Flower-CDN: A hybrid P2P overlay for Efficient Query Processing in CDN*

Manal El Dick
Atlas Team, INRIA and LINA, University of Nantes, France
manal.el-dick@univ-nantes.fr

Esther Pacitti
Atlas Team, INRIA and LINA, University of Nantes, France
esther.pacitti@univ-nantes.fr

Bettina Kemme
McGill University, Montreal, Quebec, Canada
kemme@cs.mcgill.ca

ABSTRACT

Many websites with a large user base, e.g., websites of non-profit organizations, do not have the financial means to install large web-servers or use specialized content distribution networks such as Akamai. For those websites, we have developed Flower-CDN, a locality-aware P2P based content-distribution network (CDN) in which the users that are interested in a website support the distribution of its content. The idea is that peers keep the content they retrieve and later serve it to other peers that are close to them in locality. Our architecture is a hybrid between structured and unstructured networks. When a new client requests some content from a website, a locality-aware DHT quickly finds a peer in its neighborhood that has the content available. Additionally, all peers in a given locality that maintain content of a particular website build an unstructured content overlay. Within this overlay, peers gossip information about their content allowing the system to maintain accurate information despite churn. In our performance evaluation, we compare Flower-CDN with an existing P2P-CDN strictly based on DHT and not locality aware. Flower-CDN reduces lookup latency by a factor of 9 and transfer distance by a factor of 2. We also show that Flower-CDN's gossip has low overhead and can be adjusted according to hit ratio requirements and bandwidth availability.

1. INTRODUCTION

Content Distribution Networks (CDN) such as Akamai [1], are well-known technologies for distributing the content of web-servers to large audiences. The main mechanism is to replicate requested content at dedicated and widely dispersed machines. By efficiently serving clients' queries, these technologies decrease the workload on the original web-servers, reduce bandwidth costs, and keep the client's perceived latency low.

*Work partially funded by the French ANR DataRing project.

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

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. In this paper, we propose to use peer-to-peer (P2P) technology to build a CDN infrastructure that aims at serving popular websites that cannot afford their own proprietary infrastructure. Peer-to-peer (P2P) technology is an attractive alternative for redistributing content at large scale with low costs, by exploiting the underutilized resources of clients. In fact, many projects have demonstrated that users are willing to contribute to websites with content they are interested in (e.g., fund-raising and editing in Wikipedia, sharing idle computer resources in SETI@home, etc.).

Any CDN has to address four main issues: *response time*, *scalability*, *hit ratio* and *liveness*. By replicating the content across the CDN, the CDN can serve many client requests leading to a high hit ratio and availability despite individual node failures. Additionally, response times are short if efficient routing algorithms find replicas close to the client. Finally, scalability is achieved by increasing the CDN size as the load increases, thus always providing a balanced network load. When designing such a CDN over a P2P infrastructure, particular challenges arise because the peers are autonomous and volunteer participants. Additionally churn rate is much higher than in dedicated CDN infrastructures. In many existing P2P solutions, the queried content is replicated on demand at requesting peers, resulting in a random replica placement spread over the P2P overlay (e.g., [17, 13]). Different from on demand replication, some approaches (e.g., [14]) increase the degree of replication by replicating content among peer neighborhood and thus forcing peers to store content they may not be interested in. Independently of who stored replicas, in most existing approaches [10, 14, 17, 18, 19], incoming queries are routed without considering whether the content requested in the query is available in a peer that is physically close to the requestor. In contrast, traditional CDN generally consider locality-awareness to take advantage of close-by replicas as it has the potential to dramatically reduce response times [7] as well as bandwidth consumption and thus increase system scalability. Therefore, we make locality-awareness a top priority in

the development of a P2P CDN.
Furthermore, the particular properties of P2P require the directory service used to locate content to be carefully distributed over participant peers because no participant peer should be subject to overload nor should the system expose a single-point of failure or bottleneck. Many existing P2P approaches do not sufficiently address this issue: they either rely on blind searches which induce heavy traffic and limit scalability (e.g., [19]), or they centralize the directory at a single peer, in particular the web-server, which becomes quickly overloaded (e.g., [17]). In contrast, we believe that the directory service itself should be implemented in a P2P manner in order to face churn and assure liveness.
Considering all these issues, we propose a locality and interest aware P2P CDN, *Flower-CDN*, that enables any under-provisioned website to efficiently distribute its content, with the help of the non-profit community interested in its content. To handle this, Flower-CDN combines efficient DHT indexing to provide fast lookup with gossip robustness for replica distribution and self-monitoring. The basic idea is to let each peer be connected to a *content overlay* which represents a cluster of peers that have the same interest and reside close to each other. Peers in a content overlay keep content of a certain website. A peer posing a query can find a close-by content overlay through a special directory service, called *D-ring*, which implements a locality-aware DHT. Content overlays and their connection to the D-ring are maintained via low-cost gossip techniques among participant peers.
More precisely, this paper makes the following contributions:

- We propose a scalable P2P directory service *D-ring*. Each directory peer of D-ring indexes the content of a specific content overlay. D-ring is based on novel locality- and interest-aware key management and routing services. It can be easily integrated into existing DHT overlays.
- We show how each directory peer interacts with its related content overlay and how the content overlay is managed by the use of gossip protocols. Our gossip mechanisms allow directory peers and participant peers to maintain accurate information despite dynamic changes and failures.
- We present an efficient query routing algorithm for Flower-CDN that first seeks for some content in the content overlay related to the locality of the requestor. If this search is unsuccessful, the query is forwarded to other localities or falls back to the web-server.
- Finally, we present a detailed performance evaluation. Our experimental results show that Flower-CDN can reduce lookup latency by a factor of 9 and the transfer distance by a factor of 2, compared to an existing P2P CDN (Squirrel [10]). Moreover, Flower-CDN incurs very acceptable overhead in terms of gossip bandwidth, which can also be tuned according to hit ratio requirements and bandwidth availability.

The rest of this paper is structured as follows. Section 2 gives an overview of Flower-CDN and defines the main terms used in the paper. Section 3 describes D-ring. It introduces its locality- and interest-aware key management, and presents its query routing service. Section 4 presents the details of the content overlay of Flower-CDN. Section 5 discusses how Flower-CDN is managed in the presence of churn and failures. Section 6 presents a detailed simulation-based performance analysis. Section 7 discusses related work. Finally, Section 8 concludes the paper.

2. OVERVIEW OF FLOWER-CDN

In this section, we present a general overview of Flower-CDN, introducing the main terms and assumptions used in the remainder of the paper.
Flower-CDN is designed to support a set W of websites, each of them having its own set of web-pages and documents. Flower-CDN exploits the willingness of the clients of a website to cooperate in order to redistribute the content they are interested in. A website ws is added to W on either the website's own initiative or some of its clients' initiative.
In order to implement locality-awareness in Flower-CDN, we assume that the Internet is split into network localities, which can be provided by an existing technique [15]. 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.
Participant peers belonging to the same locality loc and interested in the same website ws build together an overlay noted *content-overlay*(ws, loc), using gossip techniques. These peers, called *content peers* and noted $c_{ws,loc}$, store, manage and exchange content of ws (e.g., web pages, documents), thus considerably relieving the server of ws from its query load[1]. Flower-CDN charges one peer of each *content-overlay*(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 a structured overlay called *D-ring* based on a *Distributed Hash Table* (*DHT*), to support queries coming from new clients, that request objects of content provided by any of the websites in W (e.g., query for web page with given URL). DHTs are a special form of overlay over a set of peers that enables queries to be routed quickly to their destination peers. In our context, we use D-ring to support queries coming from new clients, that request objects of content provided by any of the websites in W (e.g., query for web page with given URL). Furthermore, directory peers of the same website ws may collaborate to provide content of ws.
In summary, Flower-CDN relies on a hybrid architecture consisting of a set of independent content overlays linked via one directory overlay (i.e., D-ring), as illustrated in Fig. 1. 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 wrt. loc i.e., $d_{ws,loc}$. Then, $d_{ws,loc}$ tries to resolve the query while relying on its content overlay or some other content overlays of 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

[1]There must also be some consistency management in place in case the website changes the content. Consistency mechanisms developed for web-caches could be applied. Such consistency policies are, however, out of the scope of this paper.

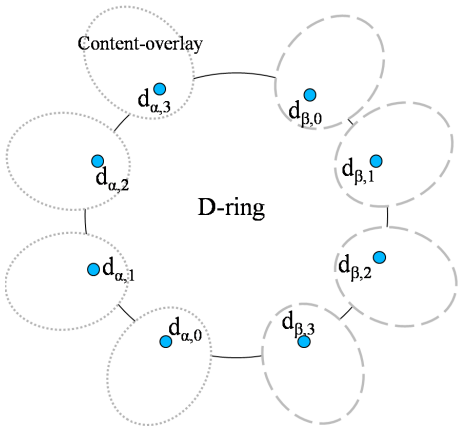


Figure 1: Flower-CDN architecture with websites α and β and four localities

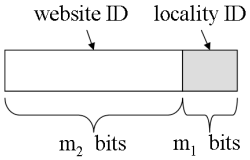


Figure 2: The peer ID structure in D-ring

can join *content-overlay*(ws, loc) as a content peer $c_{ws,loc}$, if it is willing to contribute storage resources wrt. content of ws. For further queries, $c_{ws,loc}$ searches directly in its *content-overlay*(ws, loc) instead of relying on D-ring.
This means that in Flower-CDN all peers that are willing to support a certain website $ws \in W$ become part of one of the content-overlays of ws helping ws to distribute its content. We denote this set of peers as P_{ws}:
$\forall ws \in W : P_{ws} = \bigcup_{0 \leq loc < k}$ *content-overlay(ws,loc)*

3. D-RING MODEL

In this section, we present the P2P directory overlay, *D-ring*, which ensures reliable access to Flower-CDN. 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. We describe the different aspects of D-ring: key management, routing service, directory structure and finally query processing.

3.1 Key Management

In order to ensure a fast lookup, D-Ring can be integrated into any existing structured overlay based on a standard DHT (e.g., Chord [20], Pastry [16]). For each website $ws \in W$, the directory overlay enables k participant peers from P_{ws}, where k is the number of localities, to join as directory peers for ws: each locality loc is covered by a directory peer $d_{ws,loc}$, to empower locality-aware redirection of queries. In the example of Figure 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 participating in D-ring.

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

- **locality ID**:
  - identifier of the locality to which the directory peer belongs. It is expressed using the lowest bit-segment of length m_1.
  - Each 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 by hashing the url of ws (noted $hash(ws)$). The hash function assigns identifiers to websites from the subspace $S' = [1 \cdots 2^{m_2} - 1]$.

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.

3.2 Routing Service

In its stable structure, D-ring has a directory peer for each tuple (website, locality). A message targeting the website

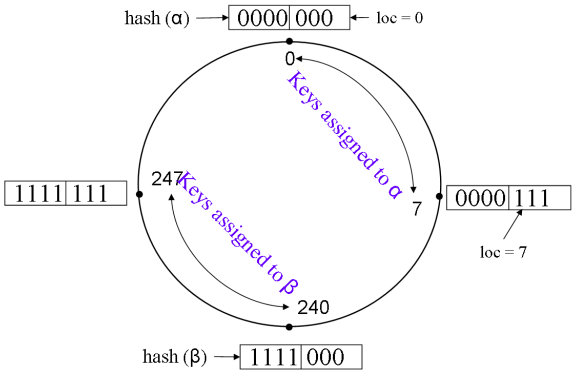


Figure 3: D-ring distribution of keys given that $k = 8$ and $W = \{\alpha, \beta\}$.

ws and the locality loc is routed using the key composed of the website ID of ws and the locality ID of loc. Thus, the message is delivered by the DHT key-based routing service to its destination, i.e., to the directory peer in charge. However, the targeted directory $d_{ws,loc}$ peer may be momentally unavailable or it may not have joined D-ring yet. In such cases, another directory peer of the same website ws should handle the message. Actually, the DHT key-based routing service redirects the message to the directory peer that has an ID that is numerically closest to $d_{ws,loc}$. Since the directory peers of ws are neighbors on the identifier circle, it's highly probable that the message reaches one of them, but it might be redirected to a directory peer of another website. Some existing DHT overlays (e.g, Chord) guarantee D-ring redirection, while others (e.g., Pastry) need to be adapted. To ensure the appropriate redirection in the latter, we apply a slight modification to their DHT routing service, in order to ensure that the message is routed towards a directory peer belonging to the same website ws as $d_{ws,loc}$.
To clearly show the modifications implied by D-ring, we first define the key-based routing API for structured overlays, based on [5]. The operation **route(key,msg)** is used to send/forward a message msg towards the peer with the ID equal or numerically closest to key. Algorithm 1 shows the DHT standard **route**, which is run at each DHT peer p that receives msg. p performs a local lookup using its routing table. If it determines it is the closest peer, then the message has reached its destination and is delivered. Otherwise, p selects among the peers it knows of the peer p' whose ID is the closest to the key. To use D-Ring on top of an existing

Algorithm 1 - DHT Standard **route(key,msg)**

```
// find closest peer to key, p', from routing table or itself
Peer p' ← local_lookup(key)
if p == p' then
  deliver msg
else
  forward(key, msg) to p'
end if
```

structured overlay, 2 steps are added to the standard **route**; D-ring version of **route** is depicted in Algorithm 2. Once the normal local lookup is performed at peer p, the website ID of p' is checked against the website ID of key. Then, an additional **conditional_local_lookup** may be launched: it searches for the numerically closest peer to key with the same website ID as key, that p knows about (p' may have a different locality ID than key). If no such peer is found, the previously found p' is kept as a result.

Algorithm 2 - D-ring **route(key,msg)**

```
// find closest peer to key, p', from routing table or itself
Peer p' ← local_lookup(key);
if p == p' then
  deliver msg
else if p'.websiteID != key.websiteID then
  // find closest peer to key, p', with equal websiteID
  p' ← conditional_local_lookup(key, key.websiteID);
end if
forward(key, msg) to p';
```

3.3 Directory Peer Structure

To handle submitted queries, a directory peer d_{ws,loc_i} uses two local data structures:

1. *Directory-index*(ws, loc_i): a directory that indexes the content of ws stored in *content-overlay*(ws, loc_i). The directory contains an entry for each content peer c_{ws,loc_i}, consisting of 3 fields:
   - information about the address of c_{ws,loc_i} (e.g., IP address)
   - age field useful for failure and leave detection (presented in Sec. 4.2)
   - 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 its *content-overlay*(ws, loc_i).
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 [9], using the identifiers of the objects listed in *directory-index*(ws, loc_j).

Figure 4 shows a simplified 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 *content-overlay*$(\beta, 1)$, their objects (e.g., A holds objects x and y which are initially provided by website β). Moreover, $d_{\beta,1}$ stores directory summaries received from its direct neighbors i.e., $d_{\beta,0}$ and $d_{\beta,2}$.

3.4 Query Processing

In the following, we refer by o_{ws} to an object of the content of ws and by *query*(o_{ws}) to the query requesting o_{ws}.
When a new client submits *query*(o_{ws}), D-ring routing service delivers *query*(o_{ws}) to the directory peer in charge of ws in the client's locality loc_i: the routed key is generated using

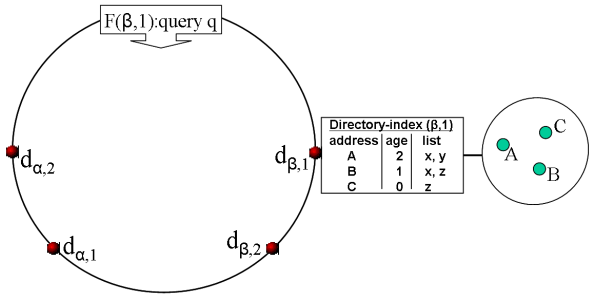


Figure 4: Query submitted by F, a new client of β in locality $loc = 1$

Algorithm 3 - 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
```

loc_i and ws, as described in Sec. 3.1. Upon the reception of $query(o_{ws})$, d_{ws,loc_i} processes it as shown in Algorithm 3. d_{ws,loc_i} searches first its directory index for the requested object o_{ws}. If $directory\text{-}index(ws, loc_i)$ shows that o_{ws} is stored by some content peer c_{ws,loc_i}, d_{ws,loc_i} redirects $query(o_{ws})$ to c_{ws,loc_i} after checking its aliveness. Otherwise, d_{ws,loc_i} queries the directory summaries, to check if some d_{ws,loc_j} might have the requested object in its directory index. In case d_{ws,loc_j} is found, $query(o_{ws})$ is redirected to d_{ws,loc_j} which proceeds with **process(**$query(o_{ws})$**)**. When no satisfying directory or content peer is found, $query(o_{ws})$ is redirected to the website ws.

In the example of Figure 4, let us consider a client F of website β that submits its query q to D-ring: q requests β's object x. Assuming that client F is located in $loc = 1$, q is forwarded to the peer $d_{\beta,1}$ which searches its directory index for x. Then, $d_{\beta,1}$ redirects q to content peer A or C, which hold a copy of the requested object x and thus can serve the query. When the client F requests x', which is not contained by any peer in $content\text{-}overlay(\beta, 1)$, $d_{\beta,1}$ first checks its *directory-summaries* for, $(\beta, 0)$ and $(\beta, 2)$ to see if they might have x' in their directory index. If it appears so, $d_{\beta,1}$ redirects q accordingly to either $d_{\beta,0}$ or $d_{\beta,2}$. Otherwise, $d_{\beta,1}$ redirects q to the website β.

After processing q, the client F becomes a content peer $c_{\beta,1}$. $d_{\beta,1}$ optimistically adds a new entry in its directory index: peer F with its requested object, i.e., x or x', and age zero. The next section explains how $d_{\beta,1}$ checks for the validity of its directory entries.

Once a client has become a content peer $c_{ws,loc}$, any subsequent queries that the client poses for website ws directly use the $content - overlay(ws, loc)$ instead of the D-ring (more details in Sec.4). Thus, D-ring only serves as a first access to content overlays, letting a new peer located in loc and interested in ws find its $content - overlay(ws, loc)$.

4. CONTENT OVERLAY MODEL

In this section, we describe how the content overlays are constructed and maintained via gossip algorithms, and how they interact with D-ring to process queries.

4.1 Construction

Recall that a $content\text{-}overlay(ws, loc)$ consists of one directory peer $d_{ws,loc}$ and several content peers $c_{ws,loc}$, all of which reside in the same locality loc and are interested in the content provided by ws.

Content overlays are dynamically built as follows. $d_{ws,loc}$ is the starting point of its $content\text{-}overlay(ws, loc)$. After the directory peer is established, subsequent peers in loc wishing to support ws, join the content overlay as content peers $c_{ws,loc}$. In practice, this occurs when a peer p performs its first search for an object o_{ws} of ws. Therefore, p first accesses $d_{ws,loc}$, using the key-based routing service described in Sec. 3.2. Then, after being served, p keeps its copy of o_{ws} for subsequent requests; p thus becomes content peer $c_{ws,loc}$ and is added to $directory\text{-}index(ws, loc_i)$.

As a client of ws, a content peer $c_{ws,loc}$ may wish to access objects of ws other than those available in its local storage. To avoid having all subsequent queries of content peers be directed to the directory peer, content peers exchange, within their overlay, summaries of their stored content of ws (more details are given in Sec. 4.2). Hence, $c_{ws,loc}$ can search the summaries of its $content\text{-}overlay(ws, loc)$ to see where a copy of its requested object might be stored.

By serving queries, Flower-CDN enables progressive replication of an object of W throughout $content\text{-}overlay(ws, loc)$, based on its popularity in locality loc. Therefore, at the redirection of queries for o_{ws} by 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}.

4.2 Gossip-based Management

Gossip-style communication is used throughout a content overlay to disseminate summaries and their updates in an epidemic manner [6]. Peers also gossip to discover new members in their overlay and to detect failed ones. We chose gossip-style communication for 3 reasons. First, it enables robust self-monitoring of clusters: each peer is in charge of monitoring a few random others, sharing the monitoring cost and thus ensuring load fairness [21]. 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 [21, 11].

Each $c_{ws,loc}$ manages locally the following data structures:

1. $content\text{-}list(c_{ws,loc})$: a list of the object identifiers of the content currently held by $c_{ws,loc}$. The list is used during gossip exchanges in two ways:

- current *content-summary*($c_{ws,loc}$): a summary of the current *content-list*($c_{ws,loc}$) built using a Bloom filter.
- Δ*list*($c_{ws,loc}$): a sublist that reflects the new changes in the list (i.e., object deletion or insertion) wrt. a threshold of changes (detailed later in this section)

2. *view*($c_{ws,loc}$): a partial view of *content-overlay*(ws, loc), which contains a fixed number V_{gossip} of entries, each one referring to some other $c'_{ws,loc}$. A view entry referring to a contact $c'_{ws,loc}$ contains 3 fields:
   - information about the address of $c'_{ws,loc}$ (e.g., IP address)
   - age: numeric field that denotes the age of the entry since the moment it was created (not an indication of $c'_{ws,loc}$'s lifetime)
   - *content-summary*($c'_{ws,loc}$)

Whenever $c_{ws,loc}$ gossips with $c'_{ws,loc}$, $c_{ws,loc}$ updates the entry related to $c'_{ws,loc}$ in *view*($c_{ws,loc}$) as follows: the age of $c'_{ws,loc}$ is set to zero, and a current *content-summary*($c'_{ws,loc}$) is received from $c'_{ws,loc}$; thus the age zero refers to the most recent entry status. Periodically (i.e., with period T_{gossip}), $c_{ws,loc}$ increments by 1 the age of all its view entries. Thus, a high age reflects that $c_{ws,loc}$ has not heard recently about $c'_{ws,loc}$ in order to refresh its view entry.
When $c_{ws,loc}$ joins *content-overlay*(ws, loc), *view*($c_{ws,loc}$) is initialized upon its first contact with a peer from its content overlay (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 *content-overlay*($\beta, 2$)), it is $d_{\beta,1}$ that provides F with a subset of its view, i.e., its *directory-index*($\beta, 1$); then, F's initial view will not have content summaries but will progressively fill them via gossip exchanges.
The gossip behavior of each content peer $c_{ws,loc}$ is illustrated in Algorithm 4: 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 in exchange, $gossipMsg'$ containing similar information from $c'_{ws,loc}$; $c_{ws,loc}$ creates *viewEntry*, a view entry related to $c'_{ws,loc}$, with the age 0 and the current summary of $c'_{ws,loc}$. The procedure **merge**() collects in a buffer all the entries from both the local view and the received information from $c'_{ws,loc}$, and discards the duplicates: if 2 entries related to the same contact exist, only the instance with the smallest age value is kept. Then, the procedure **select_recent**() selects the most recent V_{gossip} entries from the buffer i.e., the ones with the smallest age values, in order to limit the view size to V_{gossip}.
The passive behavior is triggered when $c_{ws,loc}$ receives a gossip messsage containing summary and view information from some content peer $c''_{ws,loc}$. Then, $c_{ws,loc}$ answers by sending back a gossip message with its own summary and view information, and updates its local view via **merge**() and **select_recent**() as described previously.
Through both active and passive behaviors of Algorithm 4, $c_{ws,loc}$ 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.

Algorithm 4 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
```

4.2.1 Directory Management

As a member of *content-overlay*(ws, loc), a directory peer $d_{ws,loc}$ is also involved in the overlay management. For this purpose, each content peer $c_{ws,loc}$ 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 age information. $c_{ws,loc}$ periodically increments the age of $d_{ws,loc}$'s entry, as it does with all its view entries. In every gossip exchange between content peers, $c_{ws,loc}$ sends its view entry related to $d_{ws,loc}$, along with its gossip message. This process spreads continuous updates about the directory peer throughout its content overlay, especially to ensure failure recovery (see Sec. 5.2).
In order to update *directory-index*(ws, loc), content peers $c_{ws,loc}$ communicate with $d_{ws,loc}$ via one-way gossip exchange, refered to as *push* and depicted in Algorithm 5. Each content peer monitors the changes, i.e., object deletions and additions, in its *content-list*($c_{ws,loc}$) noted *list* for simplicity; whenever the percentage of new changes reaches a threshold, $c_{ws,loc}$ creates Δ*list* to be pushed to $d_{ws,loc}$ (via **ex-**

Algorithm 5 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
```

tract_changes()). Then, the pushing $c_{ws,loc}$ resests to 0 its age field of $d_{ws,loc}$.

Algorithm 6 Behavior of $d_{ws,loc}$

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

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

Recall that $d_{ws,loc}$ maintains a complete view of its content overlay which is its own directory index (see Sec. 3.3). To monitor the liveness of its content peers, $d_{ws,loc}$ periodically increments the age fields of its view entries and waits for a push message as shown in Algorithm 6. When receiving a push message, $d_{ws,loc}$ updates the entry related to the pushing $c_{ws,loc}$ in its directory index, using $\Delta list$.

A directory peer also has to maintain its directory summaries, which are summaries of the directory-indexes of other directory peers. A directory peer only sends 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 threshold. This delayed propagation is warranted as [9] 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.

5. DEALING WITH DYNAMICITY

In this section, we discuss how our system deals with the dynamicity of peers wrt. failures and leaves, scale up and change of locality .

5.1 Redirection failure

Recall that a directory peer redirects a query to some content peer that stores a copy of the requested object. However, the targeted content peer may have failed or disconnected resulting in a redirection failure. In such cases, the directory peer removes the invalid directory entry and tries another redirection destination (i.e., another content or directory peer, or the web-server), until an available copy of the requested object is found.

Our system minimizes the number of query redirection failures, by maintaining directory indexes with recently updated entries. For this purpose, we exploit a feature inherent to P2P systems, the usage of *keepalive messages*, which are periodic messages sent to check links between peers. Thus, $c_{ws,loc}$ regularly sends *keepalive* messages to $d_{ws,loc}$. Upon the reception of the message $d_{ws,loc}$ resets the age of $c_{ws,loc}$'s entry in *directory-index*(ws, loc) to zero. Moreover, $d_{ws,loc}$ constantly checks the age of each directory entry and removes it if its age reaches the age limit noted T_{dead}.

5.2 Directory failure

A directory failure occurs when the directory peer either fails or leaves voluntarily. Normally, the DHT-based overlays replace the failed/departed peer, by reorganizing the DHT and redistributing the stored data accordingly [20, 16]. However, our system adopts its own replacement strategy, in order to preserve the D-ring model.

A directory peer $d_{ws,loc}$'s replacement is done by a peer from *content-overlay*(ws, loc), because these peers share the interest in the same website's content and belong to the same locality. Thus, in Figure 4, one of peers A, B, C and D becomes the new directory peer $d_{\beta,1}$ if the old one fails or leaves voluntarily. The replacing peer is assigned the same identifier as $d_{\beta,1}$, because they both belong to the same locality 1 and serve the same website β.

When a directory peer $d_{ws,loc}$ leaves voluntarily, it selects a content peer from its *content-overlay*(ws, loc) to replace it. In Example 2, assuming that the chosen content peer is A, the current $d_{\beta,1}$ transfers to A its directory; A joins D-ring as the new $d_{\beta,1}$.

When a directory peer $d_{ws,loc}$ fails, some content peers $c_{ws,loc}$ detect its failure while sending *keepalive* or push messages (recall that content peers regularly contact their directory peer to update it about their content cf. Sec. 4.2.1). Each content peer that detects the failure tries to replace $d_{ws,loc}$ as follows: it uses the common key assigned for $d_{ws,loc}$ and attempts to joins D-ring via the normal join procedure of the underlying structured overlay. If the directory position has already been appropriated by another content peer, the join message gets to the new $d_{ws,loc}$; thus the content peer that was trying to join D-ring gets acquainted with its new directory peer and informs other content peers while gossiping. The new $d_{ws,loc}$ gradually builds its directory upon receiving push messages. Meanwhile, $d_{ws,loc}$ answers first queries from its content summaries.

Subsequent to the directory replacement presented above, existing peers of the directory overlay should be informed to update their routing tables. For that, we rely on the stabilization procedures that are normally used in structured overlays [20, 16]. They will detect the old $d_{ws,loc}$'s departure and the new $d_{ws,loc}$'s presence.

A directory peer should have the best performance within its content overlay. A peer *profile* can be determined by characteristics such as stability, bandwidth and processing capacities, etc. Thus in Flower-CDN, each $c_{ws,loc}$ can monitor its profile and regularly compare it to the profile of its current directory (which can be propagated via gossip/push/keepalive exchanges). If $c_{ws,loc}$ has a better profile, $d_{ws,loc}$ steps down so that $c_{ws,loc}$ joins D-ring in its stead.

5.3 Scaling up

In the basic solution of Flower-CDN, we restrict the number of participant peers that can contribute to the system, by limiting the size of each content overlay. This is aimed at keeping content overlays at a manageable size, so that their directory peers are not overloaded with the maintenance of the overlay information. However, in this case the P2P system may attract more participant peers than the content overlay capacity. To address this case and warrant the extensive deployment of Flower-CDN to larger scale, Flower-CDN may allow more than one directory peer for each tuple (website, locality), to consecutively join D-ring. Each of them manages its own content-overlay, resulting in several content overlays for the same locality. To achieve this solution in practice, the peer ID shoud be extended by adding b extra bits at the end of it (to preserve the locality and website identification). We plan to further investigate such scenarios in the future and we adopt the basic solution in the current performance evaluation.

5.4 Updating Localities

Given that the underlying network dynamically changes, participant peers might change their locality. Thus, some peers would have to switch to another content overlay. Flower-CDN can handle such situations as it manages failures: the peer p that changes its locality, whether a directory or a content peer, naturally joins its new content overlay as a new client and then updates its directory peer about its held content. When peers from p's previous overlay contact p via gossip or query search, they are informed of this change and thus remove p from their contacts as they do with dead peers.

6. PERFORMANCE EVALUATION

We evaluate the performance of Flower-CDN through event-driven simulation using PeerSim [3]. Our performance evaluation consists mainly in quantifying the gains due to locality-awareness in Flower-CDN. Furthermore, we evaluate the price to be paid for achieving these gains, by examining the trade-off between hit ratio and gossip bandwidth consumption. For these purposes, we use the metrics below:

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

In the following, we first argue the choice of simulation parameters, then we discuss the results. Recall that in this paper we do not deal with cache issues such as cache expiration and replacement policies, for both Flower-CDN and Squirrel [10], an approach chosen for performance comparison.

6.1 Simulation Setup

PeerSim 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 5000 peers connected with links of variable latencies; the model inspired by BRITE [2] assigns latencies between 10 and 500 ms. Network localities are modeled using a landmark-based technique [15]. We use $k = 6$ localities which are non-uniformly populated. The simulation parameters are summarized in Table 1.

Given that D-ring relies on any existing structured overlay, we choose to simulate Chord for its simplicity; we adapt its key management and routing mechanisms as explained in Sec.3.1, to be able to simulate the D-ring protocol.

We compare Flower-CDN with Squirrel [10], where all participant peers are part of one structured overlay based on a traditional DHT (i.e., Chord here). Squirrel stores 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. In Squirrel, a query always navigates through the DHT and then receives a pointer to a peer that potentially has the object. We chose Squirrel because it shares some similarities with Flower-CDN wrt. the directory structure. This makes a comparison easier and at the same time allows us to see the effects of locality-based content overlays and their gossip-based management.

For our query workload we use synthetically generated data because available web traces reflect object accesses while we are interested in website accesses. Each website provides 500 objects which are requestable and cacheable (e.g., web page of 10-100 KB, though we do not model object size). Our simulation model assumes no correlation between different website communities and applies zipf distribution for object requests submitted to each single website [4].

Each experiment is run for 24 hours, which are mapped to simulation time units. Experiments start with a stable D-ring: for each couple (website, locality), there is one directory peer with an empty directory. Although we use $|W| = 100$ websites in the construction of D-ring, we restrict the query generation to 6 websites of W. Content overlays 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 [2]. For each query intended to

[2]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 1: Simulation Parameters

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

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 overlay. When a content overlay reaches its maximum size noted S_{co} (set by default to 100), no new clients may join the overlay. With this, we avoid that the directory peer is overloaded with the maintenance of the content overlay information. In consequence, the content overlays of a given website evolve at different rythms and sizes. Eventually, we should have up to $N = |W| * k * S_{co}$ participant peers. However, since we are only looking at 6 active websites, $N = |W| * k + (6 * k * S_{co})$ which is equal to 4200 participant peers in the current configuration.

We assume that a content peer has enough storage potential to avoid replacing its 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. 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 (i.e., nb-ob), thus we set *summary size* according to the analysis in [9], 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. Sec. 4.2.1). *View size* V_{gossip} and *gossip period* T_{gossip} comply wih the definitions given in Sec. 4.2 while *gossip length* L_{gossip} refers to the size of the view subset exchanged in a gossip round. To correctly tune the gossip parameters and adapt them to our protocol, we tested their variation in the experiments presented in Sec. 6.2.

6.2 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* (i.e., L_{gossip}), *gossip period* (i.e., T_{gossip}) and *view size* (i.e., V_{gossip}). We also varied *push threshold*; but we do not show the results which illustrate similar performance (i.e., almost same gains and same trade-off) for different values of *push threshold* (0,1; 0,5; 0,7). In each experiment, we vary one of the 3 gossip parameters (L_{gossip}, T_{gossip}, V_{gossip}) and fix the two other parameters; then after 24 simulation hours, we collect the results for each parameter value. Table 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: Effect of Gossip Bandwidth Variation

L_{gossip}	Hit ratio	Background BW
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 BW
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 BW
20	0.78	74 bps
50	0.86	74 bps
70	0.863	74 bps

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

Table 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.

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

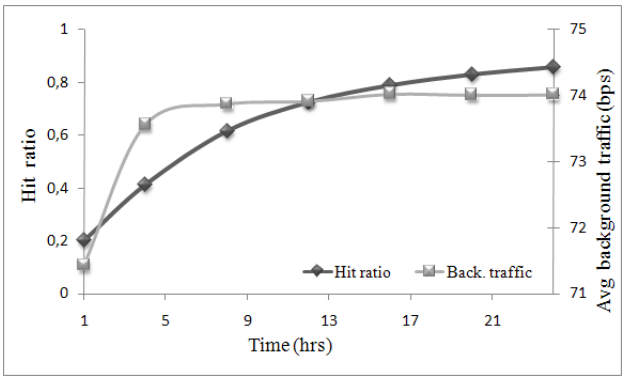


Figure 5: Trade off between hit ratio and bandwidth in Flower-CDN

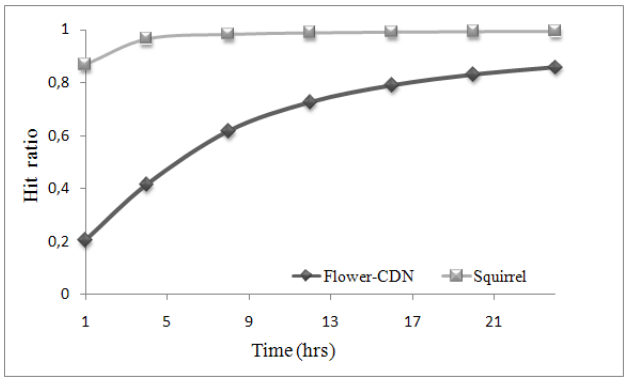


Figure 6: Comparing hit ratio in Flower-CDN and Squirrel

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 5 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 content overlays 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.

6.3 Hit ratio

The following results compare Squirrel and Flower-CDN wrt. hit ratio. Figure 6 shows that the hit ratio eventually converges to 1 for both Squirrel and Flower-CDN, but convergence takes longer for Flower-CDN given that the search space is partitioned into content overlays. In fact, after 24 hours, the hit ratio of Flower-CDN is less than that of Squirrel by 13%. This difference can be justified by the following. Once a copy of an object o_{ws} is stored in Squirrel, 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. 3.3), in order to achieve locality-awareness. Moreover, an object o_{ws} becomes available in *content-overlay*(ws, loc) only after a peer from the overlay has submitted a query for o_{ws}. Thus, once a copy of o_{ws} is available in each content-overlay, Flower-CDN achieves a hit ratio similar to Squirrel wrt. o_{ws}.

In general, a smaller hit ratio means less queries are served from the P2P and instead go to the server. This is not bad as long as the server is not overloaded. Furthermore, as we will see in the next section, Squirrel 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.

6.4 Locality-awareness

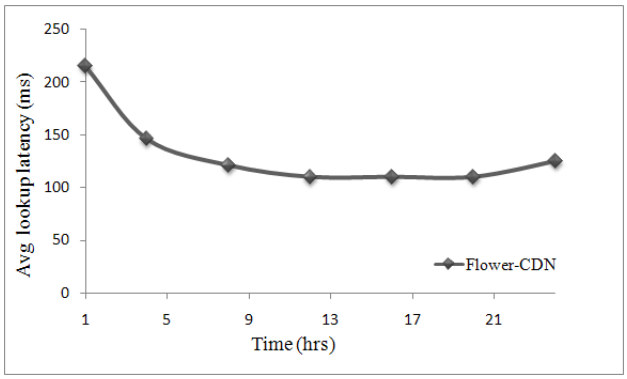
In this set of experiments, we evaluate the gains due to locality-awareness in Flower-CDN, by measuring lookup latency and transfer distance. Again we compare with Squirrel which does not leverage locality-awareness.

The first experiment, illustrated in Figure 7, measures the lookup latency. Figure 7(a) 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 7(b) shows the latency distribution of queries for both solutions: 87% of our queries are resolved within 150 ms while 61 % of Squirrel'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 content overlay, achieving very short delays. In contrast, Squirrel 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.

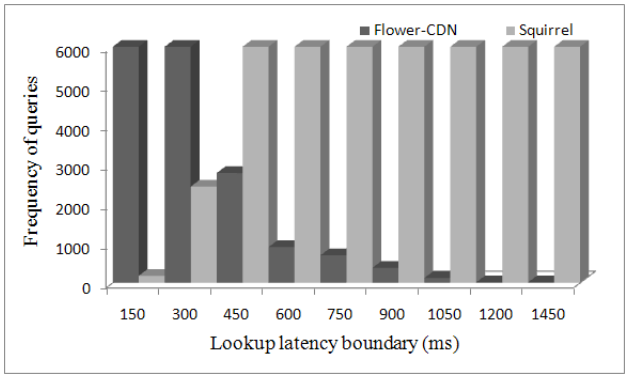
The second experiment focuses on transfer distance. We are interested in this metric because it has a significant impact on network usage and object download speed which affects response times perceived by users. At the underlying network level, higher distances generally involve more intermediate links 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 8(a) 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 8(b) 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 Squirrel's queries. Thus, Flower-CDN provides excellent results by reducing the average tranfer distance by a factor of 2 in comparison with Squirrel. Flower-CDN ensures data transfers over short distances, which limits the network load and reduces the response times perceived by users.

6.5 Discussion

We learnt two main lessons through our experiments. First, the usage of gossip when confined in content overlays ap-

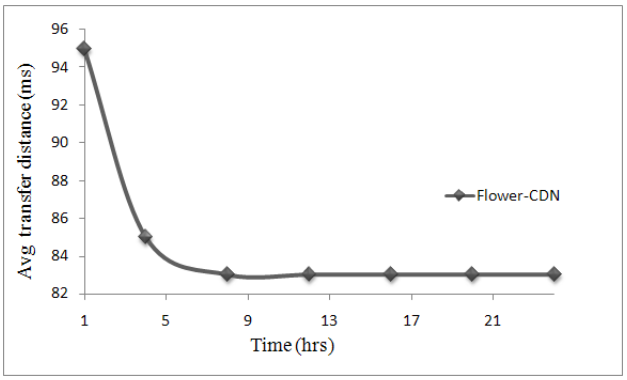


(a) Lookup latency variation in Flower-CDN

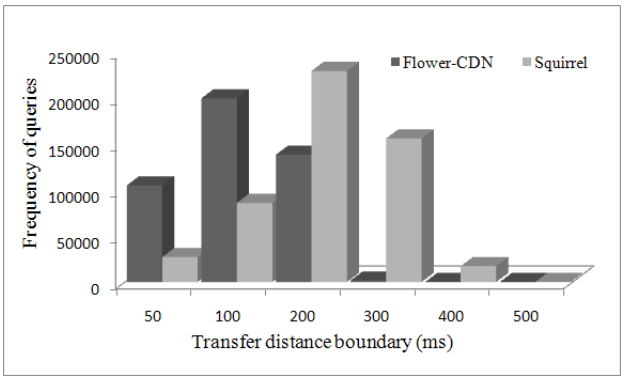


(b) Lookup latency distribution in Flower-CDN and Squirrel

Figure 7: Lookup latency



(a) Transfer distance variation in Flower-CDN



(b) Lookup latency distribution in Flower-CDN and Squirrel

Figure 8: Transfer distance

pears to be quite efficient with an acceptable overhead in terms of bandwidth consumption. Moreover, the bandwidth overhead could be adapted to the available network resources by tuning the gossip parameters, while respecting hit ratio requirements. Second, combining structured and gossip-based overlays with locality-aware considerations proved to be quite performing especially in performing fast searches (i.e., low lookup latency) and finding close-by results (i.e., low transfer distance). In Flower-CDN, D-Ring is only used to provide a first reliable access, for new participant peers wrt. a content overlay. Afterwards, they become part of this content overlay and direct subsequent queries directly to the content-overlay instead of D-ring. In contrast, Squirrel relies on the DHT-based overlay for every single query leading to high lookup latencies. Furthermore, Squirrel'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 Squirrel. Moreover, although not measured in our experiments, the high lookup rates very likely also lead to higher loads on DHT participants.

7. RELATED WORK

Several approaches exist that can be used to store web content on peer nodes. Many of these approaches rely on DHT to achieve fast lookup. Built over Pastry, Squirrel [10] proposes two strategies to be applied for organization-wide networks. The first strategy (*home-store*) replicates web objects at peers with ID numerically closest to the hash of the URL of the object. Thus, queries find the peer that has the object by navigating through the DHT. To deal with highly popular objects, object replicas are progressively put along neighbors as the number of requests increases. The second strategy (*directory*) stores at the peer identified by the hash of the object's URL a small directory of pointers to recent downloaders of the object. In this case, a query first navigates through the DHT and then receives a pointer to a peer that potentially has the object. PoPCache [14] proposes an approach similar to the home-store strategy while refining the replication technique along neighbors and computing the number of replicas per object as a function of its popularity. Backslash [18] applies similar strategies as Squirrel by inserting an object replica or its related directory at peers identified by the DHT. However, there are two main drawbacks in the DHT-based approaches described above. First, unless using a locality-aware overlay combined with proactive replication, they serve requests from random physical locations, which may deteriorate the user-perceived latency and consume considerable network resources. In contrast, Flower-CDN relies on a locality-aware directory that directs each query according to the physical location of the client. Second, the DHT-based approaches force the peers to store objects that they have not requested by themselves, while our approach exploits the interests of clients.

Proofs [19] uses an ustructured overlay in which peers' neighborhoods are continuously changing. This provides each peer with a random view of the system for each search operation. Peers keep their requested objects and can then provide them to other participants. To locate one of the

object replicas, a query is flooded to a random subset of neighbors with a *fixed time-to-live* (TTL) i.e., the max number of hops. However, searches for not-so popular objects induce heavy traffic overheads and high user-perceived latency, while Flower-CDN can locate any object within a bounded number of hops. Moreover, neither the overlay nor the search incorporate any information about the underlying network topology to forward queries to close results.

OLP [17] adopts a hybrid architecture where the website plays the role of a super-peer: it maintains a directory of peers to which its objects have been transferred in the past and manages the redirection of queries. To avoid redirection failures in a P2P dynamic environment, OLP models the object lifetime and proposes a strategy that guides the website's selection for peers (i.e., to choose the peer to which the query should be redirected). Compared to Flower-CDN where the P2P network shares the redirection workload with the server, the redirection in OLP may overload the server in case of intense flash crowds. Moreover, redirection in OLP does not take into account the physical locations of object replicas. CoopNet [13] also uses a hybrid architecture rooted at the web-server. After receiving a request, the web-server sends a list of nearby peers to the client. CoopNet tries to avoid the server redirection by creating small groups of clients. However, it does not elaborate a well-defined and decentralized structure to support searches within groups. Moreover, CoopNet does not deal with dynamic aspects such as detection of peer failures and leaves.

A two-layered overlay has also been used in [12], although neither in the context of caching or locality awareness. The authors build a DHT, where each node is a cluster of peers, whereby the cluster itself is again structured as a DHT.

8. CONCLUSION

In this paper, we proposed Flower-CDN, an interest- and locality-aware P2P CDN, that enables any under-provisioned website to efficiently distribute its content, with the help of the community interested in its content. Flower-CDN combines efficient DHT indexing to provide fast lookup with gossip robustness for replica distribution and self-monitoring. The basic idea is to exploit peer interests and localities in order to cluster participant peers in content overlays and to build a P2P directory service via D-ring. D-ring relies on a novel DHT mechanism that can be easily integrated into existing structured overlays. We proposed to use gossip-based algorithms to spread accurate information through content overlays and to robustly maintain D-ring and content overlays in face of churn. Through simulation experiments, Flower-CDN proved to be quite performing especially in performing fast searches and finding close-by results. Furthermore, gossip incured acceptable overhead in terms of bandwidth consumption, which could be adapted to the available network resources and hit ratio requirements.

We are pursuing this work in several directions. We are empirically analysing the behavior of Flower-CDN in presence of churn. We are also investigating the scalabilty of a content overlay by increasing the number of its directory peers. Finally, we plan to explore consistency aspects, in particular, cache expiration and replacement policies.

9. ACKNOWLEDGMENTS

We would like to thank Anne-Marie Kermarrec, Davide Frey and Vincent Leroy for their insightful discussions.

10. REFERENCES

[1] Akamai. http://www.akamai.com.
[2] Brite. http://www.cs.bu.edu/brite/.
[3] Peersim p2p simulator. http://www.peersim.sourceforge.net.
[4] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *INFOCOM*, 1999.
[5] F. Dabek, B. Zhao, P. Druschel, and J. Kubiatowiczet. Towards a common api for structured P2P overlays. In *IPTPS*, 2003.
[6] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *PODC*, 1987.
[7] M. El Dick, V. Martins, and E. Pacitti. A topology-aware approach for distributed data reconciliation in P2P networks. In *Euro-Par*, 2007.
[8] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulieacute. Epidemic information dissemination in distributed systems. *IEEE Computer*, 37(5), 2004.
[9] Li Fan, Pei Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *SIGCOMM*, 1998.
[10] S. Iyer, A. I. T. Rowstron, and P. Druschel. Squirrel: a decentralized P2P web cache. In *PODC*, 2002.
[11] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Middleware*, 2004.
[12] N. Ntarmos and P. Triantafillou. Aesop: Altruism-endowed self-organizing peers. In *DBISP2P*, 2004.
[13] V. N. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *IPTPS*, 2002.
[14] W. Rao, Lei Chen, Ada Wai-Chee Fu, and YingYi Bu. Optimal proactive caching in P2P network: analysis and application. In *CIKM*, 2007.
[15] S. Ratnasamy, M. Handley, R. M. Karp, and S.Shenker. Topologically-aware overlay construction and server selection. In *INFOCOM*, 2002.
[16] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale P2P systems. In *Middleware*, 2001.
[17] Y.-S. Ryu and S.-B. Yang. An effective P2P web caching system under dynamic participation of peers. *IEICE Transactions*, 88-B(4), 2005.
[18] T. Stading, P. Maniatis, and M. Baker. P2P caching schemes to address flash crowds. In *IPTPS*, 2002.
[19] A. Stavrou, D. Rubenstein, and S. Sahu. A lightweight, robust P2P system to handle flash crowds. In *ICNP*, 2002.
[20] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable P2P lookup service for internet applications. In *SIGCOMM*, 2001.
[21] S. Voulgaris, D. Gavidia, and M. van Steen. Cyclon: Inexpensive membership management for unstructured P2P overlays. *J. Network Syst. Manage.*, 13(2), 2005.