Structured Peer-to-Peer Networks: Faster, Closer, Smarter

P. Felber [1], K.W. Ross [2], E.W. Biersack [3], L. Garcés-Erice [4], G. Urvoy-Keller [3]
[1] University of Neuchâtel, Switzerland [2] Polytechnic University, NY, USA
[3] Institut EURECOM, Sophia Antipolis, France [4] IBM Research, Zürich, Switzerland

Abstract

Peer-to-peer (P2P) distributed hash tables (DHTs) are structured networks with decentralized lookup capabilities. Each node is responsible for a given set of keys (identifiers) and lookup of a key is achieved by routing a request through the network toward the current peer responsible for the desired key. DHT designs are usually compared in terms of degree (number of neighbors) and diameter (length of lookup paths). In this paper, we focus three other desirable properties of DHT-based systems: We first present a topology-aware DHT that routes lookup requests to their destination along a path that mimics the router-level shortest-path, thereby providing a small "stretch." We then show how we can take advantage of the topological properties of the DHT to cache information in the proximity of the requesters and reduce the lookup distance. Finally, we briefly discuss techniques that allow users to look up resources stored in a DHT, even if they only have partial information for identifying these resources.

1 Introduction

Several important proposals have recently been put forth for providing distributed peer-to-peer (P2P) lookup services based on distributed hash tables (DHTs). A DHT maps keys (data identifiers) to the nodes of an overlay network and provides facilities for locating the current peer node responsible for a given key. DHT designs differ mostly by the way they maintain the network and perform lookups: there is a fundamental trade-off between the degree of the network (the number of neighbors per node, i.e., the size of the routing tables) and its diameter (the average number of hops required per lookup) [12]. There are, however, other important aspects that should be taken into consideration when designing a DHT.

In this paper, we present our research on three facets of DHT-based systems. We first explore in Section 2 the design of a P2P lookup service for which topological considerations take precedence so as to provide *faster* lookup. We propose a P2P network design with a new lookup service, TOPLUS (Topology-Centric LookUp Service). In TOPLUS, peers that are topologically close are organized into groups. In turn, groups that are topologically close are organized into supergroups, close supergroups into hypergroups, etc. The groups within each level of the hierarchy can be heterogeneous in size and in fan-out. Groups can be derived directly from the network prefixes contained in BGP tables or from other sources. TOPLUS has many strengths, including a small "stretch" (the ratio of the latency between two hosts through the overlay to the latency through layer-3 IP routing), efficient forwarding mechanisms, and a fully symmetric design.

We also elaborate on the natural caching capabilities of TOPLUS and show how it can be leveraged to replicate content *closer* to the requesters. Distributed caches can be deployed in a straightforward manner at any level of the TOPLUS hierarchy in order to reduce transfer delays and limit the external traffic of a given network (campus, ISP, etc.).

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Finally, we present in Section 3 distributed indexing techniques that allow users to locate data using incomplete information, i.e., partial keys, and hence provide a *smarter* lookup. These techniques have been designed for indexing data stored in arbitrary 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.

Due to space limitations, we shall refer the reader to other publications [5, 4, 3] for a more detailed description of our research on DHTs, including additional technical information, experimental evaluation, and exhaustive comparison with related work.

2 Topology Awareness

Early DHT designs did not take into account network topology. In particular, small steps in the logical address space usually resulted in large geographical jumps, which was observed to be a major bottleneck. As a result, some researchers have modified their DHTs to take topology into special consideration (e.g., [10, 1, 6]), but as an add-on rather as a foundation of the P2P system. For instance, in Pastry [1], a message takes small topological steps initially and big steps at the end of the route so as not to wander too far off the source, but the message is unlikely to proceed physically *toward* the destination.

In contrast, our TOPLUS lookup service was designed from the ground up with topological considerations in mind. It strives to route messages along a path that mimics the router-level shortest-path distance, i.e., by taking steps that systematically reduce the distance remaining to the destination. Informally, routing proceeds as follows: Given a message with key k under the responsibility of a peer p_d in the system, (1) source peer p_s sends the message (through IP-level routers) to a first-hop peer p_1 that is "topologically close" to p_d; (2) after arriving at p_1, the message remains topologically close to p_d as it is routed closer and closer to p_d through the subsequent intermediate peers. Clearly, if the lookup service satisfies these two properties, the stretch should be very close to 1.

We now formally describe TOPLUS in the context of IPv4. Let I be the set of all 32-bit IP addresses.[1] Let $\mathcal{G}$ be a collection of sets such that $G \subseteq I$ for each $G \in \mathcal{G}$. Thus, each set $G \in \mathcal{G}$ is a set of IP addresses. We refer to each such set G as a *group*. Any group $G \in \mathcal{G}$ that does not contain another group in $\mathcal{G}$ is said to be an *inner group*. We say that the collection $\mathcal{G}$ is a *proper nesting* if it satisfies all the following properties: (1) $I \in \mathcal{G}$; (2) for any pair of groups in $\mathcal{G}$, the two groups are either disjoint, or one group is a proper subset of the other; (3) For each $G \in \mathcal{G}$, if G is not an inner group, then G is the union of a finite number of sets in $\mathcal{G}$; and (4) each $G \in \mathcal{G}$ consists of a set of contiguous IP addresses that can be represented by an IP prefix of the form $w.x.y.z/n$ (for example, $123.13.78.0/23$).

As shown in [5], the collection of sets $\mathcal{G}$ can be created by collecting the IP prefix networks from BGP (border gateway protocol) tables and/or other sources [7, 11]. An autonomous system (AS) uses BGP to know which ASes it is connected to, and more important, which other ASes can be reached through each of them. A network is represented by an IP prefix and prefix aggregation allows to hide the complexity of routing inside the AS while offering other ASes enough information to route IP packets.

In TOPLUS, many of the sets $\mathcal{G}$ would correspond to ASes, other sets would be subnets in ASes, and yet other sets would be aggregations of ASes. This approach of defining $\mathcal{G}$ from BGP tables requires that a proper nesting is created. In order to reduce the size of the nodal routing tables, groups may be aggregated and artificial tiers may be introduced. Note that the groups differ in size, and in the number of subgroups (the fanout).

If $\mathcal{G}$ is a proper nesting, then the relation $G \subset G'$ defines a partial ordering over the sets in $\mathcal{G}$, generating a partial-order tree with multiple tiers. The set I is at tier-0, the highest tier. A group G belongs to tier-1 if

[1]For simplicity, we assume that all IP addresses are permitted. Of course, some blocks of IP addressed are private and other blocks have not been defined. TOPLUS can be refined accordingly.

there does not exist a G' (other than I) such that $G \subset G'$. We define the remaining tiers recursively in the same manner (see Figure 1).

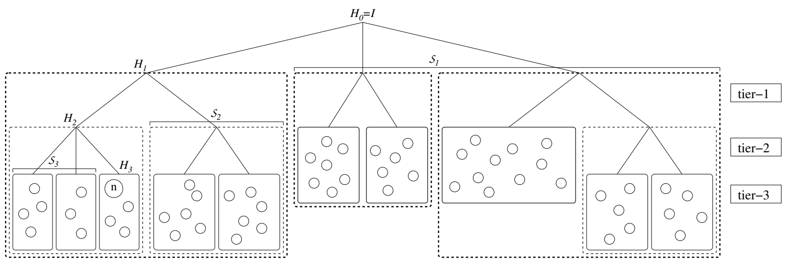

Figure 1: A sample TOPLUS hierarchy (plain boxes are inner groups)

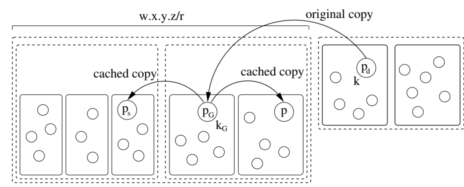

Figure 2: Data caching in TOPLUS.

Peer state. Let L denote the number of tiers in the TOPLUS tree, let U be the set of all current up peers and consider a peer $p \in U$. Peer p is contained in a collection of telescoping sets in $\mathcal{G}$; denote these sets by $H_i(p), H_{i-1}(p), \cdots, H_0(p) = I$, where $H_i(p) \subset H_{i-1}(p) \subset \cdots \subset H_0(p)$ and $i \leq L$ is the tier depth of p's inner group. Except for $H_0(p)$, each of these telescoping sets has one or more siblings in the partial-order tree (see Figure 1). Let $\mathcal{S}_i(p)$ be the set of siblings groups of $H_i(p)$ at tier-i. Finally, let $\mathcal{S}(p)$ be the union of the sibling sets $\mathcal{S}_1(p), \cdots, \mathcal{S}_i(p)$.

For each group $G \in \mathcal{S}(p)$, peer p should know the IP address of at least one peer in G and of all the other peers in p's inner group. We refer to the collection of these two sets of IP addresses as peer p's *routing table*, which constitutes peer p's state. The total number of IP addresses in the peer's routing table in tier i is $|H_i(p)| + |\mathcal{S}(p)|$.

The XOR metric for DHTs. Each key k' is required to be an element of I', where I' is the set of all b-bit binary strings ($b \geq 32$ is fixed). A key can be drawn uniformly randomly from I', or it can be biased (e.g., for balancing the keys among the groups). For a given key $k' \in I'$, denote k for the 32-bit suffix of k' (thus $k \in I$ and $k = k_{31}k_{30}\ldots k_1k_0$). Throughout the discussion below, we shall refer to k rather than to the original k'.

The XOR metric defines the distance between two IDs j and k as $d(j,k) = \sum_{\nu=0}^{31} |j_\nu - k_\nu| \cdot 2^\nu$. Let $c(j,k)$ be the number of bits in the common prefix of j and k. The metric $d(j,k)$ has the following properties: if $d(i,k) = d(j,k)$ for any k, then $i = j$; if $d(i,k) \leq d(j,k)$, then $c(i,k) \geq c(j,k)$; and $d(j,k) \leq 2^{32-c(j,k)} - 1$. The XOR metric is a refinement of longest-prefix matching. If j is the unique longest-prefix match with k, then j is the closest to k in terms of the metric. Further, if two peers share the longest matching prefix, the metric will break the tie. The Kademlia DHT [8] also uses the XOR metric. The peer p' that minimizes $d(k,p), p \in U$ is "responsible" for key k.

The lookup algorithm. Suppose peer p_s wants to look up key k. Peer p_s determines the peer in its routing table that is closest to k in terms of the XOR metric, say p_j. Then p_s forwards the message to p_j. The process continues, until the message with key k reaches a peer p_d such that the closest peer to k in p_d's routing table is p_d itself. p_d is trivially the peer responsible for k.

If the set of groups form a proper nesting, then it is straightforward to show that the number of hops in a lookup is at most $L+1$, where L is the depth of the partial-order tree. In the first hop the message will be sent to a peer p_1 that is in the same group, say G, as p_d. The message remains in G until it arrives at p_d. Each peer in TOPLUS mimics a router in the sense that it routes messages based on a generalization of longest-prefix matching of IP addresses using highly-optimized algorithms.

Overlay maintenance. When a new peer p joins the system, p asks an arbitrary existing peer to determine (using TOPLUS) the closest peer to p (using p's IP address as the key), denoted by p'. p initializes its routing table with p''s routing table. Peer p's routing table should then be modified to satisfy a "diversity" property: for each peer p_i in the routing table, p asks p_i for a random peer in p_i's group. This way, for every two peers in a group G, their respective sets of delegates for another group G' will be disjoint (with high probability). This guarantees that, in case one delegate fails, it is possible to use another peer's delegate. Finally, all peers in p's inner group must update their inner group tables.

Note that groups, which are virtual, do not fail (individual peers fail) but they can be partitioned or aggregated at runtime when needed.

Data caching. In DHT storage systems, caching can help reduce access times by creating local copies of popular data so as to avoid fetching the original data from the responsible peer. The caching mechanisms that have been proposed in major DHT systems consist in replicating the data along the lookup path of a query after a successful lookup. This approach suffers from two major drawbacks: First, it assumes that the lookup paths of two requests for a given key converge quickly, which is not always the case (especially when the number of peers grows). Second, the data is typically not cached close to the clients: even if a local copy does exist near the client, that copy is unlikely to be on the lookup path.

In addition to improving lookup performance, a major objective of data caching is to save on bandwidth costs by caching content within a given network (company, ISP, AS, etc.). Because of its hierarchical organization derived from IP prefixes, TOPLUS can straightforwardly fulfill this goal and provide a powerful caching service for network infrastructures.

Suppose that a peer p_s wants to obtain the file f associated with key $k \in I$, located at some peer p_d (see Figure 2). It would be preferable if p_s could obtain a cached copy of file f from a topologically close peer. To this end, suppose that some group $G \in \mathcal{G}$, with network prefix $w.x.y.z/r$, at any tier, wants to provide a caching service to the peers in G. Further suppose all pairs of peers in G can send files to each other relatively quickly (high-speed LAN) or at least quicker than to any other peer outside of G. Peer $p_s \in G$ creates a new key, k_G, which is equal to k but with the first r bits of k replaced with the first r bits of $w.x.y.z/r$. Peer p_s then looks up the peer p_G responsible for k_G. This peer is obviously inside group G and the lookup message will not leave the group.

If p_G has a copy of f (cache hit), then it will serve the file to p_s at a relatively high rate. Otherwise (cache miss), p_G will use key k to obtain f from the global lookup service. After downloading the file, p_G will send f to p_s and create a local cache copy for other users of G. Another peer $p \in G$ will obtain the file faster by directly downloading it from p_G (Figure 2). It is obviously possible to cache data at multiple levels of the hierarchy (e.g., company, ISP, AS) and hence implement a hierarchical Internet cache.

Performance. We have measured the efficiency of TOPLUS with $1,000$ nodes using IP network prefixes obtained from several BGP tables and routing registries.[2] When constructing a TOPLUS hierarchy from a direct mapping of IP prefixes to groups, we obtain an average stretch of 1.17, that is, a query in TOPLUS takes on average only 17% more time to reach its destination than using direct IP routing. Unfortunately, this hierarchy has many tier-1 groups, which leads to large routing tables. We can reduce their size by "aggregating" small groups that have a long prefix into larger groups not present in our IP prefix sources. Even after aggressive aggregation of large sets of tier-1 groups into coarse 8-bit prefixes, we have measured a remarkably small stretch of 1.28. Complete evaluation results are available in [5].

Limitations. In designing TOPLUS to optimize topological locality, we had to sacrifice some other desirable properties of P2P lookup services. In particular, TOPLUS suffers from limitations related to the non-uniform population of the ID space (peers in sparse regions may be overloaded if keys are uniformly distributed over the ID space), the lack of support for virtual peers, and the vulnerability to correlated peer failures (a whole inner group being unavailable, e.g., due to a network partition). Some solutions to these problems are discussed in [5].

[2]BGP tables were provided by the University of Oregon and by the University of Michigan and Merit Network. Routing registries were provided by Castify Networks and RIPE.

3 Content Indexing

A major limitation of DHT lookup services is that they only support exact-match lookups: one needs to know the exact key (identifier) of a data item to locate the peer responsible for 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 section, we propose to augment DHT-based P2P systems with mechanisms for locating data using incomplete information. We do not aim at answering complex database-like queries, but rather at providing practical techniques for searching data within a DHT. Our mechanisms rely on indexes, stored and distributed across the peers 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 her original query; the DHT lookup service 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 DHT lookup service, including TOPLUS, and thus benefit from any advanced features implemented in the DHT (e.g., replication, load-balancing).

P2P storage system. We assume an underlying DHT-based P2P data storage system in which each data item is mapped to one or several peers. Example of such systems are Chord/DHash/CFS [2] and Pastry/PAST [9]. We shall 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 (in their simplest form, they can be file names). Let $h(descriptor)$ be a hash function that maps identifiers to a large set of numeric keys. The peer 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 lookup service to determine the peer responsible for f. In order to find f, a peer p has to know the numeric key or the complete descriptor.

```
<article>
  <author>
    <first>John</first>
    <last>Smith</last>
  </author>
  <title>TCP</title>
  <conf>SIGCOMM</conf>
  <year>1989</year>
  <size>315635</size>
</article>
```

d_1

```
<article>
  <author>
    <first>John</first>
    <last>Smith</last>
  </author>
  <title>IPv6</title>
  <conf>INFOCOM</conf>
  <year>1996</year>
  <size>312352</size>
</article>
```

d_2

```
<article>
  <author>
    <first>Alan</first>
    <last>Doe</last>
  </author>
  <title>Wavelets</title>
  <conf>INFOCOM</conf>
  <year>1996</year>
  <size>259827</size>
</article>
```

d_3

Figure 3: Sample File Descriptors.

```
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
```

Figure 4: Sample File Queries.

Data descriptors and queries. In the rest of this section, we shall assume that descriptors are semi-structured XML data, as used by many publicly-accessible databases (e.g., DBLP). Examples of descriptors for bibliographic data are given in Figure 3. These descriptors have fields useful for searching files (e.g., author, title), as well as fields useful for administering the database (e.g., size).

To search for data stored in the peer-to-peer network, we need to specify broad queries that can match multiple file descriptors. For this purpose, we shall use a subset of the XPath XML addressing language, 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 using various types of predicates and wildcards. 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.

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.[3] 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 4 is the most specific query for descriptor d_1 in Figure 3.

[3] 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.

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. Note that the covering relation introduces a partial ordering on the queries.

Indexing algorithm. 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-peer (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 in the P2P system. 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 P2P storage system returns the file (or indicates the peer responsible for that file). By iteratively querying the index service, a user can traverse upward the partial order graph of the queries and discover all the indexed files that match her broad query.

In order to manage indexes, the underlying P2P storage system must be slightly extended. Each peer 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 peer 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 peer 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 peer 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 peer responsible for each k_i in the P2P network. We iterate the process shown for q to every q_i, and we continue recursively until all the desired index entries have been created.

Indexing example. To best illustrate the principles of our indexing techniques, consider a P2P bibliographic database that stores the three files associated to the descriptors of Figure 3. We want to be able to look up publications using various combinations of the author's name, the title, the conference, and the year of publication. A possible hierarchical indexing scheme is shown in Figure 5. Each box corresponds to a distributed index, and indexing keys are indicated inside the boxes. The index at the origin of an arrow stores mappings 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 most specific 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 6 (left). The top-level *Publication* index corresponds to the raw entries stored in the underlying P2P storage system: complete keys 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 peer in the P2P network, 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 (right) details the individual query mappings stored in the indexes of Figure 6 (left). 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 her query using d_1 and d_2.

Lookups. We can now describe the lookup process more formally. When looking up a file f using a query q_0, a user first contacts the peer p responsible for $h(q_0)$. That peer may return f if q_0 is the most specific query for f, or a list of queries $\{q_1, q_2, \ldots, q_n\}$ such that the mappings $(q_0; q_i)$, with $q_0 \sqsupseteq q_i$, are stored at p. 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_0 to f ("guided tour"). The lookup process can be interactive, i.e., the user directs the search and restricts her query at each step, or automated, i.e., the system recursively explores the indexes and returns all the file descriptors that match the original query.

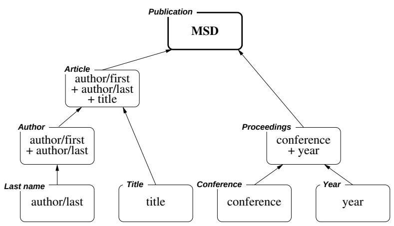

Figure 5: Sample indexing scheme for a bibliographic database.

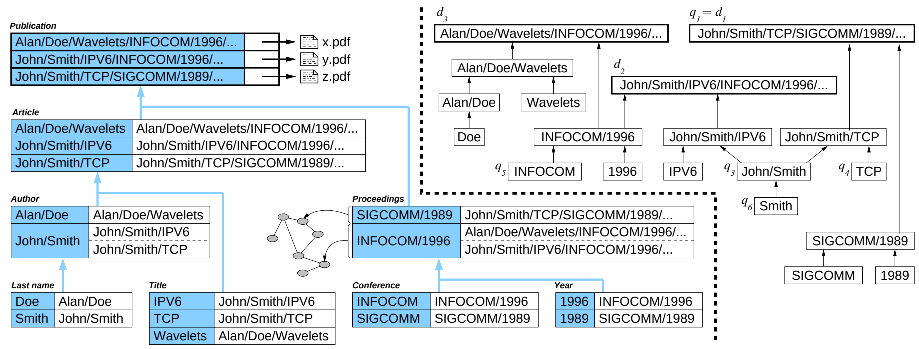

Figure 6: Sample distributed indexes (left) and query mappings (right) for the three documents of Figure 3 and the indexing scheme of Figure 5.

Lookups may require several iterations when the most specific query for a given file is not known. Deeper index hierarchies 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 other indexes. In particular, the size of the lists (result sets) returned by the index service 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.

When a user wants to look up a file f using a query q_0, it may happen that q_0 is not present in any index, although f does exist in the peer-to-peer system and q_0 is a valid query for f. It is still possible to locate f, by (automatically) looking for a query q_i such that $q_i \sqsupseteq q_0$ and q_i is on some index path that leads to f. For instance, given the distributed indexes of Figure 6, it appears that query q_2 in Figure 4 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.

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.

Index entries can also be created dynamically to adapt to the query patterns of the users. For instance, a user who tries to locate a file f using a non-indexed query q_0, 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 P2P system to speed up accesses to popular files. Assume that each peer 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 (i.e., the least used entries in the cache are replaced by the new ones once there is no cache space left), 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 popularity.

An in-depth discussion of the properties of our indexing techniques, such as space-efficiency, scalability, loose coupling, flexibility and adaptability, can be found in [4], together with a detailed experimental evaluation.

4 Conclusion

DHTs suffer from a number of limitations over unstructured P2P networks due to the rigidness of their logical organization. We have discussed some of these limitations and presented several techniques to help alleviate them. TOPLUS integrates network topology in its design, by organizing peers in a group hierarchy defined by IP prefixes that map directly to the underlying topology. The TOPLUS architecture also allows for the straightforward deployment of distributed caches in order to reduce transfer delays and limit the external traffic of a given network. Finally, we have studied the problem of looking up information in a DHT using incomplete information—a major improvement over the exact-match lookup of DHTs. Our distributed indexing techniques can be used to locate data in a DHT matching a broad query. We hope that these various improvements can contribute to the large-scale deployment of DHTs in the Internet and bring them on par with unstructured P2P networks in terms of popularity.

References

[1] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Topology-aware routing in structure peer-to-peer overlay network. In *International Workshop on Future Directions in Distributed Computing (FuDiCo)*, pages 103–107, June 2002.

[2] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, pages 202–215, October 2001.

[3] L. Garcés-Erice, E.W. Biersack, P. Felber, K.W. Ross, and G. Urvoy-Keller. Hierarchical peer-to-peer systems. *Parallel Processing Letters*, 13(4):643–657, 2003.

[4] L. Garcés-Erice, P. Felber, E.W. Biersack, K.W. Ross, and G. Urvoy-Keller. Data indexing in DHT peer-to-peer networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 200–208, March 2004.

[5] L. Garcés-Erice, K.W. Ross, E.W. Biersack, P. Felber, and G. Urvoy-Keller. TOpology-centric Look-Up Service. In *Proceedings of COST264/ACM 5th International Workshop on Networked Group Communications (NGC)*, volume 2816 of *LNCS*, pages 58–69. Springer, September 2003. Best paper award.

[6] A.D. Joseph, B.Y. Zhao, Y. Duan, L. Huang, and J.D. Kubiatowicz. Brocade: Landmark routing on overlay networks. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *LNCS*, pages 34–44. Springer, March 2002.

[7] B. Krisnamurthy and J. Wang. On network-aware clustering of Web sites. In *Proceedings of SIGCOMM*, pages 97–110, August 2000.

[8] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer informatic system based on the XOR metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *LNCS*, pages 53–65. Springer, March 2002.

[9] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, pages 188–201, October 2001.

[10] S. Shenker, S. Ratnasamy, M. Handley, and R. Karp. Topologically-aware overlay construction and server selection. In *Proceedings of INFOCOM*, pages 1190–1199, June 2002.

[11] J. Wang. *Network Aware Client Clustering and Applications*. PhD thesis, Cornell University, May 2001.

[12] J. Xu, A. Kumar, and X. Yu. On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks. *IEEE Journal on Selected Areas in Communications*, 22(1):151–163, January 2004.