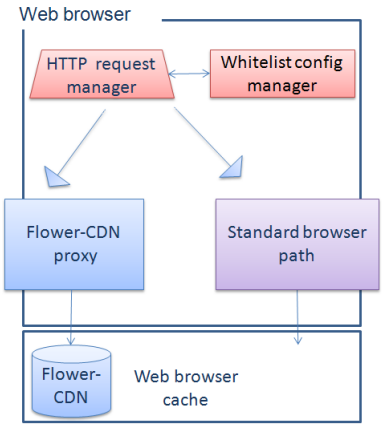

Figure 25: Flower-CDN extension within the web browser

7. Deployment

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 particpants 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.

In this section, we give some guidelines on how Flower-CDN can be deployed and used in practice.

7.1. Flower-CDN browser extension

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.

Flower-CDN functionality can be implemented as a browser extension. As shown in Figure 25, two main Flower-CDN components are integrated into a browser that installs Flower-CDN extension: an HTTP request manager and a Flower-CDN proxy.

As shown in Figure 25, 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 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 2.3.3 and the directory information via push and keepalive exchanges cf. Section 4.1).

The web browsing process begins when the user inputs a URL into the browser and initiates an HTTP request. The HTTP request is first handled by the Flower-CDN HTTP request manager. This manager has 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. Thus, the manager 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 deepen our explanation on how a user is connected to the Flower-CDN network (i.e., D-ring or petals).

7.2. 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 in her Flower-CDN whitelist 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 7.2 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* targetted 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 [3], 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 that each possible landmark ordering identifies a locality, the user detects her locality *loc* based on her ordering.

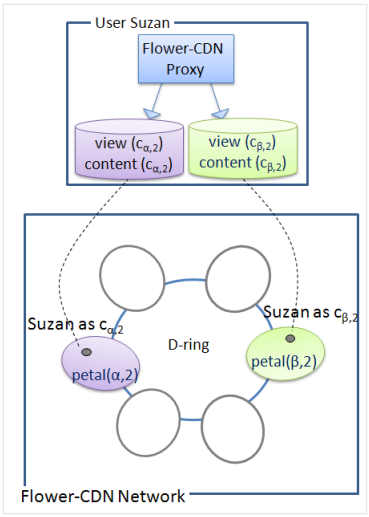

Figure 26: A user in Flower-CDN as two content peers related to two different websites

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 [15]. The Flower-CDN proxy running within the user's web browser only needs the web-server's public key to verify the digital signature of an object related to this web-server and received by the user from some content peer. This solution is indifferent to peer dynamicity and copes well with a loosely-trusted environment.

7.3. 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 targetted 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 targetted 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 targetted 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 into its petal. However, if *p*'s view contains no available contact, *p* cannot reintegrate into its petal and thus has to rejoin Flower-CDN as a new client.

8. Related Work

We categorize existing P2P CDNs into three main classes: centralized, unstructured and structured.

The first category [16, 17] 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. 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. In contrast, PetalUp-CDN pools the resources of the clients to expand and support higher scales.

The second category of approaches (e.g., Proof [18] and BuddyWeb[19]) uses unstructured overlays for their flexibility and inherent robustness. For instance, in Proofs [18], peers continuously exchange neighbors among each other so that each peer gets a random view of the network for each search operation. Peers keep their requested objects and provide them to other participants. They use flooding to locate their requested objects. The continuous randomization of the overlay has the benefit of improving the network fault-tolerance and load balancing. However, searching for not-so popular objects induces heavy traffic and high latency. In Flower-CDN/PetalUp-CDN, search is confined to petals and guided by the content-summaries. Moreover, Proofs does not leverage locality-awareness.

The third category of approaches (e.g., Squirrel [12], PoP-Cache [20] and Backslash [21]) relies on DHT to achieve fast lookup and propose basically two types of strategies, DHT-Home and DHT-Directory. The first one replicates web objects at peers with ID numerically closest to the hash of the URL of the object without any locality or interest considerations. 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. The DHT-Directory 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. 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: the directory information is abruptly lost at the failure of its storing peer, which may severely degrade performance. Additionally, there are two main drawbacks in the query routing of both strategies. First, each query has to navigate through the whole DHT, which implies more routing load and higher response times than Flower-CDN/PetalUp-CDN. Second, unless using a locality-aware overlay combined with proactive replication, the query is served from a random physical location whereas our protocols

rely on a locality-aware infrastructure that directs each query according to the physical location of the client.

In [22], peers are organized using a hash function into $\sqrt{N}$ groups where N = total number of peers. 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 whereas gossip in Flower-CDN/PetalUp-CDN only involves peers of the same petal which are interested in the same content. Furthermore, since directory information is highly replicated, aggressive updates are required under churn and dynamic changes.

To the best of our knowledge, the P2P CDNs that are currently available for public use comprise CoralCDN [23], CoDeeN [24] and CobWeb [25]. These systems are deployed over PlanetLab which provides a relatively trusted environment consisting of nodes donated largely by the research community. We examine one such representative system. CoralCDN [23] 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. A key is mapped to several nodes whose IDs are numerically close to the key. A node stores pointers related to the object whose key is mapped to its node ID. 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 DNS 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 wrt. the object in the different overlays to which belongs this proxy. To handle dynamicity, pointers are associated with ttl values and are periodically refreshed by their referenced proxy. 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 in terms of proxy caches.

9. Conclusion

In this paper, we describe 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. 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 can be tuned according to the available network resources and hit ratio requirements.

For scalability purposes, we proposed PetalUp-CDN which enables Flower-CDN's infrastructure to dynamically evolve and avoid overload situations. The performance evaluation demonstrated that this new scheme does not affect hit ratio and response times, thus enabling efficient scalability.

We ensured the robustness of 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 comparison with an existing P2P CDN, hit ratio is ameliorated by 40% and response times reduced by a factor of 12.

We plan to extend this work for social networks, mainly by elaborating more on the concept of interests and adding personalized searching features.

References

[1] M. E. Dick, E. Pacitti, B. 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), 2009, pp. 427–438.

[2] M. E. Dick, E. Pacitti, B. 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), 2009, pp. 180–185.

[3] S. Ratnasamy, M. Handley, R. M. Karp, S. Shenker, Topologically-aware overlay construction and server selection, in: Proceedings of the 21st IEEE International Conference on Computer Communications (INFOCOM), 2002, pp. 1190–1199.

[4] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. 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), 2001, pp. 149–160.

[5] A. Rowstron, P. Druschel, Pastry: Scalable, decentralized object location, and routing for large-scale P2P systems, in: Proceedings of the 2nd ACM/IFIP International Conference on Middleware, Vol. 2218 of LNCS, Springer, 2001, pp. 329–350.

[6] L. Fan, P. Cao, J. Almeida, A. 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), 1998, pp. 291–293.

[7] S. Voulgaris, D. Gavidia, M. Steen, Cyclon: Inexpensive membership management for unstructured P2P overlays, Journal of Network and Systems Management 13 (2) (2005) 197–217.

[8] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, L. Massoulieacute, Epidemic information dissemination in distributed systems, IEEE Computer 37 (5) (2004) 60–67.

[9] A. Datta, M. Hauswirth, K. Aberer, Updates in highly unreliable, replicated peer-to-peer systems, in: Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS), 2003, pp. 76–.

[10] M. Jelasity, A. Montresor, G. P. Jesi, S. Voulgaris, The PeerSim simulator, `http://peersim.sf.net`.

[11] A. Medina, A. Lakhina, I. Matta, J. Byers, BRITE topology generator, `http://www.cs.bu.edu/brite/` (2002).

[12] S. Iyer, A. I. T. Rowstron, P. Druschel, Squirrel: a decentralized P2P web cache, in: Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC), 2002, pp. 213–222.

[13] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, Web caching and Zipf-like distributions: evidence and implications, in: Proceedings of the

18th IEEE International Conference on Computer Communications (INFOCOM), 1999, pp. 126–134.
[14] D. Stutzbach, R. Rejaie, Characterizing churn in P2P networks, Technical report CIS-TR-2005-03, University of Oregon (2005).
[15] I. J. Taylor, From P2P to Web services and Grids: peers in a client/server world, Springer, 2004, Ch. Security.
[16] Y.-S. Ryu, S.-B. Yang, An effective P2P web caching system under dynamic participation of peers, IEICE Transactions 88-B (4) (2005) 1476–1483.
[17] V. N. Padmanabhan, K. Sripanidkulchai, The case for cooperative networking, in: Proceedings of the 1st International Workshop on P2P Systems (IPTPS), Vol. 2429 of LNCS, Springer, 2002, pp. 178–190.
[18] A. Stavrou, D. Rubenstein, S. Sahu, A lightweight, robust P2P system to handle flash crowds, in: Proceedings of the 10th IEEE International Conference on Network Protocols (ICNP), 2002, p. 226.
[19] X. Wang, W. S. Ng, B. C. Ooi, K.-L. Tan, A. Zhou, Buddyweb: A P2P-based collaborative web caching system, in: Revised papers from the NETWORKING Workshops on Web Engineering and Peer-to-Peer Computing, Vol. 2376 of LNCS, Springer, 2002, pp. 247–251.
[20] W. Rao, L. C. 0002, A. W.-C. Fu, Y. Bu, Optimal proactive caching in P2P network: analysis and application, in: Proceedings of the 6th ACM International Conference on Information and Knowledge Management (CIKM), 2007, pp. 663–672.
[21] T. Stading, P. Maniatis, M. Baker, P2P caching schemes to address flash crowds, in: Proceedings of the 1st International Workshop on P2P Systems (IPTPS), Vol. 2429 of LNCS, Springer, 2002, pp. 203–213.
[22] P. Linga, I. Gupta, K. Birman, A churn-resistant P2P web caching system, in: Proceedings of the ACM Workshop on Survivable and Self-Regenerative Systems (SSRS), 2003, pp. 1–10.
[23] M. J. Freedman, E. Freudenthal, D. Mazières, Democratizing content publication with Coral, in: Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2004, pp. 239–252.
[24] V. S. Pai, L. Wang, K. Park, R. Pang, L. Peterson, The dark side of the Web: an open proxy's view, ACM SIGCOMM Computer Communication Review 34 (1) (2004) 57–62.
[25] Y. J. Song, V. Ramasubramanian, E. G. Sirer, Optimal resource utilization in content distribution networks, Technical report TR2005-2004, Cornell University (2005).