Data Indexing in Peer-to-Peer DHT Networks

L. Garcés-Erice, P.A. Felber, E.W. Biersack, G. Urvoy-Keller
Institut EURECOM, 06904 Sophia Antipolis, France
{garces|felber|erbi|urvoy}@eurecom.fr

K.W. Ross
Polytechnic University, Brooklin, NY 11201 , USA
ross@poly.edu

Abstract— **Peer-to-peer distributed hash table (DHT) systems make it simple to discover specific data when their complete identifiers—or keys—are known in advance. In practice, however, users looking up resources stored in peer-to-peer systems often have only partial information for identifying these resources. In this paper, we describe techniques for indexing data stored in peer-to-peer DHT networks, and discovering the resources that match a given user query. Our system creates multiple indexes, organized hierarchically, which permit users to locate data even using scarce information, although at the price of a higher lookup cost. The data itself is stored on only one (or few) of the nodes. Experimental evaluation demonstrates the effectiveness of our indexing techniques on a distributed peer-to-peer bibliographic database with realistic user query workloads.**

I. INTRODUCTION

Peer-to-peer (P2P) systems make it possible to harness the computing power and resources of large populations of networked computers in a cost-effective manner. In this paper, we focus on P2P systems where data items are spread across distributed peer computers (nodes) and the location of each item is determined in a decentralized manner using a distributed hash table (DHT), such as Chord [1], CAN [2], Pastry [3], or Tapestry [4].

A major limitation of P2P DHT systems is that they only support exact-match lookups: one needs to know the exact key (identifier) of a data item to locate the node(s) responsible for storing that item. In practice, however, P2P users often have only partial information for identifying these items and tend to submit broad queries (e.g., all the articles written by "John Smith").

In this paper, we propose to augment P2P DHT systems with mechanisms for locating data using incomplete information. Note that we do not aim at answering complex database-like queries, but rather at providing practical techniques for searching data in a DHT. Our mechanisms rely on indexes, stored and distributed across the nodes of the network, that maintain useful information about user queries. Given a broad query, a user can obtain additional information about the data items that match his original query; the DHT can be recursively queried until the user finds the desired data items. Indexes can be organized hierarchically to reduce space and bandwidth requirements, and to facilitate interactive searches. They integrate an adaptive distributed cache to speed up accesses to popular content.

Our indexing techniques can be layered on top of an arbitrary P2P DHT infrastructure, and thus benefit from any advanced features implemented in the DHT (e.g., replication, load-balancing). We have conducted a comprehensive evaluation that demonstrates their effectiveness in realistic settings. In particular, we have observed that our techniques are scalable, adapt well to user search patterns, and have reasonably-small space requirements. Look-up times depend on the "precision" of the initial query: broad queries incur higher lookup times than specific queries.

The organization of this paper is as follows: In Section II, we discuss related work, and we introduce the system model in Section III. We describe our distributed indexing techniques in Section IV and we evaluate them in Section V. Finally, Section VI concludes the paper.

II. RELATED WORK

Of the various related project, INS/Twine [5] is most similar to our work. INS/Twine is an architecture for intentional (i.e., based on what we are looking for, not where it is located [6]) resource discovery, which allows to easily locate services and devices in large scale environments, using intentional descriptions. INS/Twine works on top of a DHT, such as Chord [1], by setting up a number of resolvers, which collaborate to distribute resource information and to resolve simple queries. Given a semi-structured resource description, INS/Twine extracts prefix subsequences of attributes and values, called "strands". INS/Twine then computes the hash values for each of these strands, which constitutes numeric keys used to map resources to resolvers. The resource and device information are stored redundantly on *all* peer resolvers that correspond to the numeric keys. When looking up some resource, INS/Twine sends the query to the resolver node identified by one of the longest strands; the query is further processed by the resolver, which returns the matching resource descriptions.

Unlike Twine, we do not replicate data at multiple locations; we rather provide a key-to-key service, or more precisely a query-to-query service. We do not introduce dedicated resolvers in our architecture; we only require the underlying distributed data storage system to allow for the registration of multiple entries using the same key. As we allow index keys to be tree-structured or non-prefix sub-keys, data can be looked up using more expressive and selective queries. For improved scalability, index entries are further organized hierarchically.

In [7], the authors discuss techniques for performing complex queries in DHT-based P2P networks, using traditional relational database operators (selection, projection, join, grouping and aggregation, and sorting) and elaborate text retrieval techniques (like splitting a query string and using each piece to create a key matching the query).

In [8], the authors develop a P2P data sharing architecture for computing approximate answers for complex queries by finding data ranges that are similar to the user query. Relevant data is located using "locality sensitive hashing" techniques. In [9], the

```
<article>                        <article>                        <article>
  <author>                         <author>                         <author>
    <first>John</first>              <first>John</first>              <first>Alan</first>
    <last>Smith</last>               <last>Smith</last>               <last>Doe</last>
  </author>                        </author>                        </author>
  <title>TCP</title>              <title>IPv6</title>             <title>Wavelets</title>
  <conf>SIGCOMM</conf>            <conf>INFOCOM</conf>            <conf>INFOCOM</conf>
  <year>1989</year>              <year>1996</year>               <year>1996</year>
  <size>315635</size>            <size>312352</size>             <size>259827</size>
</article>                       </article>                       </article>

          $d_1$                            $d_2$                            $d_3$
```

Fig. 1. Sample file descriptors.

same authors extend the CAN [2] system to support the basic range operation on data shared in the form of database relations.

More recently, the authors of [10] combine the flexibility of searches in unstructured P2P networks (like Gnutella [11]) with the efficiency of DHTs. Peers are organized in groups with common interests. Groups are located with a DHT-like algorithm, and searches are performed within groups using more flexible algorithms. In [12], the authors question the feasibility of an Internet wide search engine based on P2P, but their conclusion do not apply to our techniques, which are targeted at smaller scale systems with well-specified content.

III. System Model and Definitions

The distributed indexes are layered on top of a P2P DHT infrastructure—or substrate—and use various other technologies to describe content and represent user queries. We now describe the specific requirements of our indexing techniques.

A. DHT Storage

A DHT system maps keys to nodes in a peer-to-peer infrastructure. Any node can use the DHT substrate to determine the current live node that is responsible for a given key. In this paper, we assume an underlying DHT-based P2P data storage system, in which each data item is mapped to one or several peer nodes. Example of such systems are Chord/DHash/CFS [13] and Pastry/PAST [14].

Throughout the paper, we will use the example of a bibliographic database system that stores scientific articles. Files are identified by *descriptors*, which are textual, human-readable descriptions of the file's content. Let $h(descriptor)$ be a hash function that maps identifiers to a large set of numeric keys. The peer node responsible for storing a file f is determined by transforming the file's descriptor d into a numeric key $k = h(d)$. This numeric key is used by the DHT substrate to determine the node responsible for f.

B. Data Descriptors and Queries

We assume that descriptors are semi-structured XML data [15], as used by many publicly-accessible databases (e.g., DBLP [16]). Examples of descriptors for bibliographic data are given in Figure 1. These descriptors have fields useful for searching files (author, title), as well as fields useful for an administrator of the database (size).

To search for data stored in the peer-to-peer substrate, we need to specify broad queries that can match multiple file descriptors. For this purpose, we use a subset of the XPath XML addressing language [17], which offers a good compromise between expressiveness and simplicity. XPath treats XML documents as a tree of nodes and offers an expressive way to specify and select parts of this tree. An XPath expression contains one or more *location steps*, separated by slashes (/). In its more basic form, a location step designates an element name followed by zero or more predicates specified between brackets. Predicates are generally specified as constraints on the presence of structural elements, or on the values of XML documents using basic comparison operators. XPath also allows the use of wildcard (*) and ancestor/descendant (//) operators, which respectively match exactly one and an arbitrarily long sequence of element names. An XML document (i.e., a file descriptor) *matches* an XPath expression when the evaluation of the expression on the document yields a non-null object.

```
q₁ = /article[author[first/John][last/Smith]]···
       [title/TCP][conf/SIGCOMM][year/1989][size/315635]
q₂ = /article[author[first/John][last/Smith]][conf/INFOCOM]
q₃ = /article/author[first/John][last/Smith]
q₄ = /article/title/TCP
q₅ = /article/conf/INFOCOM
q₆ = /article/author/last/Smith
```

Fig. 2. Sample file queries.

For a given descriptor d, we can easily construct an XPath expression (or query) q that tests the presence of all the elements and values in d.[1] We call this expression the *most specific query* for d or, by extension, the *most specific descriptor*. Conversely, given q, one can easily construct d, compute $k = h(d)$, and find the file. For instance, query q_1 in Figure 2 is the most specific query for descriptor d_1 in Figure 1.

Given two queries q and q', we say that q' *covers* q (or q is covered by q'), denoted by $q' \sqsupseteq q$, if any descriptor d that matches q also matches q'. Abusing the notation, we often use d instead of q when q is the most specific query for d and we consider them as equivalent ($q \equiv d$); in particular, we say that q' covers d when $q' \sqsupseteq q$ and q is the most specific query for d. It should be noted that the covering relation introduces a partial ordering on the queries.

The partial ordering graph for queries in Figure 2 is shown in Figure 3, where $q_i \rightarrow q_j$ is read $q_i \sqsupseteq q_j$ (more specific queries are represented above less specific queries).

---

[1]In fact, we can create several equivalent XPath expressions for the same query. We assume that equivalent expressions are transformed into a unique normalized format.
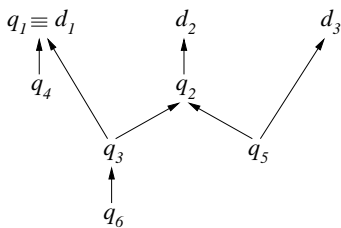
Fig. 3. Partial ordering tree for the queries of Figure 2 (self-covering and transitive relations are omitted).

Fig. 4. Sample indexing scheme for a bibliographic database.

IV. INDEXING

When the most specific query for the descriptor d of a file f is known, finding the location of f is straightforward using the key-to-node (and hence key-to-data) underlying DHT lookup service. The goal of our architecture is to also offer access to f using less specific queries that cover d.

The principle underlying our technique is to generate multiple keys for a given descriptor, and to store these keys in indexes maintained by the DHT infrastructure. Indexes do not contain key-to-data mappings; instead, they provide a key-to-key service, or more precisely a query-to-query service. For a given query q, the index service returns a (possibly empty) list of more specific queries, covered by q. If q is the most specific query of a file, then the DHT storage system returns the file (or indicates the node responsible for that file). By iteratively querying the index service, a user can traverse upward the partial order graph of the queries (see Figure 3) and discover all the indexed files that match his broad query.

In order to manage indexes, the underlying DHT storage system must be slightly extended. Each node should maintain an index, which essentially consists of query-to-query mappings. The "$insert(q, q_i)$" function, with $q \sqsupseteq q_i$, adds a mapping $(q; q_i)$ to the index of the node responsible for key q. The "$lookup(q)$" function, with q not being the most specific query of a file, returns a list of all the queries q_i such that there is a mapping $(q; q_i)$ in the index of the node responsible for key q.

Roughly speaking, we store files and construct indexes as follows: Given a file f and its descriptor d, with a corresponding most specific query q, we first store f at the node responsible for the key $k = h(q)$. We generate a set of queries $Q = \{q_1, q_2, \ldots, q_l\}$ likely to be asked by users (to be discussed shortly), and such that each $q_i \sqsupseteq q$. We then compute the numeric key $k_i = h(q_i)$ for each of the queries, and we store a mapping $(q_i; q)$ in the index of the node responsible for k_i in the DHT. Optionally, we iterate the process shown for q with every q_i and recursively until all the desired index entries have been created.

A. Example

To best illustrate the principle of our indexing techniques, consider a P2P bibliographic database that stores the three files associated to the descriptors of Figure 1. We want to be able to lookup publications using various combinations of the author's name, the title, the conference, and the publication year. A possible hierarchical indexing scheme is shown in Figure 4. Each box corresponds to a distributed index, and indexing keys are
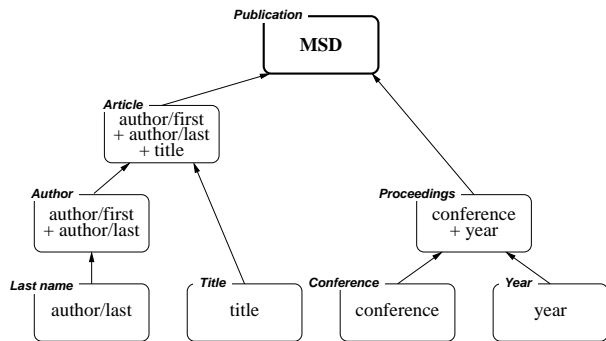
indicated inside the boxes. The index at the origin of an arrow stores mapping between its indexing key and the indexing key of the target. For instance, the *Last name* index stores the full names of all authors that have a given last name; the *Author* index maintains information about all articles published by a given author; the *Article* index stores the descriptors (MSDs) of all publications with a matching title and author name.

After applying this indexing scheme to the three files of the bibliographic database, we obtain the distributed indexes shown in Figure 5. The top-level *Publication* index corresponds to the raw entries stored in the underlying DHT-based storage system: complete key provide direct access to the associated files. The other indexes hold query-to-query mappings that enable the user to iteratively search the database and locate the desired files. Each entry of an index is potentially stored on a different node in the DHT substrate, as illustrated for the *Proceedings* index. One can observe that some index entries associate a query to multiple queries (e.g., in the *Author* index).

Figure 6 details the individual query mappings stored in the indexes of Figure 5. Each arrow corresponds to a query-to-query mapping, e.g., $(q_6; q_3)$. The files corresponding to descriptors d_1, d_2, and d_3 can be located by following any valid path in the partial order tree. For instance, given q_6, a user will first obtain q_3; the user will query the system again using q_3 and obtain two new queries that link to d_1 and d_2; the user can finally retrieve the two files matching its query using d_1 and d_2.

B. Lookups

We can now describe the lookup process more formally. When looking up a file f using a query q, a user first contacts the node n responsible for $h(q)$. That node may return f if q is the most specific query for f, or a list of queries $\{q_1, q_2, \ldots, q_n\}$ such that the mappings $(q; q_i)$, with $q \sqsupseteq q_i$, are stored at n. The user can then choose one or several of the q_i and repeat this process recursively until the desired files have been found. The user effectively follows an "index path" that leads from q to f.

Lookups may require several iterations when the most specific query for a given file is not known. Higher index hierarchy usually necessitate more iterations to locate a file, but are also generally more space-efficient, as each index factorizes in a compact manner the queries of its child indexes. In particular, the size of the lists (result sets) returned by the index service
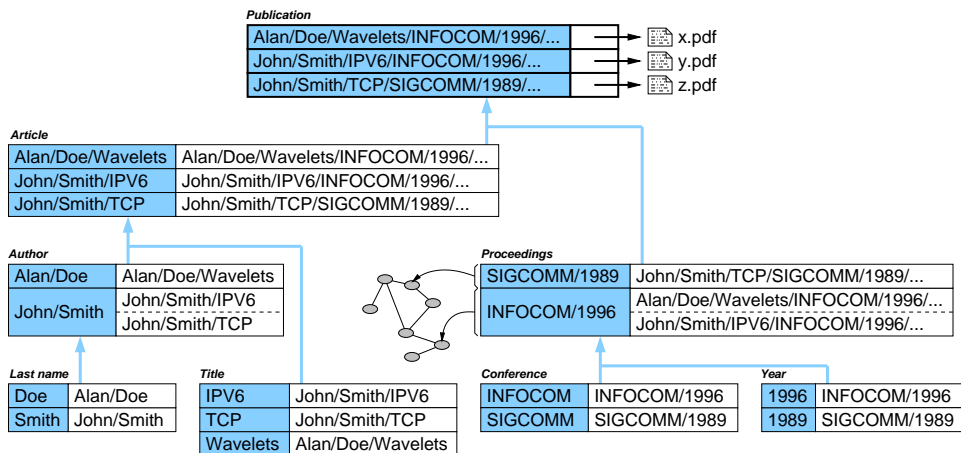
Fig. 5. Sample distributed indexes for the three documents of Figure 1 and the indexing scheme of Figure 4 (query syntax has been simplified).
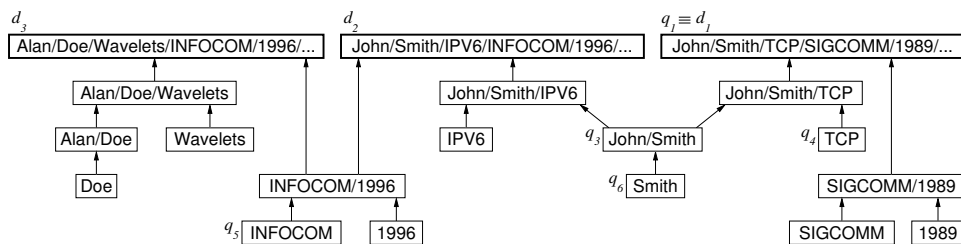

Fig. 6. Query mappings for the indexes of Figure 5 (identifiers correspond to Figures 1 and 2; query syntax has been simplified).

may be prohibitively long when using a flat indexing scheme (consider, for example, the list of all articles written by the persons whose last name is "Smith"). There is therefore a trade-off between space requirements, size of result sets, and lookup time, as we shall see in the experimental evaluation.

The lookup process can be interactive, i.e., the user directs the search and restricts its query at each step, or automated, i.e., the system recursively explores the indexes and return all the file descriptors that match the original query.

When a user wants to look up a file f using a query q, it may happen that q is not present in any index, although f does exist in the peer-to-peer system and q is a valid query for f. In that case, it is still possible to locate f by (automatically) looking for a query q_i such that $q_i \sqsupseteq q$ and q_i is on some index path that leads to f.

For instance, given the distributed indexes of Figures 5 and 6, it appears that query q_2 in Figure 2 is not present in any index ($q_2 =$/article[author[first/John][last/Smith]][conf/INFOCOM]). We can however find q_3, such that $q_3 \sqsupseteq q_2$ and there exists an index path from q_3 to d_1. Therefore, the file associated to d_1 can be located using this generalization/specialization approach, although at the price of a higher lookup cost. We believe that it is natural for lookups performed with less information to require more effort.

C. Building and Maintaining Indexes

When a file is inserted in the system for the first time, it has to be indexed. The choice of the queries under which a file is indexed is arbitrary, as long as the covering relation holds. As files are discovered using the index entries, a file is more likely to be located rapidly if it is indexed "enough" times, under "likely" names. The quantity and likelihood of index queries are hard to quantify and are often application-dependent. For instance, in a bibliographic database, indexing a file by its size is useless for users, as they are unlikely to know the size beforehand. However, indexing the files under the author, title, and/or conference are appropriate choices.

Note that the length of the index paths that lead to a given file is arbitrary, although it directly affects the lookup time. Less popular content may be indexed using a deeper index hierarchy, to reduce space and bandwidth requirements, and to facilitate interactive searches. In contrast, a very popular file can be linked to deep in the hierarchy to short-circuit some indexes and speed up lookups. For instance, given the distributed indexes of Figures 5 and 6, one can add the $(q_6; d_1)$ index entry to speed up searches for the popular file described by d_1 (e.g., the author's most popular publication).

Note also that more generic queries can be obtained from more specific queries by removing only portions of element names (i.e., using substring matching). For instance, one can create an index with all the files of an author that start with the letter "A", the letter "B", etc. One can also envision to use techniques similar to those discussed in [7] for substring matching. In general, determining good decompositions for indexing each given descriptor type (e.g., articles, music files, movies, books, etc.) requires human input.

In a system model where files are injected in the system, but are never deleted (write-once semantics), index entries never need to be updated. In a read/write system, when a file is

deleted we have to find all the indexes that refer to the descriptor of that file and delete the associated mappings. Locating the index entries can be achieved straightforwardly by using the same process used to generate them in the first place when the file was injected in the system. When deleting the last mapping for a given key, we can recursively delete the references to that key to clean up the indexes.

Index entries can also be created dynamically to adapt to the users query patterns. For instance, a user who tries to locate a file f using a non-indexed query, and eventually finds it using the query generalization/specialization approach discussed above, can add an index entry to facilitate subsequent lookups from other users.

More interestingly, one can easily build an adaptive cache in the DHT to speed up accesses to popular files. Assume that each node allocates a limited number of index entries for caching purposes. After a successful lookup, a peer can create "shortcuts" entries (i.e., direct mappings between generic queries and the descriptor of the target file) in the caches of the indexes traversed during the lookup process. Another user looking for the same file via the same path will be able to "jump" directly to the file by following the shortcuts stored in the caches. With a least-recently used (LRU) cache replacement policy, we can guarantee that the most popular files are well represented in the caches and are accessible in few hops. The caching mechanism therefore adapts automatically to the query patterns and file popularities.

D. Discussion

We outline below some interesting properties of our indexing techniques, which we will further study in the experimental evaluation:

- *Space efficient:* First, as indexes contain key-to-key mappings, the data items do not have to be stored on multiple nodes. Second, although data items may be reached through multiple index paths, the space requirements remain reasonably small because coarse-level indexes are shared by many data items (e.g., given the mappings of Figure 6, $(q_6; q_3)$ is on index paths to both d_1 and d_2). Finally, the hierarchical index organization significantly reduces the size of results sets, and consequently the bandwidth requirements.
- *Scalability:* As data items may be accessed through distinct paths and are referred to in distinct indexes, the load is expected to be spread across multiple indexes, and thus multiple nodes (in contrast to a centralized index). In addition, since indexes are stored as regular data item, they can benefit from the mechanisms implemented by the DHT substrate for increasing availability and scalability, such as data replication or caching.
- *Loose coupling between data and indexes:* When the data items change, only the nodes responsible for the complete key of the data need to be updated. Indexes do not need to be updated. This is a consequence of the key-to-key mapping technique.
- *Versatility:* It is possible not to index some data, and enforce access using the complete key. Conversely, some

popular data may be aggressively indexed to speed up accesses.
- *Adaptability:* The system can create index entries on-demand to match user querying habits or for caching purposes.
- *Decentralized architecture:* Indexes are uniformly distributed across all nodes. The lookup load is therefore balanced among all the nodes.
- *Resilient to arbitrary linking:* When inserting a file in the system, it can only be indexed at locations that correspond to keys covering the file's key. Arbitrary links (or aliases) to a file cannot be inserted in the system. This makes it harder for a user to inject a file with malicious or offensive content and masquerade it as another existing file.

V. Evaluation of Data Indexing on P2P Networks

The indexing mechanism lies in the protocol stack on top of a P2P lookup and storage layer. Thus, one aspect of indexing performance deals with the optimization of resources offered by those lower layers. As our indexing techniques do not depend on a specific lookup and storage layer, we do not explicitly study the performance of the P2P substrate.

The index protocol layer also interacts with the end user, who expects to find data of interest using partial information. From the user point of view, it is clearly desirable to find the desired data in a minimal number of interactions with the system, and the information returned by the system should be as concise and relevant as possible. From the system point of view, the search process should be simple, the amount of network traffic should be minimized, and the storage space dedicated to the indexing metadata should remain within reasonable limits. These various criteria are studied in the rest of this section.

A. Distributed Bibliographic Database

To study the behavior of a P2P indexed network, we model a bibliographic database distributed among interconnected hosts. The bibliographic database contains articles published in journals and conference proceedings. The underlying P2P lookup and storage system, and the physical characteristics of the network, are not important: we simply assume that the underlying DHT is able to find a node n responsible for a given key k, where n stores (or knows the location of) the data associated with key k.

In order to build the bibliographic database, we used the publicly-available DBLP [16] archive, which consists of an XML-formatted list of publications. The DBLP archive, as of January 21st, 2003, contains more than $346,000$ entries: articles, theses, proceedings, etc. For our experiments we used the $115,879$ article entries in the archive to build the descriptors (MSDs) of the corresponding articles. The archive being in XML format, entries are pretty similar to those in Figure 1 (actual field names differ).

B. Building Indexes

From the MSD, which actually links to the real data, we build chains of queries, i.e., sequences of queries where each query

covers (⊒) the next one. The last member of each chain is obviously the MSD, which is covered by all the previous ones. Each chain corresponds to a path from a leaf to a root in Figure 3.

To create indexes that correspond to queries that users are likely to ask, we have observed the query logs of two other bibliographic database sites: BibFinder [18] and NetBib [19] (the DBLP service does not store and share query logs). NetBib offers an interface where a user can search for papers using fields like: words in title or abstract, exact title, author, publication date (year intervals), and citation key. BibFinder displays a similar interface: a user can issue queries with the author name, title, conference or journal, and year of publication (exact, or published before/after a given year).
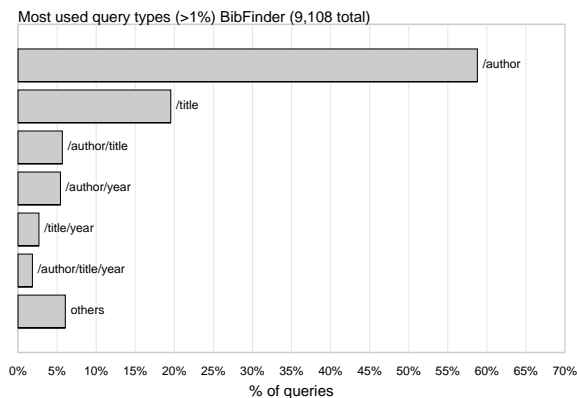

Fig. 7. Distribution of the types of queries extracted from BibFinder's log.

The log from BibFinder's site contains $9,108$ queries. As shown in Figure 7, most of them use only the "author" field (57%), followed by those using only the "title" field (20%), and those with a given publication date.

The NetBib trace represents $5,924$ different queries. Similarly to the BibFinder traces, there are three main kinds of queries (summing up to more than 95% of the total): queries for a given author, topic (title), and date of publication.

Based on this information, we have simulated and compared the following three indexing schemes, shown in Figure 8.

- *Simple:* A query for an author or a title returns a set of author and title pairs. Each of them points to a MSD. A query for a conference or a year returns a set of conference-year pairs, pointing to a MSD.
- *Flat:* All possible queries in the *simple* scheme point directly to the MSD, so that the index query length is always 2.
- *Complex:* Some queries in the *simple* scheme are split into more specific queries, in order to avoid long result lists. For instance, a query specifying an author and a conference returns a list of queries that further indicate all the publication years for the given author and conference. Although we do not expect index hierarchies to be very deep in practice, this scheme allows us to observe the effect of hierarchy depth.

The *simple* indexing scheme is the most space-efficient of the three, requiring 152 MB of extra storage in the system for the full DBLP article collection. The *complex* scheme requires 25% more space, and the *flat* scheme a 37% increase, being the most space-consuming.

By comparison, we have evaluated the cost of storing the actual article files in the P2P infrastructure. Based on an average file size of 250 KB (estimated from the size of the articles in our own private digital library, counting more than $7,000$ files in Postscript and Adobe PDF format), 29.1 GB are required to store all the articles in the DBLP archive. In the worst case, the indexes require an additional storage capacity of 0.5%. For a P2P network already storing the full article collection, index storage requirements are clearly negligible.

C. User Model

The mean number of interactions needed to reach an article gives a measure of the efficiency of a given indexing scheme. This can be computed as the average length of all possible query chains a user can follow to find an article. This is, however, only valid in an scenario where all kinds of queries are used with equal probability, and all articles are equally requested; such assumptions are not realistic.

To model realistic user behavior, we built a query generator that reflects which information is used to build queries (query structure), and which data is requested (data popularity).

a) Realistic query structure model: To generate realistic queries, we have used the information gathered from the query logs of the BibFinder and NetBib sites. For both of these archives, results from queries are always a list of matching articles. Any refinement made by a user (possibly overwhelmed by a huge list of results) is independent from the previous query. We may then consider all queries as independent from each other. Both logs agree on the fact that queries are mainly made using author names, with conference and publication date as second and third criteria. We have therefore mapped our experimental query model directly to the structure and frequency of the queries in the BibFinder log, as shown in Figure 7.

b) Realistic popularity model: After modeling *how* articles are queried, we need to model *which* of them are most often requested. We have first observed author popularities, by counting the number of queries to each author in the BibFinder and NetBib log traces: the popularity of an author is defined as the probability of having a query with the "author" field being for that author. Similarly, we have computed the popularity of requested article titles in the BibFinder log. Finally, we have observed the probability that each of the top-$10,000$ articles in the CiteSeer database [20] gets cited; although this is a measure of how frequently a given article is cited, it can be considered as a clear indication of the article's popularity.

The observed probability distribution are shown in Figure 9 (logarithmic scales, with articles ordered by decreasing rank of popularity). It appears clearly that all probabilities follow roughly a power-law [21]. That is, the popularity of an article with rank i is $p_i = \frac{k}{x^i}$, and $\sum p_i = 1$. A few articles appear in many queries, while most authors are seldom requested.

To model article popularities for our simulations, we have computed (using the minimum square method) from the plot of BibFinder's author probabilities the line that best fits the
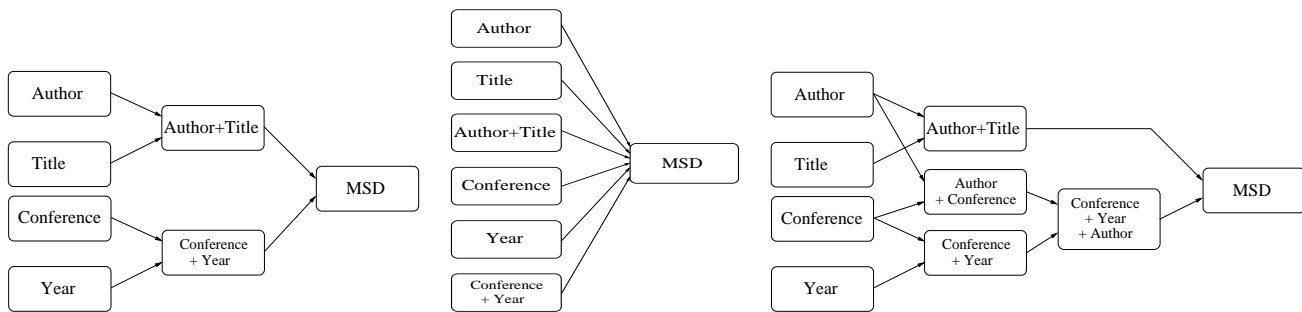
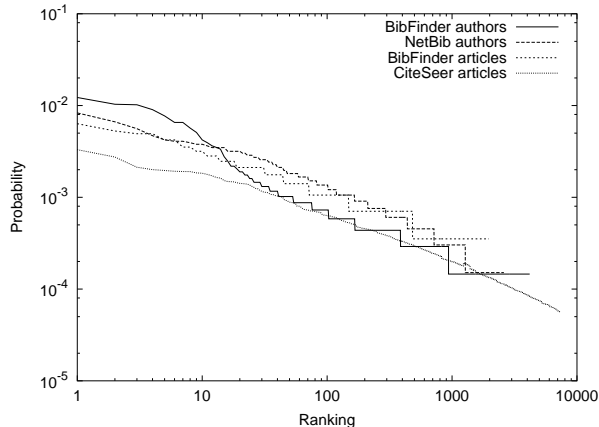Fig. 8. Indexing schemes: Simple (left), Flat (center) and Complex (right).

Fig. 9. Popularity distribution for authors and titles present in NetBib, BibFinder, and CiteSeer.
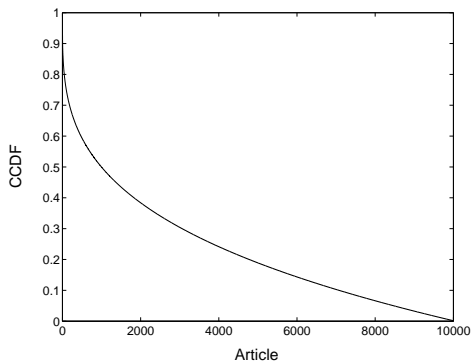

Fig. 10. Complementary cumulative distribution function of the articles ranking.

distribution; switching to a linear scale, we obtain the power-law distribution describing the popularity of each article and the associated cumulative distribution function. In order to simplify the simulations, we have considered a limited collection of $10,000$ articles. After adapting the parameters of the power-law distribution to match the finite population of articles, we obtain a complementary cumulative distribution function of $\bar{F}(i) = 1 - F(i) = 1 - 0.063 \cdot i^{0.3}$, where i is the ranking of the article. Numerical values are given for illustration purposes only: the behavior is defined by the *family* of the probability distribution function, i.e., power-law functions. The distribution function is plotted in Figure 10.

From the figure we can appreciate that, because of the skewed distribution, using only $10,000$ articles does not change significantly the behavior of the model. The remaining articles from the original DBLP archive would be requested so seldom that we can effectively neglect their existence. When constructing the query workload for the simulation, we first choose an article according to the popularity distribution. Then, we select the structure of the query and assign the corresponding fields, according to the following probabilities: author only (with probability 0.6); title only (0.2); year only (0.1); both author and title (0.05); both author and year (0.05). The resulting query is subsequently used as input to locate the article selected initially.

D. Caching

As previously discussed, some articles are much more popular than others, and hence will be requested more frequently. By creating cache entries (shortcuts) for these articles, we can probabilistically improve lookup times and reduce the load on the system. We have observed the effectiveness of the adaptive caching mechanism described in Section IV, in which peers create cache entries along the lookup paths of successful queries. In the simulation, we study three different caching policies:

- *Multi-cache:* Shortcuts are created on each node along the lookup path. The size of the cache is unbounded.
- *Single-cache:* Shortcuts are created only on the first node that was contacted. The size of the cache is unbounded.
- *LRU:* Similar to the single-cache policy, but only a limited number of shortcuts can be stored on each node. When the cache is full, a least-recently used (LRU) replacement algorithm is used.

E. Simulation

Simulating P2P networks of different sizes is of no use for our experiments. The number of nodes can affect the DHT lookup latency, and the number of keys stored per node, but does not impact the effectiveness of our indexing techniques. Any optimization of the underlying P2P network to reduce lookup latency will improve the response time when searching through the indexes, but these are completely independent issues (layered protocols). Our experiments simulate a P2P network of 500 nodes, on top of which a distributed bibliographic database storing $10,000$ articles is implemented. For each of the indexing schemes and caching policies previously

described, we measure different important metrics during simulation. Each simulation consists of sequentially feeding the indexing network with $50,000$ queries from our query generator. The studied parameters are: number of user-system interactions required to find data, traffic generated, efficiency of shortcuts, and storage dedicated to caching. In all the figures of this section, S, F, and C stand for *simple*, *flat*, and *complex* indexing, respectively. The LRU replacement caching policy is tested for an allowed maximum of 10, 20, and 30 cached keys per node.

c) User-system interactions: A user sends a query and obtains as result a list of more specific queries (covered by the original query). The user then selects one query from the results that matches the target article. This process iterates until the article is found. Isolated from other concerns, a user would like to experience a minimal number of iterations to locate the desired data. One should note, however, that the number of iterations is expected to increase when the user initiates the search with a generic query; there is clearly a trade-off between the amount of information initially known about the searched data, and the number of steps necessary to locate that data.
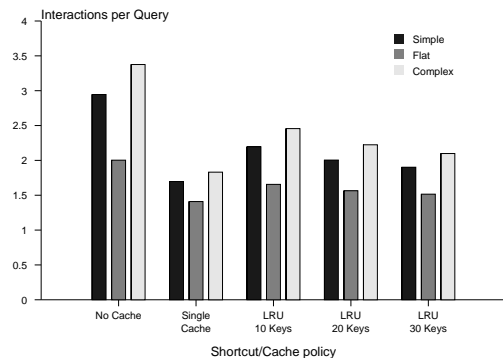

Fig. 11.   Average number of interactions required to find data.

Simulation results are shown in Figure 11. The *flat* indexing scheme, which creates the shortest query chains (see Figure 8), also requires the fewest interactions to locate data. Caching further reduce the number of lookup steps, which becomes smaller as the cache capacity increases. The multi-cache policy is not shown here because it presents the same characteristics as the single-cache policy. Due to the rather simple representation of data, index chains are in general short. More complex data representations would need longer index chains, where the effect of shortcuts is more important.

d) Generated traffic: Like in any other network, it is desirable to generate as little traffic as possible to avoid network congestion and save system resources. Another more subtle interpretation of the traffic generated is the number of responses that a user receives for a query, i.e., the size of the result sets (traffic is mainly driven by responses, which usually outnumber a single query). The more traffic, the more responses a user gets for a query. Large results sets increase the burden of the user because responses are less relevant and more post-processing is required. Note that, since our search process is completely deterministic, more results do not imply that more information is available (as could be the case of an Internet search engine

such as Google [22]), but a *less precise answer*. The query and response traffic measured during the simulations is shown in Figure 12.
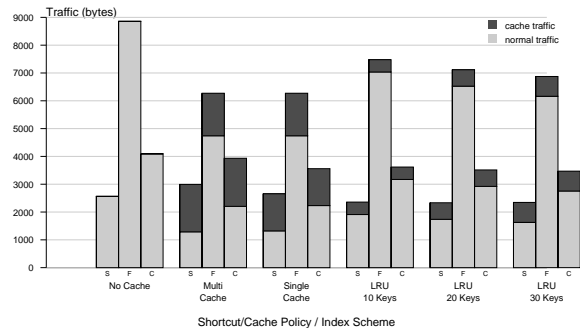

Fig. 12.   Average network traffic (bytes) generated per query.

Cache traffic, resulting from the creation of cache entries after successful lookups, is shown in dark gray. It appears clearly that the *flat* indexing scheme generates much more traffic than any other. In fact, it does not allow for any indirection, and each query receives directly the descriptors of *all* articles that match the query, instead of a relevant set of more specific queries that allows refinement of the search process. We observe that the utilization of the cache saves network bandwidth. Unsurprisingly, larger cache sizes yield more cache traffic and less overall traffic transmitted over the network. The multi-cache policy leads to more cache traffic than single-cache, because the latter creates only one cache entry upon successful lookup.

e) Cache efficiency: We have observed the effectiveness of the adaptive distributed cache by measuring its hit ratio, that is, the fraction of requests that do not need to go through a full search process because the relevant data is already available in the cache.
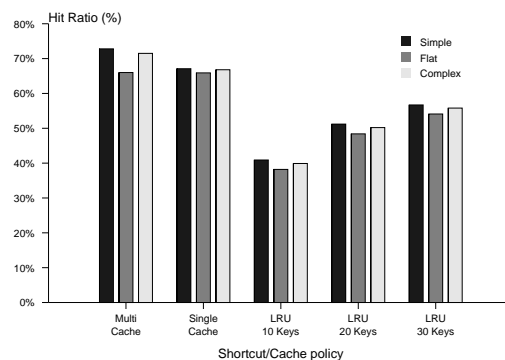

Fig. 13.   Cache efficiency: distributed hit ratio.

Figure 13 shows the results for the different policies tested. It is interesting to note that the multi-cache policy is only marginally more efficient than the single-cache policy. Although the multi-cache policy stores a cached key on every node in the index chain followed to find the data, most cache hits occur in the first node of the chain (86% for the *simple* scheme, 99.9% for *flat*, and 84% for *complex*). Indeed, in our

user model queries are usually very simple and broad, thus directing the user to the beginning of an index chain. It is also worth noting that, when limiting the number of cache entries per node to just 10, cache efficiency is still more than half that of policies with unbounded cache size.

f) Cache storage: The storage dedicated to cached keys should remain relatively small, while allowing for an efficient distributed cache. The number of regular keys stored on each node depends on the indexing scheme. After feeding the $50,000$ queries, we observed an average of 155 keys per node for *simple*, 195 for *flat*, and 180 for *complex*. Cache storage sizes are shown in Figure 14.
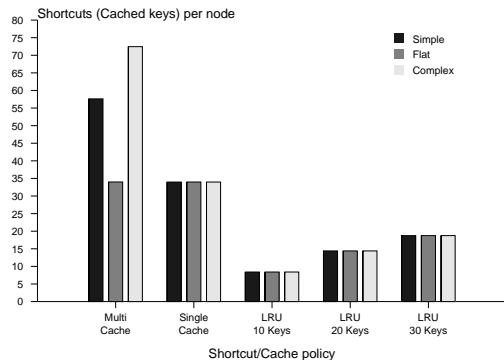

Fig. 14.   Average number of cached keys per node.

As expected, the single-cache policy is approximately twice as space-efficient as the multi-cache policy, which creates more cache entries. The *flat* indexing scheme is unaffected because its index chains are so short that they only allow for a key to be cached in the first node.

We also observed the maximum number of cache entries stored across all nodes. For the multi-cache policy, we find up to 345 cached keys in a node with the *simple* indexing. For the *flat* and *complex* indexing, the maxima are 253 and 413 keys, respectively. The single-cache policy uses a maximum of 253 cached keys, regardless of the indexing scheme used. For the limited cache size policies, the maxima are obviously the cache capacities (10, 20, and 30). 72% of the caches were full after feeding the $50,000$ queries with the LRU10 policy, 51.2% with LRU20, and 37.6% with LRU30, regardless of the indexing scheme. Overall, 4.4% of the caches were completely empty, with not a single entry.

As the cache management directly depends on the query workload, cache utilization is not uniform. The distributed cache improves lookup performance, but does not solve load-balancing issues in the overlay network.

g) Hot-spots: As just discussed, the user query patterns leads to an unbalanced utilization of the distributed cache. We have studied the distribution of the processing of queries among nodes. In Figure 15, we show the percentage of the $50,000$ queries issued during the simulation that accessed each node (for clarity, we only show the results for the *simple* indexing scheme). Note that they sum to more than 100% because each original query may generate other queries during the iterative lookup process. We observe that the busiest node is affected by
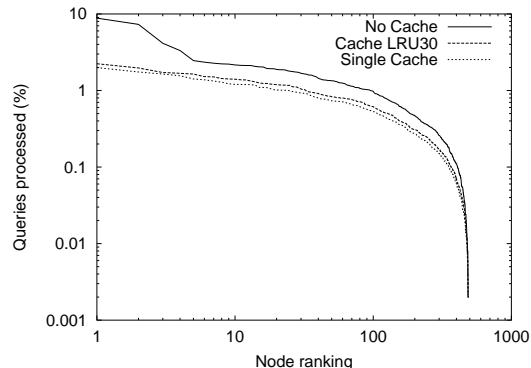

Fig. 15.   Percentage of queries processed by each node in the network.

almost 1 out of every 10 queries. Caching slightly improves the situation for the most severely stressed nodes. We expect the load to be better balanced with larger content and query workloads. Note also that any optimization of the underlying P2P DHT substrate for hot-spot avoidance (e.g., using replication) will apply to index accesses as well.

h) Locating non-indexed data: As previously mentioned, it may happen that a user issues a query using a combination of fields (author, year, etc.) that has not been indexed. In that situation, the system can first generalize the original query to find a matching index entry, and then specialize it by following indexes until the target data is located. In our experiments, this event happened approximately $2,500$ times for all indexing schemes when no cache is used. When an error is encountered, one extra interaction is generally necessary to find a suitable index (two interactions in a few rare cases).

	Simple	Flat	Complex
No cache	2,502	2,507	2,506
LRU30	810	874	838
Single-cache	563	600	581

TABLE I

NUMBER OF QUERIES TO NON-INDEXED DATA.

Table I shows the number of accesses to non-indexed data, i.e., recoverable errors, as a function of the cache policy deployed in the system. We can observe that the cache reduces the number of errors, because an index entry is created automatically after the first lookup; subsequent queries from other users can locate the data using the cache entry, and hence do not experience an error. Our indexing techniques, together with the adaptive distributed cache, are thus very flexible: although the indexing scheme is generally chosen at deployment time, the system can still adapt to the user querying habits by creating shortcuts dynamically.

VI. FINAL NOTES

A major limitation of DHT peer-to-peer system is that they only support exact-match lookups: one needs to know the exact key of a data item to locate the node responsible for storing that item. Since peer-to-peer users tend to submit broad queries

to look up data items, DHT peer-to-peer systems need to be augmented with mechanisms for locating data using incomplete information.

In this paper, we have proposed techniques for indexing the data stored in the peer-to-peer network. Indexes are distributed across the nodes of the network and contain key-to-key (or query-to-query) mappings. Given a broad query, a user can look up the more specific queries that match its original query; the DHT can be recursively queried until the user finds the desired data items. As they can be layered on top of an arbitrary P2P DHT substrate, our indexing techniques directly benefit from any mechanisms implemented in the DHT to deal with failures or hot-spot avoidance. We have performed a comprehensive evaluation to demonstrate the effectiveness of data indexing on a distributed P2P bibliographic database, using different strategies and realistic user query workloads. Finally, we have proposed an adaptive caching mechanism that has proved to improve performance of accesses to popular content.

Although our data indexing techniques permit to look up data based on incomplete information, they still depend on the exact matching facilities of the underlying DHT. "Fuzzy" matching techniques offer interesting research perspectives for dealing with misspelled data descriptors or queries. Misspellings can also often be taken care of by validating descriptors and queries against databases that store known file descriptors, such as CDDB [23] for music files.

REFERENCES

[1] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceeding of SIGCOMM*, 2001.

[2] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceeding of SIGCOMM*, 2001.

[3] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proceedings of Middleware*, Nov 2001.

[4] B.Y. Zhao, J. Kubiatowicz, and A.D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Tech. Rep. UCB/CSD-01-1141, University of California, Berkeley, Apr 2001.

[5] M. Balazinska, H. Balakrishnan, and D. Karger, "INS/Twine: A scalable peer-to-peer architecture for intentional resource discovery," in *Proceedings of the International Conference on Pervasive Computing*, August 2002.

[6] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The design and implementation of an intentional naming system," in *Proceedings of SOSP*, 1999.

[7] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica, "Complex queries in dht-based peer-to-peer networks," in *Proceedings of IPTPS*, Cambridge, USA, March 2002.

[8] A. Gupta, D. Agrawal, and A. Abbadi, "Approximate range selection queries in peer-to-peer systems," Tech. Rep. UCSB/CSD-2002-23, University of California at Santa Barbara, 2002.

[9] O.D. Sahin, A. Gupta, D. Agrawal, and A. Abbadi, "Query processing over peer-to-peer data sharing systems," Tech. Rep. UCSB/CSD-2002-28, University of California at Santa Barbara, 2002.

[10] E. Cohen, A. Fiat, and H. Kaplan, "Associative search in peer to peer networks: Harnessing latent semantics," in *Proceedings of Infocom*, San Francisco, CA, April 2003.

[11] "Gnutella," http://gnutella.wego.com.

[12] J. Li, B.T. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. Morris, "On the feasibility of peer-to-peer web indexing and search," in *Proceedings of IPTPS*, Berkeley, CA, February 2003.

[13] F. Dabek, M. F. Kaashoek, David Karger, Robert Morris, and Ion Stoica, "Wide-area cooperative storage with cfs," in *Proceedings of SOSP*, Oct. 2001.

[14] A. Rowstron and P. Druschel, "Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility," in *Proceedings of SOSP*, Oct. 2001.

[15] W3C, "Extensible Markup Language (XML) 1.0," http://www.w3.org/TR/REC-xml, Oct. 2000.

[16] "DBLP," http://dblp.uni-trier.de.

[17] W3C, "XML Path Language (XPath) 1.0," http://www.w3.org/TR/xpath, Nov. 1999.

[18] "Bibfinder," http://kilimanjaro.eas.asu.edu.

[19] "Netbib," http://edas.info/S.cgi?search=1.

[20] "Citeseer," http://citeseer.nj.nec.com/cs.

[21] G.K. Zipf, *Human Behavior and the principle of least effort*, Addison-Wesley Press, Cambridge (MA), 1949.

[22] "Google," http://www.google.com.

[23] "CDDB," http://www.cddb.org.