Location-Aware Index Caching and Searching for P2P Systems

Manal El Dick, Esther Pacitti, Patrick Valduriez

Location-Aware Index Caching and Searching for P2P Systems

Manal El Dick, Esther Pacitti, and Patrick Valduriez

ATLAS Group, INRIA and LINA, University of Nantes, France
{manal.el-dick,esther.pacitti}@univ-nantes.fr,patrick.valduriez@inria.fr

Abstract. Unstructured P2P networks remain widely deployed in file-sharing systems, due to their simple features. However, the P2P traffic, mainly composed of repetitive query messages, contributes the largest portion of the Internet traffic. The principal causes of this critical issue are the search inefficiency and the construction of the P2P overlay without any knowledge of the underlying topology. In order to reduce the P2P redundant traffic and to address the limitations of existing solutions, we propose a solution that performs index caching and efficient query routing while supporting keyword search. We aim at improving the probability of finding available copies of requested files by leveraging file replication. In addition, our scheme tries to direct queries to close results, by using topological information in terms of file physical distribution. We believe that the traffic can be significantly reduced and the user experience ameliorated in terms of faster downloads, with minimum overhead.

1 Introduction

Despite the emergence of sophisticated overlay structures, unstructured P2P systems remain highly popular and widely deployed. They exhibit many simple yet attractive features, such as low-cost maintenance and high flexibility in data placement and node neighborhood. Unstructured P2P systems are particularly used in file-sharing communities due to their capacity of handling keyword queries i.e. queries using some keywords instead of the whole filename. However, some studies [13] observed that the P2P traffic is mainly composed of query messages and contributes the largest portion of the Internet traffic.

The principal cause of the heavy P2P traffic is the inefficient search mechanism, blind flooding, which is commonly employed in unstructured P2P networks. Many researchers have focused on this critical issue that may compromise the benefits of such systems by drastically limiting their scalability. In fact, inefficient searches cause unnecessary messages that overload the network, while missing requested files. Several analyses [5, 7, 14] found the P2P file-sharing traffic highly repetitive because of the temporal locality of queries. They actually observed that most queries request a few popular files and advocated the potential of caching query responses, to efficiently answer queries without flooding over the entire network (a query response holds information about the location

of the requested file).
However, searches in general, suffer significantly from the dynamic nature of P2P systems where nodes are run by users with high autonomy and low availability. In fact, it is rather impossible to ensure the availability of a single file copy and thereby to satisfy queries for this file. In P2P file sharing, a node that requested and downloaded a file, can provide its copy for subsequent queries. As a consequence, popular files, which are frequently requested, become naturally well-replicated [2]. Hence, search techniques should leverage the natural file replication to efficiently find results with minimum overhead.

Another important factor that aggravates the traffic issue, consists in constructing P2P overlays without any knowledge of the underlying topology. A typical case that illustrates this problem is the following: a query can be directed to a copy of the desired file which is hosted by a physically distant node, while other copies may be available at closer nodes. Hence, file download can consume a significant amount of bandwidth and thereby overload the network. In addition, the user experience dramatically degrades due to the relatively high latency perceived at transfer.

Our work is inspired by DiCAS [15], an index caching and adaptive search algorithm. In DiCAS, query responses are cached, in the form of file indexes, in specific groups of nodes based on a specific hashing of the filenames. Guided by the predefined hashing, queries are then routed towards nodes which are likely to have the desired indexes. However, DiCAS is not optimized for keyword searches which are the most common in the context of P2P file sharing. Moreover, caching a single index per file does not solve the problem of file availability given the dynamic nature of P2P systems, while it may overload some nodes located by previous queries. DiCAS also lacks of topological information to efficiently direct queries to close nodes.

Aiming at reducing the P2P redundant traffic and addressing the limitations of existing solutions, we propose a solution that leverages the natural file replication and incorporates topological information in terms of file physical distribution, when answering queries. To support keyword searches, a Bloom filter is used to express keywords of filenames cached at each node, and is then propagated to neighbors. A node routes a query by querying its neighbors' Bloom filters. To deal with issues concerning availability and workload, a node caches several indexes per file along with topological information. As a consequence, a node answers a query by providing several possibilities, which significantly improves the probability of finding an available file. In addition, based on the topological information, we expect that queries are satisfied in a way that optimizes the file transfer and thus the bandwidth consumption.

The rest of this paper is organized as follows. Section 2 examines the different kinds of approaches aiming at reducing the unnecessary P2P traffic. Section 3 introduces the background of our work and defines the main concepts on which we rely. Section 4 describes in details our contributions. Section 5 discusses the trade-off between the expected benefits of our approach and the resulted overhead. Section 6 concludes.

2 Related Work

Many studies have focused on reducing P2P traffic in unstructured systems, while different types of approaches have been exploited. Our work mainly falls into three categories: search-based, caching-based and topology-based.

Search-based approaches [3, 9, 16], basically consist of maintaining information about neighbors, and using them to route queries. A node can either hold summarized lists of files stored at its neighbors or statistics based on previous searches (e.g. number of results returned from each neighbor). Each time the node needs to forward a query, it selects a subset of its neighbors according to the maintained information. These techniques do not incorporate location awareness, typically by directing a query to a physically distant file, which consumes high bandwidth at transfer and contributes to increasing the traffic.

Caching-based approaches aim at caching content (e.g. files) or indexes (e.g. file locations) to limit the extent of flooding, when searching for some content. Index caching is performed in [11, 14, 15], where query responses are cached in the form of indexes, on their way back to the originator. Centralized caching [11] at the gateway of an organization does not exploit nodes resources and is likely to produce bottlenecks. The authors in [14] propose that each node caches all passing query responses, which results in large amount of duplicated and redundant cached among neighboring nodes. In DiCAS [15], file indexes are cached in specific groups of nodes and queries are selectively routed. Similarly to the search-based approaches, DiCAS do not handle the location awareness issue. Moreover, the well-replication of popular files is not taken into advantage, which results in missing available file copies or concentrating the load on a few nodes holding copies. Thus, further searches may be needed and more messages injected into the traffic.

Topology-based approaches focus on optimizing the P2P overlay topology by exploiting information about the underlying network. The goal is to construct an overlay that reflects the underlying topology. Some proposals such as [10] group nodes into clusters based on network distance or IP addresses. Others [8, 12] try to improve nodes neighborhood in terms of proximity. This category of solutions enable location-aware searches without improving their efficiency in finding results.

3 Problem Definition

In this section, we introduce the main terms and concepts in order to define the problem. First, we briefly describe unstructured P2P file-sharing systems (Gnutella). Second, we present DiCAS, a protocol that performs index caching over Gnutella and which inspired us to build our protocol. Third, we give the problem statement.

3.1 P2P File-Sharing Systems - Case of Gnutella

Gnutella is an unstructured P2P network which is a logical overlay built on top of the Internet. Each node joins the network by establishing logical links to randomly chosen nodes, referred to as its neighbors. Normally, the neighborhood of a node is set without knowledge of the underlying topology. Participating nodes are highly dynamic and autonomous, failing or leaving the network at any moment.

In such P2P systems, nodes share files of any type specified by the application. We note F, a file object and f its filename. Whenever there is no ambiguity, we may use f or F automatically. Filenames are broken into keywords following predefined rules and are stored at nodes accordingly. Nodes request files by submitting queries to the P2P network. A query q is commonly expressed by some keywords related to the queried filename instead of the whole filename.

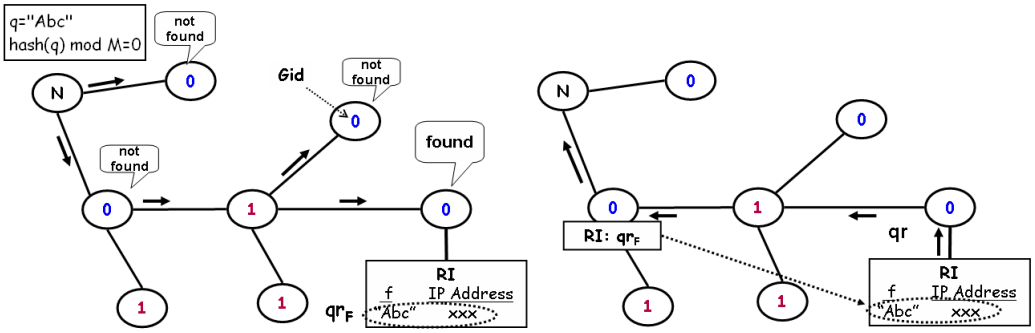
Query routing is done by blindly flooding q over the P2P network and is bounded by a fixed *time-to-live*, TTL, i.e. the maximum number of hops. Query responses qr follow the reverse path of their corresponding q, back to the requesting node. q can be satisfied by any file F that has a filename f containing all keywords of q. Thus, a query response qr_F contains the filename f and the location of a copy of the file F, which refers to the IP address of a node i located by the query and providing F (node i is noted $P_{F,i}$). The requesting node downloads F via direct connection (i.e. out of the P2P network) and then becomes a provider node $P_{F,i}$ for subsequent requests for F.
If a file F is requested frequently, then as more requesting nodes download F, there will be many copies of F and thereby many providers $P_{F,i}$ within the system. As shown in [2], most queries are for popular files F, and thus for well-replicated files F.

3.2 Basic DiCAS

DiCAS [15] is a distributed index caching and adaptive search algorithm designed for P2P systems like Gnutella. File locations are cached, in the form of indexes, in specific groups of nodes by matching filenames against group Ids. Hence, index caching consists of caching a query response qr in the form of a file index by nodes along q reverse path. An index of F contains thus the filename f and the IP address of some node $P_{F,i}$. Therefore, each node i maintains an index cache called *response index* and noted RI_i.

At join, node i randomly chooses a group Id noted Gid_i ($Gid_i \in [0\,..\,M-1]$ with M a system parameter). Gid_i *matches* a filename f if the following condition is satisfied: $Gid_i = \text{hash}(f) \bmod M$.
Rather than flooding, group Ids are used to route a query q, towards nodes that have a high probability of caching indexes of files satisfying q. A query q is thus sent to neighbors with a matching Gid wrt. q. If no such neighbors are found, q is sent to a highly connected neighbor.
Another essential functionality of Gids is defining a selective policy for index caching, in order to avoid redundant indexes among neighbors. A query response

qr_F is only cached in RI of nodes with matching Gid wrt. f.
Figure 1 shows a typical example of DiCAS algorithm. The query q submitted by node N_q is matched by $Gid = 0$, according to the predefined hashing. Thus, q is sent, whenever it is possible, to nodes with $Gid = 0$, until an index of some file F that satisfies q is found. On its way back, qr_F is cached in matching nodes wrt. f i.e. with $Gid = 0$.



(a) Query routing. (b) Query reponse path.

Fig. 1. A typical example of DiCAS: $q =$ "Abc" and $\text{hash}(q) \bmod M = 0$.

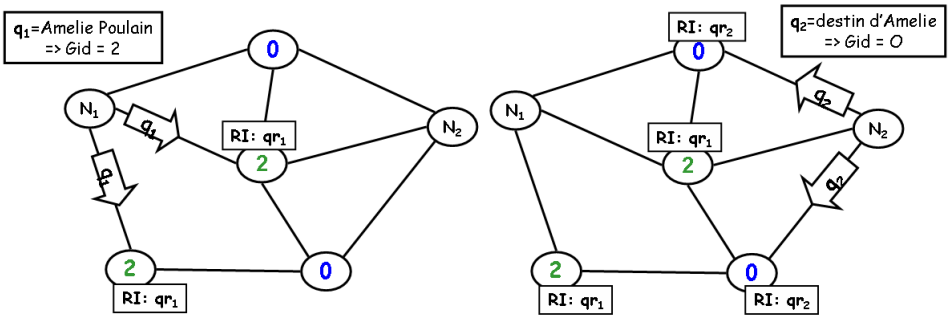
3.3 DiCAS Limitations

We have identified two main weaknesses in DiCAS approach when applied in real P2P-file sharing systems, well-known for the spontaneous and dynamic nature of their participants. In the following, we describe the two limitations that we address in our work.

DiCAS with Keyword Search. The authors of DiCAS pointed out a weakness in their initial solution, concerning keyword queries and they proposed an optimization. In fact, indexes of some file F are cached in nodes i with matching Gid wrt. the filename f (i.e. $Gid_i = \text{hash}(f) \bmod M$) while a query q for F is forwarded to nodes j with matching Gid wrt. q (i.e $Gid_j = \text{hash}(q) \bmod M$). Given that common queries are expressed using some keywords of the filename, q is sent to nodes with unmatched Gid, i.e. nodes that most probably do not have indexes of F in their response indexes (i.e. $q \neq f$, then $Gid_i \neq Gid_j$).

The proposed optimization consists in caching indexes based on query keywords instead of the whole filename. To illustrate the new strategy, let us consider two consecutive queries q_1 and q_2 that both request the file F named "le fabuleux destin d'Amelie Poulain": q_1 =Amelie Poulain and q_2 = Destin Amelie. The example is shown in Fig. 2. Query q_1 submitted by N_1 is forwarded to nodes

with Gid = 2 ($\text{hash}(q_1) \bmod M = 2$). Then, after searching the P2P network and finding satisfying responses, responses qr_1, on their way back to N_1, are cached in nodes with Gid = 2. After a while, N_2 submits a different query q_2 for the same file F ($q_1 \neq q_2$). q_2 is forwarded to nodes with Gid = 0 ($\text{hash}(q_2) \bmod M = 0$) and thus misses indexes of F cached in neighbors with Gid = 2. Similar to qr_1, responses qr_2 are cached in nodes with Gid = 0.
Obviously, this optimization causes a larger amount of duplicated and redundant cached indexes. Furthermore, DiCAS looses some of its efficiency in reducing the P2P traffic because many results available on the query routing path are missed and thus more messages are flooded to locate the queried file.



(a) First query q_1. (b) Second query q_2.

Fig. 2. An example of DiCAS limitation in supporting keyword queries: q_1 and q_2 requesting file F named "le fabuleux destin d'amelie poulain'

Weak Exploitability of Natural File Replication. In DiCAS, a node n caches in its RI only one index of a given file F matching its Gid and referring to some provider node $P_{F,1}$. All other indexes of F which refer to other providers $P_{F,i}$ and that node n may overhear afterwards, are not taken into account. Subsequently, node n redirects to $P_{F,1}$ all passing queries that could be satisfied by F.

Three consequences that may result from the above approach, illustrate the inefficiency of DiCAS caching policy:

- The cached index of F may expire after a small time period because $P_{F,1}$ may disconnect or discard its copy of F at any moment. Thus, queries directed to $P_{F,1}$, remain unsatisfied, which leads to further and repetitive searches i.e. extra messages disseminated over the network.
- Given the temporal locality of P2P queries, $P_{F,1}$ may become quickly overloaded since n may receive several queries for F.
- $P_{F,1}$ may be physically distant from requesting nodes. As a result, more bandwidth is consumed and a relatively high latency is perceived by the corresponding users at the file download.

3.4 Problem Statement

We formally define our problem as follows. Given an unstructured P2P file-sharing system with the characteristics and terms described in Sec. 3.1, let:

- N be the set of all participant nodes n, such that each node n has a Gid. Then based on DiCAS, $\forall n \in N$; RI_n caches query responses qr_F such that $\text{hash}(f) \bmod M = Gid_n$.
- PF be the set of popularly shared files F, such that each F may be provided by multiple nodes $P_{F,i} \in N$ (as discussed in 3.1).
- PQ be the set of common queries q which normally request files of PF while using some keywords of the filenames.

Our objectives are the following:

- Routing q efficiently towards nodes n with indexes of F in RI_n
- Exploiting replication of F i.e. several $P_{F,i}$ as well as their physical locations and incorporating this information in RI_n

4 Efficient Caching and Searching in P2P File-Sharing

We now propose two main improvements that address the problems identified above. As a first improvement, we propose an efficient support for keyword queries. A second improvement consists of exploiting location awareness and file replication.

4.1 Efficient Support for Keyword Search

We assume that index caching is based on the whole filename as in basic DiCAS. To remediate the problem of keyword searches, we use a Bloom filter to express filenames' keywords in a response index and to send the filter to neighbors.
A Bloom filter [1, 6] is a simple space-efficient data structure for representing a set of elements, in order to support membership queries. When querying a Bloom filter, it never returns false negatives but it may lead a false positive when it suggests that an element belongs to the set even though it does not.

Each node n maintains a Bloom filter, noted BF_n, that represents the set of keywords of all cached filenames in RI_n. Whenever node n overhears a response qr_F such that f matches Gid_n, it caches it in RI_n similarly to basic DiCAS (Sec. 3.2), and then inserts each keyword kw_i of f as an element of BF_n. Moreover, we define that a query q matches BF_n if $\forall kw_i \in q$ is member of BF_n.

Neighboring nodes exchange their own group Ids as well as their bloom filters. Thus, node n stores its own RI_n and BF_n as well as its direct neighbors' Gid and BF. To forward a query q, node n checks first its neighbors' BF. Node n sends q to neighbors with matched BF wrt. q. If no such neighbors, query q is sent to neighbors with matched Gid wrt. q or to a highly connected neighbor as a last resort.

A Bloom filter BF_n is built incrementally as new filenames are inserted in RI_n and existing ones discarded. Updating BF_n locally is done automatically since membership changes are supported by Bloom filters. Copies of BF_n held by neighbors of node n must also reflect the content of RI_n, thus n periodically propagates updates of BF_n to neighbors after a threshold i.e. a given percentage of changes in RI_n.

4.2 Exploiting Location Awareness and File Replication

In order to provide more accurate and efficient responses to queries, we introduce location awareness in response indexes, in terms of information about the physical location of the file provider. In addition, we exploit file replication, based on the fact that a node which has recently requested a file F is likely to have it and can thereby serve subsequent requests for F.
In the following, we first describe our physical locations basis then we present our location-aware approach. Finally, we propose a technique to control the cache size at a node.

Physical Locations We assume that participant nodes are physically dispersed and can be grouped based on their locations. To model these physical locations, we use a common technique used in other works such as [4, 12] that relies on the existence of a set of well-known machines spread across the Internet, called *landmarks*. A node n can determine its physical network distance to each landmark by measuring its latency to that landmark. An ordering of the set by increasing latencies reflects the physical location of node n. Thus, physically close nodes are likely to produce the same ordering. We thereby associate to each possible ordering a location Id noted $locId$. Then, at its arrival, each node computes its own locId based on its latency measurements.

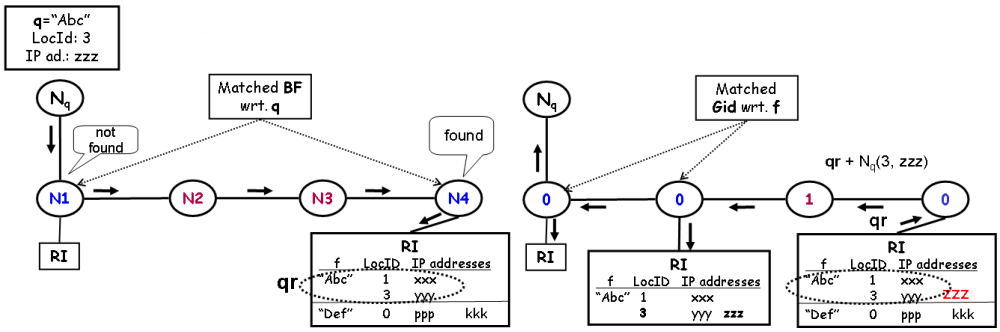
Location-Aware Caching and Search In the following, we consider the example shown in Fig. 3 to illustrate our new approach. The response index of a node n may hold for some cached filename f, several provider addresses $P_{F,i}$ and their locIds. Hence, n can selectively answer a query q that can be satisfied by F, according to the locId of the querying node (noted n_q). In our example, $N4$ has in its RI the IP addresses of 2 nodes that can provide the file "Abc", one associated to $locId = 1$ and another to $locId = 3$. As also shown in Fig. 3, A query q and its responses qr_F should contain both the IP address and the locId of the node n_q, which will be considered as a new provider by nodes intercepting the responses.
Algorithm 1 describes how each node n visited by some query q, processes and routes q. First, n checks its response index for a filename that can satisfy q. In the case where such a filename is found, n tries to find a provider $P_{F,i}$ with the same locId as n_q (i.e. lines 6 to 10). Node n's response also includes IP addresses of some other providers of F with their associated LocIds, to guarantee that n_q will find an available copy of F (i.e. lines 11 to 13). In the other case, node n

queries its neighbors' Bloom filters and as a last resort their Gids (i.e. lines 15 to 25).
In Fig. 3, the query q is routed to neighbors with matching BF wrt. q whenever it is possible, until a satisfying filename f is found in the response index of some node N_4. The query response sent by node N_4 is constituted of the entry $(3, yyy)$ corresponding to the locId of N_q and another random entry $(1, xxx)$ for availability reasons. Node N_4 then adds the entry $(3, zzz)$ of node n_q. On its way back, the response qr is cached, with the information about N_q, in nodes with matching Gids wrt. filename f.

Algorithm 1 Routing(q, n_q) at each visited node n

1: $RI_n \langle f \rangle$: LocId entries in RI_n corresponding to filename f
2: $RI_n \langle f \rangle \langle locId \rangle$: IP addresses in RI_n corresponding to some specific $locId$ and f
3: qr_F: query response that n may generate.
4: **Receive(**$q, \langle n_q.IPaddress, n_q.locId \rangle$**)**
5: *// Check local response index*
6: **if** $\exists f \in RI_n$ that satisfies q **then**
7: **if** $(RI_n \langle f \rangle$.contains$(n_q.locId))$ **then**
8: *// Add to qr_F the addresses of $P_{F,i}$ with the same locId as n_q*
9: qr_F.add$(RI_n \langle f \rangle \langle locId \rangle)$
10: **end if**
11: *// Add to qr_F some random addresses of $P_{F,i}$*
12: qr_F.randomAdd$(RI_n \langle f \rangle \langle \rangle)$
13: **Send back**(qr_F) and **Quit** the algorithm
14: **end if**
15: *// Check neighbors' Bloom filters*
16: **for** each neighbor i **do**
17: **if** $(BF_i$.matches$(q))$ **then**
18: Send(q) to neighbor i
19: **end if**
20: **end for**
21: // Check neighbors' Gids
22: **for** each neighbor i **do**
23: **if** $(Gid_i$.matches$(q))$ **then**
24: **Send**(q) to neighbor i
25: **end if**
26: **end for**

Controlling the Cache Size The cache size refers in our solution to the temporary storage space allocated for the response index. Given the inherent heterogeneity of P2P systems, each node contributes with a different amount of memory. The maximal amount of memory that a node can invest is denoted by $maxMemo$. A node detects a storage excess when its cache size surpasses its



(a) Query routing. (b) Query response path.

Fig. 3. An example of our complete approach

maxMemo. In that case, the node needs to discard some of its response index content.
To perform an efficient cleanup, the node determines the excess entries which has the lowest expected utility. It proceeds with the following cleanup consisting of two phases:

1. If for a given locId of a filename, there are more than two IP addresses cached, only the two most recently cached entries are kept while the rest is discarded.
2. If after the previous cleanup the cache size remains in excess, the node searches for the filename with the largest number of locId entries and discards the ones that are the least recently cached.

The above strategy ensures a level of freshness in the response index while controlling its storage size.

5 Discussion

In this section, we discuss the trade-off between the potential benefits of our proposed techniques and their costs in terms of traffic and storage requirements.

5.1 Keyword Searches with Bloom Filters

A node stores its Bloom filter as well as its neighbors' filters. A Bloom filter provides a trade-off between its memory requirements and its false positive ratio. Hence, given a response index with 50 filenames of 3 keywords in average, an optimal representation by a Bloom filter needs a negligeable amount of memory, varying between 0.15 and $0.6\,KB$.[1] Since the average connectivity degree d is

[1] To support changes, 4-bit counts are added to a BF [6](i.e. extra memory $= 0.6\,KB$). These counts are only stored locally and not propagated to neighbors

equal to 3 in Gnutella, then the average storage space allocated for Bloom filters at a node is equal to $(3+1) * 0.15 = 0.6\,KB$, which is very small.
In addition, in [6], it has been shown that such optimal representation can have almost the same accuracy as when querying straightforward the represented set, that is in our case the set of filename keywords of the corresponding response index. Thus, when a node queries its neighbors' Bloom filters, we expect a significantly more efficient routing than with Gids.

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

5.2 Location-Awareness in Response Index and Query Responses

Recall that our approach involves caching multiple indexes per file. Hence, a query response can consist of several indexes pointing to different providers of the same file. In contrast, DiCAS is limited to one index per query response. Obviously, our approach improves the availability of finding an available copy of the desired file, by providing several possibilities instead of one. Moreover, based on the results of other location-aware works [4, 12], we expect that the location-awareness of query responses limits the wasted bandwidth at file transfer as well as the user perceived latency.

Concerning the storage requirements due to the extension of the response index, we have proposed a strategy to bound the cache size at each node (Sec. 4.2).

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

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

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

6 Conclusion

traffic and improve the user experience. In this paper, we identified limitations of existing solutions. Then, we proposed a solution that leverages typical properties of P2P-file sharing environments such as file replication, as well as useful information about the underlying topology. To efficiently route queries towards desired results, our approach caches file indexes in groups of nodes based on the filenames, while efficiently supporting keyword searches.

In addition, it aims at improving the probability of finding available copies of requested files, and at directing queries to close results. Through discussion and comparison with similar works, we strongly believe that the traffic can be significantly reduced and the user experience ameliorated in terms of faster downloads, with minimum overhead. In the immediate future, we intend to explore the trade-offs of our solution, in more detail, through simulation.

References

1. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM, 1970.
2. Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., Shenker, S.: Making Gnutella like P2P systems scalable. Proc. of ACM SIGCOMM, 2003.
3. Crespo, A., Garcia-Molina, H.: Routing indices for P2P systems. Proc. of ICDCS, 2002.
4. El Dick, M., Martins, V., Pacitti, E.: A topology-aware approach for distributed data reconciliation in P2P networks Proc. of Euro-Par, 2007
5. Evangelos P. Markatos: Tracing a large-scale P2P System: an hour in the life of Gnutella. Proc. of CCGRID, 2002.
6. Fan, L., Cao, P., Almeida, J.: Summary Cache: a scalable wide-area web cache sharing protocol. Proc. of SIGCOMM, 1998
7. Leibowitz, N., Bergman, A., Ben-Shaul, R., Shavit, A.: Are file swapping networks cacheable? Characterizing P2P traffic. 7th International Workshop on Web Content Caching and Distribution (WCW'03), 2002.
8. Liu, Y., Xiao, L., Liu, X., Ni, L.M. Zhang, X.: Location awareness in unstructured P2P pystems. IEEE Trans. Parallel Distrib. Syst., 16(2), 2005.
9. Lv, Q., Cao, P., Cohen, E., Li, K., Shenker, S.: Search and replication in unstructured P2P networks. Proc. of ICS, 2002.
10. Krishnamurthy, B., Wang, J., Xie, Y.: Early measurements of a cluster-based architecture for P2P systems. Proc. of ACM SIGCOMM, 2001.
11. Patro, S., Charlie Hu, Y.: Transparent query caching in P2P overlay networks. Proc. of IPDPS, 2003.
12. Ratnasamy, S., Handley, M., Karp, R.M., Shenker, S.: Topologically-aware overlay construction and server selection. Proc. of IEEE INFOCOM, 2002
13. Saroiu S., Gummadi, K., Dunn, R., Gribble, S., Levy, H.: An analysis of Internet content delivery systems. Proc. of OSDI, 2002.
14. Sripanidkulchai K.: The popularity of Gnutella queries and its implication on scaling. http://www.cs.cmu.edu/ kunwadee/research/p2p/gnutella.html.
15. Wang, C., Xiao, L., Liu, Y., Zheng, P.: DiCAS: an efficient distributed caching mechanism for P2P systems. IEEE Trans. Parallel Distrib. Syst., 2006.

16. Yang, B., Garcia-Molina, H.: Improving search in P2P networks. Proc. of ICDCS, 2002.