Data Indexing and Querying in DHT Peer-to-Peer Networks

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

I. Introduction

Peer-to-peer DHT systems, such as Chord [8], CAN [5], Pastry [6], or Tapestry [11], 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 and tend to submit broad queries.

In this paper, we describe techniques for indexing data stored in peer-to-peer networks, and discovering the resources that match a given user query. Our system creates multiple indexes, organized hierarchically, which permit users to access data in many different ways. 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 the original query; the DHT can be recursively queried until the user finds the desired data items. The data itself is stored on only one (or few) of the nodes.

Our indexing techniques have several interesting properties, such as good scalability, loose coupling between data and indexes, decentralized architecture, and reasonably-small space requirements. Look-up times depend on the "precision" of the initial query: broad queries incur higher lookup times than specific queries. Note that we do not aim at answering complex database-like queries, but rather at providing practical techniques for searching data using more advanced tools than exact or simple keyword lookups.

II. Related Work

INS/Twine [1] is an architecture for intentional resource discovery, which allows client applications to easily locate services and devices in large scale environments. INS/Twine works by setting up a number of resolvers, which collaborate as peers to distribute resource information and to resolve simple queries. As resources are expected to be rather volatile, they must refresh their descriptions periodically. Furthermore, nodes limit the number of resources that are registered using popular keys, to avoid being overwhelmed with advertisements.

INS/Twine builds on top of a distributed hash table (DHT), such as Chord [9]. 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.

In [4], the authors discuss techniques for performing complex queries in DHT-based peer-to-peer networks, using traditional relational database operators (selection, projection, join, grouping and aggregation, and sorting) and elaborate text retrieval techniques. For instance, the authors propose to achieve substring matching by splitting each strings into distinct n-grams (a sequence of n consecutive characters) used as keys to store file IDs in the DHT. Upon lookup, the query string is also split into n-grams that are looked-up individually; a file ID that is returned by each n-gram lookup is a possible match. According to the research directions outlined in [4], complex query processing in peer-to-peer networks is still a very open issue.

In [3], 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 [7], the same authors extend the CAN [5] system to support the basic range operation on data shared in the form of database relations. This work represents an important step toward advanced, database-like query processing in peer-to-peer systems.

```
<song>
  <artist>David Bowie</artist>
  <title>Changes</title>
  <album>Hunky Dory</album>
  <size>3156354</size>
</song>
```

d_1

```
<song>
  <artist>David Bowie</artist>
  <title>Amsterdam</title>
  <album>At the Beeb</album>
  <size>4123523</size>
</song>
```

d_2

```
<song>
  <artist>Jacques Brel</artist>
  <title>Amsterdam</title>
  <album>Olympia 1964</album>
  <size>2598273</size>
</song>
```

d_3

Fig. 1. Sample File Descriptors.

III. System Overview

A. System Model and Definitions

A distributed hash table (DHT) system maps keys to nodes in a peer-to-peer infrastructure. For a given key k, any node n can use the DHT substrate to determine the current live node n' that is responsible for k.

We consider a distributed data storage system, in which each data item (or file) is mapped to one or several peer nodes. Files are identified by *descriptors*, which are textual, human-readable descriptions of the file's content. We assume that descriptors are semi-structured XML data. Examples of descriptors for a music file sharing system are given in Figure 1.

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. In order to find f, a node n has to know the numeric key or the complete descriptor.

To lookup data stored in the peer-to-peer substrate, we use a subset of the XPath XML addressing language [10]. 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 steps designate 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. We say that an XML document (i.e., a file descriptor) *matches* an XPath expression when the evaluation of the expression 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.[1] We call this expression the *most specific* query for d. 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 for descriptor d_1 in Figure 1.

q_1 = `/song[artist/David Bowie][title/Changes]` $\cdots$
  $\cdots$ `[album/Hunky Dory][size/3156354]`
q_2 = `/song[artist/David Bowie][title/Amsterdam]`
q_3 = `/song/artist/David Bowie`
q_4 = `/song/title/Changes`
q_5 = `/song/title/Amsterdam`
q_6 = `//David Bowie`

Fig. 2. Sample File Queries.

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; in particular, we say that q' covers d when $q' \sqsupseteq q$ and q is the most specific query for d. Abusing the terminology, we often use the term "key" instead of "query"when the context is clear.

In Figure 2, omitting self-covering relations, we have: $q_3 \sqsupseteq q_1$, $q_3 \sqsupseteq q_2$, $q_4 \sqsupseteq q_1$, $q_5 \sqsupseteq q_2$, $q_6 \sqsupseteq q_1$, $q_6 \sqsupseteq q_2$, $q_6 \sqsupseteq q_3$. Given the descriptors of Figure 1, q_1 and q_4 cover d_1; q_2 covers d_2; q_3 and q_6 cover d_1 and d_2; and q_5 covers d_2 and d_3.

B. 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) lookup service provided by DHT. The goal of our architecture is to also offer access to f using less specific queries that cover d.

Similarly to INS/Twine, we generate multiple keys for a given descriptor. 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. In addition, we do not restrict queries to be prefix subsequences of the

[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.

descriptors and we allow for multiple levels of indexing, in order to increase scalability.

Roughly speaking, our system works as follows: Given a file f and its descriptor d, with a corresponding most specific query q, we first store f at the node identified by the key $k = h(q)$. We generate a set of "plausible" queries $Q = \{q_1, q_2, \ldots, q_l\}$ such that each $q_i \sqsupseteq q$. The way in which we choose these queries will be discussed later. We then compute the numeric key $k_i = h(q_i)$ for each of the queries, and we store a tuple $(q_i; q)$ at the node identified by k_i in the DHT. We apply the process shown for q to every q_i, and we continue recursively until the resulting queries become too "generic".

Note that the "covered-by" relationship creates a partial order on all the queries that match a given file descriptor. One way for representing these queries is to organize them hierarchically in a rooted directed acyclic graph. The file's most specific query is located at the root, and each edge represents an index entry that maps the child query to the parent query. Multiple paths lead to the root, and the number of lookups necessary to locate a file are the number of nodes along the shortest paths from the initial query to the root.

C. Lookups

When looking up a file f using a query q_0, a user first contacts the node n associated to $h(q_0)$. That node may return f if q_0 is the most specific query for f, or a list of queries $Q = \{q_1, q_2, \ldots, q_n\}$ such that the tuples $(q_0; q_i), q_i \in Q$ 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 lead from q_0 to f.

For instance, given the descriptors and queries of Figures 1 and 2, we can create the following index entries: $(q_2; d_2)$, $(q_3; q_2)$, $(q_3; d_1)$, $(q_4; d_1)$, $(q_5; q_2)$ $(q_5; d_3)$, and $(q_6; q_3)$. Given q_3, a user will first obtain q_2 and d_1; the user will query the system again using q_2 and obtain d_2; the user can finally retrieve the two files matching its query using d_1 and d_2.

Lookups require several iterations when the most specific query for a given file is not known. On the one hand, the higher the index hierarchy, the more iterations are necessary to find a file. On the other hand, higher index hierarchies are generally also more space-efficient, as each index factorizes in a compact manner the queries of its child indexes. There is therefore a tradeoff between space requirements 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, whereas f exists in the peer-to-peer system and q_0 is a valid index key for f. To locate f, one can (automatically) look for a query q_i such that $q_i \sqsupseteq q_0$, $q_i \sqsupseteq q_j$, $q_0 \sqsupseteq q_j$, and q_i and q_j are on some index path that leads to f.

For instance, given the descriptors and queries of Figures 1 and 2 and the sample index entries given above, the query:

$q_0 =$ `/song[artist/David Bowie][album/Hunky Dory]`

is not present in any index. We can however find q_3, such that $q_3 \sqsupseteq q_0$ and there exists an index entry $(q_3; d_1)$. Therefore, the file associated to d_1 can be located, although at the price of a higher lookup cost. We believe that it is natural for more effort to be required when lookups are performed with less information.

IV. 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 our music file sharing example, indexing a file by its size is useless, as users are unlikely to know the size beforehand. However, indexing the files under the artist, title, and/or album 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 requirements. In contrast, a very popular file can be linked to high in the hierarchy to short-circuit some indexes and speed up lookups. For instance, given the descriptors and queries of Figures 1 and 2, one can add both the $(q_6; q_3)$ and $(q_6; d_1)$ index entries at the node identified by $h(q_6)$ to speed up searches for the popular file described by d_1.

Note also that more generic queries can be obtained from more specific queries by removing only portions of element names. For instance, one can create an index with all the files of an artist that start with the letter "A", the letter "B", etc. One can also envision to use techniques similar to those discussed in [4] for substring matching.

In general, determining good decompositions for indexing each given descriptor type (e.g., music files, movies, pictures, etc.) requires human input. However, if we have information about the nature of the user queries—for instance, we can construct a synopsis of the users queries observed over a period of time—then we can use automated tools to determine the combinations of elements in the descriptor type that are likely to be used for queries,

and construct indexes accordingly. These issues are open for further research.

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. If a file has to be deleted, then we have to recursively find all the indexes that refer to the descriptor of that file, and remove those entries that do not refer to any other file. 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.

V. Evaluation

Our data indexing techniques have several interesting properties. We outline some of these properties below:

- *Space efficient:* The hierarchical organization allows for space-efficient data indexing. First, as indexes contain key-to-key mappings, the data items are not stored on multiple nodes (unlike for instance INS/Twine). 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 descriptors and queries of Figures 1 and 2, the index entry $(q_6; q_3)$ is on index paths to both d_1 and d_2).
- *Scalability:* As data items may be accessed through distinct paths and are referred to in distinct indexes, the lookup load is expected to be spread across multiple indexes (and thus multiple nodes). 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 corresponding to the complete key of the data need to be updated. Indexes do not need to be updated. This is 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 indexed in many different manners, or short-circuits some levels in the indexing hierarchy.
- *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 corresponds 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 a genuine file by advertising it under many different names.

Although there are similarities between our indexing scheme and INS/Twine, there are also several notable differences between both approaches to resource discovery.

In particular, INS/Twine proposes an approach specialized for the discovery of services and devices using intentional descriptions and its design has been driven by the nature of the data that is registered in the DHT. In contrast, we aim at providing *generic* mechanisms for indexing any kind of data. As data items may be very large (e.g., music files), we maintain key-to-key (instead of key-to-data) mappings. For improved scalability, index entries can be organized hierarchically based on query containment relationships. This architecture allows for space-efficient indexing, helps to avoid the "node overwhelming" problem, and makes data updating easier (although lookups become slower as queries become less specific).

In addition, 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 (as long as covering relationships are preserved), data can be looked up using more expressive and selective queries that do not, we believe, require the presence of a resolver.

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. This process can either be driven interactively by the user, or all matching data items can be recursively collected automatically.

Although our data indexing techniques permit looking 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 [2] for music files.

REFERENCES

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

[2] Gracenote. CDDB. http://www.cddb.org.

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

[4] 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 IPTPS02*, Cambridge, USA, March 2002.

[5] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, 2001.

[6] 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.

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

[8] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM*, 2001.

[9] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

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

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