Performance of Full Text Search in Structured and Unstructured Peer-to-Peer Systems

Yong Yang, Rocky Dunlap, Michael Rexroad and Brian F. Cooper
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332, USA

***Abstract*— While structured P2P systems (such as DHTs) are often regarded as an improvement over unstructured P2P systems (such as super-peer networks) in terms of routing efficiency, it is not clear which architecture is better for full text search. This paper provides a quantitative comparison of full text keyword search in structured and unstructured P2P systems. We examine three techniques (and optimizations to those techniques) proposed in the literature: using a DHT along with inverted lists and Bloom filters; using a super-peer network; and using a random walk over an unstructured network. We use real Web documents and user queries to measure the cost for both document publishing and query processing, in terms of bandwidth and response time. Our results show that all three techniques use roughly the same bandwidth to process queries (with the super-peer technique having a slight edge). The structured network provides the best response time (30 percent better than a super-peer network), but has a high cost of document publishing, using six times as much bandwidth as the super-peer system. The random walk technique requires no publishing, but has a very long response time unless multiple random walks operate in parallel.**

I. Introduction

Structured peer-to-peer networks (e.g., distributed hash tables [1], [2], [3], [4], [5]) are generally regarded as a significant improvement over the original unstructured peer-to-peer protocols for many tasks. However, it is not clear which architecture is better for full text keyword search of documents. Although structured networks offer efficient message routing, it has been demonstrated that a straightforward application of keyword searching techniques to DHTs results in performance that is unfeasible by at least an order of magnitude [6]. In contrast, unstructured networks provide straightforward support for full text search since peers (or super-peers) can store and index whole documents. Unfortunately, it is often difficult to achieve scalability for unstructured networks for any task, including full text search. Despite these difficulties, developing a peer-to-peer infrastructure for full text search is still a sought-after goal, and such a system could be used in a variety of searching applications (from web search to searching networks of digital libraries to searching for documents in far-flung offices of a large corporation).

Recent research has resulted in significantly improved techniques for conducting full text search in DHTs [7], [8] and significantly improved techniques for routing searches in unstructured networks [9], [10], [11], [12], [13], [14], [15], [16], [17], [18]. However, it is still not clear whether the latest techniques for full text searching in DHTs result in better performance than leading techniques for searching in unstructured networks. In particular, there has been little study of whether techniques proposed in the literature are better than the popular and widely deployed super-peer networks used in systems such as Kazaa and the latest versions of Gnutella. Investigators proposing new techniques usually compare only to similar techniques (e.g., comparing structured to structured or unstructured to unstructured). Zhong et al [19] have made some progress in comparing different types of techniques. However, they focus on indexing strategies, especially in DHT systems. Although they examine one unstructured technique (multicast-based flooding) they do not consider the widely used super-peer strategy, or walk-based techniques proposed by several researchers [9], [10], [11], [17]. As such, a direct comparison of structured and unstructured techniques is still needed.

In this paper, we experimentally compare several leading techniques for full text search in structured and unstructured peer-to-peer systems. In particular, we examine:

- Using a DHT (specifically, Chord [1]) along with Bloom filters and several other optimizations [7]
- Using a super-peer network like that in Kazaa [16]
- Using a random-walk search along with several optimizations [10], [9], [18]

These techniques are representative of the progress made in both structured and unstructured systems. Of course, they do not cover the full spectrum of available techniques. However, we believe our study provides valuable insight into the relative strengths and weaknesses of structured and unstructured approaches for full text search.

We conducted our performance evaluation using real web pages and user queries. We downloaded documents from 1,000 web sites, and each peer in our experiments stored the data from one web site. Then, each peer "published" their content to the network. For the DHT approach, publishing involves inserting indexes for terms into the Chord ring, while for the super-peer approach publishing means sending a full copy of the peer's index to a super-peer. For the random walk technique, no "publishing" is needed since each peer processes searches over their own data. Next, we used queries from the search.com search engine to search the data. We measured the bandwidth cost for both the publishing and searching phases, and the relative search latency of the searching phase, for each

Appears in IEEE INFOCOM 2006. ©2006 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

of the three techniques under study.

Our results show that all three techniques use roughly the same bandwidth to process queries, although the super-peer network is slightly better (11 percent less bandwidth than the structured technique and 14 percent less than random walks). The structured network provides the best response time (30 percent better than a super-peer network), but has a high cost of document publishing, using six times as much bandwidth as the super-peer system. The random walk technique requires no publishing, but has a very long response time unless multiple random walks operate in parallel.

The remainder of this paper is organized as follows. In Section II, we summarize the techniques under study and describe our evaluation setup. In Section III we examine searching in structured networks in detail. Next, we examine searching in unstructured networks: super-peer techniques in Section IV, and random walk techniques in Section V. We then compare the performance of all three techniques in Section VI. In Section VII we examine related work and in Section VIII we present our conclusions.

II. Searching Overview and Evaluation Setup

A. Searching Overview

We now briefly summarize full text searching and the three techniques under study. Each technique is described in more detail in the following sections. This overview is intended to provide the reader with a high-level picture of the different techniques and their tradeoffs.

In full text keyword search the goal is to find text documents that match keyword queries. Many techniques for matching queries and documents have been developed in the field of information retrieval (IR) [20]. In particular, there are a variety of ways to measure how well a query "matches" a document, including the conjunctive query model (a document must match all terms in the query), the vector space model (a document is given a score based on how many of the query terms it contains), the TF/IDF weighting technique for the vector space model (a document is given a score based on the number of query terms it contains and whether those query terms appear infrequently in other documents) and so on.

Information retrieval techniques usually focus on a centralized search server that either stores the documents directly or processes searches over documents stored at remote servers [21], [22]. In contrast, peer-to-peer full text search systems are decentralized and massively distributed, with no "central server" per se. Decentralization by itself is not necessarily desirable, and introduces complexity and scalability challenges. However, a peer-to-peer system may be chosen over a centralized IR server for a number of reasons, including the potential for massively parallel processing of searches, rapidly changing data that makes it difficult to keep a central IR server up to date, non-functional considerations (such as the unwillingness of peers to relinquish control to a central server), and so on. Despite decentralization, such peer-to-peer systems attempt to provide the same or similar search semantics as centralized IR servers, namely finding documents that closely match user queries.

The traditional approach for conducting full text searches is to use an *inverted index*, which stores for each term a list of documents containing that term. (The term "inverted" is in contrast to the documents themselves, which can be seen as storing for each document a list of the terms it contains.) Inverted indexes may also contain other information, such as the number of occurrences of a term in a document, the position of the terms in the document, and so on. Most searching operations can be efficiently executed by intersecting or unioning the inverted lists for different terms. For example, to find the documents that match the search "apple banana," an IR system can find the "apple" inverted list and the "banana" inverted list, and then take the intersection to construct a list of documents that contain both "apple" and "banana." This process is significantly faster than scanning all of the documents looking for occurrences of "apple" and "banana."

1) Structured peer-to-peer searches: Distributed hash tables provide an interface to *put* and *get* key/value pairs. A straightforward way to implement full text search is to use DHTs to store and retrieve inverted lists. The "keys" inserted into the DHT would be the terms, while the "values" inserted would be the associated inverted lists. To conduct a search, we would perform a lookup for each query term to retrieve its inverted list, and then intersect the inverted lists to obtain the list of matching documents.

However, it is unlikely that we would insert whole inverted lists into the DHT. Instead, the inverted lists would be constructed incrementally by "publishing" documents. Such publishing would take place when a peer joins the network, or adds a new document to a collection. To publish a document, the peer must parse its documents to extract the terms, and then send a $publish(term, document_{ID})$ message for each term. The DHT would route each publish message to the peer responsible for the given term; this peer would then add the document ID (and the ID of the peer holding the document) to its inverted list for that term.

Li et al [6] have demonstrated that this simple scheme is by itself inefficient. The key problem is the cost to transport the inverted lists between peers so they can be intersected, since inverted lists are likely to be large. Reynolds and Vahdat [7] suggest several optimizations to reduce the cost of transporting inverted lists. In particular, they propose encoding inverted lists as Bloom filters to reduce their size, and caching Bloom filters at multiple peers to reduce the need to retrieve the original inverted list. We describe these optimizations, and evaluate their effectiveness, in Section III.

2) Super-peer searches: In a super-peer network, each peer is classified as either a "super-peer" or "leaf peer." Each leaf peer is assigned to a single super-peer, while super-peers may be assigned multiple leaf peers. Each peer constructs a full inverted index over their documents. Leaf peers send a copy of their index to their assigned super-peers during the "publishing" phase. This allows super-peers to process searches over both their own content and the content of their

leaf peers. When a search originates at a leaf peer, it is forwarded to that peer's super-peer, who processes the query and forwards it to other super-peers. In this way, multiple super-peers cooperate to process searches, while leaf peers do not process any searches. This scheme is designed to take advantage of the wide resource heterogeneity observed among peers in real systems: more powerful peers with high-bandwidth connections are chosen as super-peers.

The primary cost for processing searches in super-peer networks is forwarding of the searches themselves and their associated search results. Although inverted lists are copied to super-peers during the publishing phase, they do not need to be transported during the searching phase itself. Each super-peer has a set of complete inverted indexes, and can completely process searches over its own content and that of its leaf peers. However, searching can still be expensive, since searches are flooded to all super-peers (or all super-peers within a TTL horizon). In particular, super-peers themselves may become heavily loaded.

Even though the super-peer/leaf peer distinction adds structure to the peer-to-peer network, super-peer networks are still considered by many to be "unstructured" networks, because the structure is not particularly rigid (e.g., super-peer neighbors are not constrained by protocol rules) and the structure does not enforce performance guarantees (such as the $O(logN)$ hops of many DHTs). We describe our implementation of the super-peer technique in detail in Section IV.

3) Random-walk searches: In an unstructured network without super-peers, every peer processes and forwards queries. Such networks are usually *partially connected*, which means that each peer has a small number of neighbors. In the original Gnutella protocol, each peer that received a search message both processed the search and forwarded it to all of its neighbors. A peer that received the same search message multiple times ignored all but the first message. In this way, searches were flooded to the whole network, an approach that scales poorly.

Several investigators have demonstrated that random walk searches are significantly more scalable than flooding [9], [10], [23]. In a random walk search, a peer forwards searches to one randomly chosen neighbor, instead of all of its neighbors. This causes the search message to "walk" randomly around the network, until a predefined number of results have been found, or a predefined TTL has been reached. Because random walk searches usually find enough results after reaching only a fraction of the network, the cost of flooding the message to every peer is avoided.

Each peer in the network constructs an inverted index over its own content. Since each peer also processes searches directly over its content, no "publishing phase" is necessary. The primary cost in processing searches is forwarding the search messages and search results themselves. Like the super-peer network, inverted lists do not need to be forwarded between peers during the searching process.

By constraining the unstructured network in a few simple ways, the performance of random walk searches can be further improved. For example, by fixing the degree (e.g., number of neighbors) of peers based on the popularity of their content, we can improve the efficiency of random walk searches by a factor of two or more [18], [24]. In Section V we describe and evaluate random walk searches in such networks in detail.

TABLE I
DOCUMENT AND QUERY SET PROPERTIES

Total number of source websites	1000
Total number of documents	89756
Total size of documents before preprocessing (GB)	2.03
Average number of distinct terms per document	238.70
Total number of queries	81206
Average number of terms per query	1.995
Number of unique queries	58058

A variety of other variations of random walks and walk-based searches have been proposed [9], [10], [11], [12], [13], [14], [15], [17], [25]. It is out of the scope of this paper to evaluate and compare the myriad techniques that have been proposed. Our experiments can be seen as evaluating the inherent performance of random walk searches, and this performance may be improved in certain situations by these different techniques.

B. Evaluation Setup

Our evaluation is based on processing real user queries over real Web documents. We assume the conjunctive query model, in which "matching" documents must contain all of the query terms. The structured searching system we studied only supports conjunctive queries due to the use of Bloom filters, and to support an apples-to-apples comparison we similarly restricted the query model in the unstructured networks to the conjunctive query model. The document set is a collection of web documents, which we downloaded from 1,000 randomly-chosen websites. We pre-processed documents to filter out HTML tags, remove duplicate occurrences of the same terms in each document, and stem all terms. Stemming reduces distinct terms to their common grammatical root [20] to improve matching accuracy. For example, *connect* is the stem for *connected, connecting, connection,* and *connections*. Table I shows the properties of the document set.

The queries come from the publicly available search.com database. We pre-processed each query by stemming its terms. Table I also summarizes the properties of the query set. Because the query set was a trace of real user queries, it contained duplicates, and in fact some popular queries appeared many times.

For each searching system (structured, super-peer and random walk), we started 1,000 peers, and assigned each peer the content of one website. A random peer is chosen to issue each query. We did not assume that peers failed, or frequently joined and left the network (e.g., "churn.") Failures and churn are likely to have large impact on the performance of the network, but their effects are complex and examining them is outside the scope of this paper.

We evaluated both phases of full text keyword searching:

document publishing and query processing. We measured the bandwidth used and the response time for both phases. Bandwidth is measured by the total communication cost when publishing one document or processing one query. The response time consists of message transmission time and local processing time. We assume that compared to the transmission time, the local processing time is negligible. The transmission time depends on the characteristics of the network, and varies for a number of reasons besides the specific peer-to-peer technique being studied. In order to understand the inherent efficiency of different techniques, separate from the characteristics of the underlying network, we counted the number of hops taken by messages in place of wall-clock response time. Of course, our approach provides only an approximate picture of response time, and the relative performance of different techniques may be highly dependent on the network infrastructure. Nonetheless, our measurements of hop count still provide a useful comparison of the inherent response time of different techniques.

To evaluate the structured searching technique, we used the MIT Chord simulator [26] and augmented it with our Java implementation of the full text searching approaches of [7]. To evaluate the unstructured techniques, we used Topaz, a Java-based P2P toolkit we have developed. Topaz can be configured to support both the super-peer and random walk techniques. We do not report wall-clock timing results, so the efficiency of our implementation, our use of Java or the specifics of the machines we used in our evaluation are immaterial to our performance results.

We do not rank the web pages in the searching result, because the ranking rarely affects the bandwidth or response time, and the techniques we study do not directly support global ranking of results. If users want ranking, their client software can rank results after query responses are received. Therefore, searching results contain an unordered set of IDs for those documents in which all keywords in the query appear.

In our experiments, document IDs are 16 bytes, and all types of messages have a 10-byte header. The address of a peer is 4 bytes (e.g., the IPv4 address). Different encodings or compression techniques may change these values, and thus change the absolute values of our bandwidth measurements.

III. Structured peer-to-peer networks

We now examine the performance of a structured network for full text search. The techniques we examine in this section were proposed by Reynolds and Vahdat [7], and are based on storing inverted lists, keyed by the associated term, in a Chord network [1]. Bloom filters and caching are used to reduce the need to transport inverted lists during query processing.[1] We now describe both document publishing and searching in detail, and present performance numbers obtained using our implementation of these techniques over the Chord simulator [26].

[1]To ensure an apples-to-apples comparison, we do not implement another of Reynolds and Vahdat's proposals, *incremental results*, since it is not directly supported in super-peer networks.

TABLE II
Document publishing in the structured system

Average message size per term (bytes)	35.9263
Average hops per term publish message	2.6272
Average bandwidth per term (bytes)	94.3856
Average number of distinct terms per document	239
Average bandwidth per document (KB)	22.558

A. Document Publishing

Peers store inverted lists, and the collection of inverted lists at all peers represents a complete index over all of the content in the network. In the Chord DHT, each peer is assigned a b-bit ID in the interval $[0, 2^{b-1}]$. By doing all arithmetic modulus 2^b, Chord forms a circular ID space. The inverted list of each term is assigned to a Chord peer p when the b-bit term ID, computed by hashing the term, falls in the interval between p's ID and the ID of p's predecessor. Each peer may have multiple inverted lists.

To publish a document d, the publishing peer p_d sends a $publish(t_i, (d, p_d))$ message for each distinct term t_i in d. Chord routes the message to the peer p_{t_i} assigned to term t_i. Peer p_{t_i} adds the ID of document d (and the peer ID p_d) to the inverted list of term t_i. In the base technique, other information about t_i (such as the number of occurrences of term t_i in document d) is not included, although extensions of this technique might add such information to support more complex document ranking.

With high probability, the number of hops required to find the peer that maintains the inverted list for term t_i in an N-peer Chord is $O(logN)$. The bandwidth required to publish a term is the product of the number of hops and the size of the publish message:

$$Hops \times (s(Term) + s(DocID) + s(PeerAddress) + s(MessageHeader)) \quad (1)$$

where $s(x)$ is the size of string x. The bandwidth to publish a document would be the sum of the bandwidths for publishing individual unique terms. Note that we cannot easily batch multiple terms into the same publish message, since we do not know in advance which terms are destined for the same peer. Because all terms in a document can be published in parallel, the time required to publish a document is equal to the time to publish the term requiring the most hops.

Experimental results for publishing documents into Chord are shown in Table II. Note that the number of hops in the experiment result is less than $logN$. This is because we used the simulator's default finger table size of 50-entries, which is much larger than $log1000$; and a finger table larger than $logN$ results in a smaller hop count on average.

B. Query Processing

To process a query Q, a peer needs to find and intersect the inverted lists for each term in Q. The result of the intersection is a set of documents, each of which contain all the keywords in the query. More complex query processing is possible (such

as the vector space model); however, the use of Bloom filters (described in the next section) makes it difficult to directly implement such complex query processing.

To answer a query with k terms, we first determine which peers store the inverted lists for those terms. This is called the *discovery phase*. The discovery phase requires k lookup messages; although some peers may have multiple terms, we do not know this fact in advance. Each lookup message requires, on average, $O(logN)$ hops. The total bandwidth for each lookup message is:

$$Hops \times (s(keyword) + s(PeerAddress) + s(MessageHeader)) \quad (2)$$

The PeerAddress is needed so the network knows which peer to respond to with the ID of the peers storing inverted lists. The total bandwidth for the discovery phase is the sum of the bandwidth for looking up each term.

After we have found the peers storing the inverted lists, we must retrieve the lists and intersect them. This *intersection phase* represents the major use of bandwidth, since the inverted list for a term is much larger than the term itself. To save bandwidth, the searching peer p_s does not actually itself retrieve whole inverted lists. Instead, the peers holding the inverted lists perform the intersections themselves, and only forward intersected lists, which are smaller than (or at least as small as) the original inverted lists. Fig. 1.a shows an example. In this example, a three term query is submitted to peer p_3. After the discovery phase, p_3 knows that the relevant peers are p_5, p_6 and p_1. Peer p_3 sends the query, and the list of peers, to p_5. Peer p_5 then forwards the inverted list for term t_0 to the next peer, p_6. Peer p_6 computes the intersection between the inverted lists for t_0 and its own t_1, and sends the intersection to peer p_1. Peer p_1 takes the intersection of the list it received from p_6 and the full inverted list for its term t_2. The result of this intersection is the query result, which is then returned to the searching peer p_3.

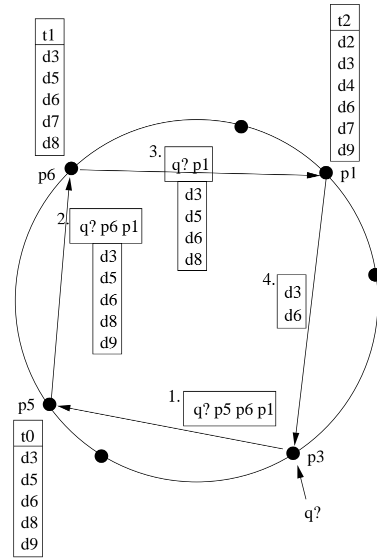
Since we already know which peers hold each inverted list, we only need to forward each intermediate intersected list one hop, directly to the next peer. Thus, the bandwidth required for a single step in the intersection is equal to the bandwidth required to send the resulting intermediate list once, and the total bandwidth for the intersection is the sum of the bandwidths required to forward all of the intermediate lists one hop.

One way to reduce the bandwidth is to sort the k peers based on the size of inverted lists in ascending order, because processing the query beginning with the peer holding the smallest inverted list results in smaller interim intersections. The size of inverted lists can be included in the response messages to p_s in the discovery phase.

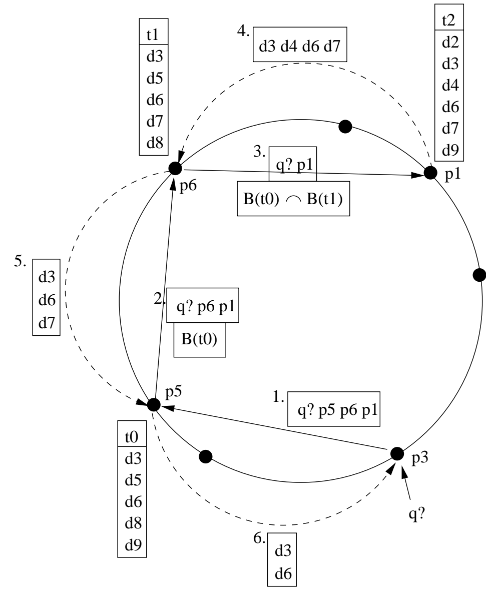
The response time of the search is the sum of the response time for the discovery and intersection phases. First, it takes *logN* hops to find the peer that has the inverted list for each keyword in the query. But we can search the inverted list for all keywords in parallel during the discovery phase, so the total response time is the maximum of the response times for looking up each individual term. The intersection phase proceeds sequentially, with inverted lists being sent from one peer to another. The intersection phase requires $k + 1$ hops, one for each inverted list plus one to return the result to the querying peer. So the overall response time for a query is *O(k+logN)*:

$$MAX(discoveryhops) + k + 1 \quad (3)$$

It is possible to reduce this response time somewhat by doing some intersections in parallel. For example, a four term query can be processed by simultaneously intersecting the lists for two pairs of terms and then intersecting the results. We have not implemented this optimization.



(a) inverted lists



(b) Bloom filters

Fig. 1. Query Processing in the Structured System

TABLE III
PERFORMANCE OF SEARCHING USING INVERTED LISTS

Metric	Value
Average discovery bandwidth (KB)	0.259
Average discovery response time (hops)	5.8
Average intersection bandwidth (KB)	32.0
Average intersection response time (hops)	2.35
Average number of matches for each query	854

Table III shows the measured bandwidth and response time of searching using inverted lists. As the table shows, the cost to intersect inverted lists (32.0 KB) is orders of magnitude higher than the cost of the discovery phase. Several techniques proposed in [7], described in the next sections, can help to reduce the bandwidth for intersecting inverted lists among peers. However, these optimizations do not reduce the cost of the discovery phase.

1) Bloom filters: The communication cost for an intersection grows proportionally with the length of the inverted lists. To reduce the cost, we can use Bloom filters [27] to summarize the inverted lists. A Bloom filter is a hash-based data structure that summarizes membership in a set. The membership test could return false positives with a tunable, predictable probability and never returns false negatives. Given an optimal choice of hash functions, the probability of a false positive is $P_{fp} = 0.6185^{\frac{m}{n}}$, where m is the number of bits in the Bloom filter and n is the number of entries (elements) in the set represented by the filter [28]. Thus, to maintain a fixed probability of false positives, the size of the Bloom filter must be proportional to the number of elements represented. So the performance of Bloom filters and hence the query processing with Bloom filters depends on the "bits per entry": the number of bits used to summarize each entry in the original set.

We can use Bloom filters to summarize inverted lists and thus reduce the bandwidth required to transmit lists in the intersection phase. Because of the false positive nature of Bloom filters, the final intersection result has to be passed back along the peers holding the original inverted lists to eliminate false positives. An example is shown in Fig.1.b. In this figure, $B(t_i)$ indicates a Bloom filter representing the inverted list for term t_i. As the figure shows, Bloom filters and their intersections are forwarded between peers. Peer p_1 converts the result of the final intersection into a list of documents, which is then sent back to peers p_6 and p_5 (dashed arrows) to eliminate false positives before the final result is returned to p_3. If an inverted list-based intersection requires $k+1$ hops, then an equivalent Bloom filter-based intersection phase requires $2(k)$ hops. These extra hops, and the extra bandwidth required to eliminate false positives, somewhat reduce the benefit of Bloom filters. However, in many cases the bandwidth used is still much smaller than sending whole inverted lists; see Section III-B.4 for quantification of when Bloom filters are beneficial.

The bandwidth required when using Bloom filters depends on the size of the sets represented by the Bloom filters, the bits per entry $BPE = m/n$ used to construct the Bloom filter, and the probability of false positives. In the example of Fig.1.b, the size of the Bloom filter for term t_0 is $|list(t_0)| \times BPE$ bits, where $|list(t_0)|$ is the size of the inverted list for term t_0. Including false positives, the expected size of the intersection $B(t_0) \cap B(t_1)$ is

$$\frac{|list(t_0) \cap list(t_1)| \times BPE}{0.6185^{BPE}} \tag{4}$$

bits. The actual bandwidth depends on the actual size of the intersection, and can be measured experimentally.

Despite the reduced bandwidth used by Bloom filters, the cost to transport Bloom filters and query results is usually still significantly larger than the cost of the discovery phase. As in the intersection phase with inverted lists, the intersection phase with Bloom filters does not depend on the number of peers in the system, since each intermediate Bloom filter is sent directly over one hop to the next peer.

2) Caching: Caching can be used for further improvement. Each peer caches the Bloom filters it received during the processing of previous queries. When processing a query, the peer uses cached Bloom filters when possible. Each cached Bloom filter that is used reduces the number of hops needed, with a corresponding decrease in bandwidth used. In our experiments, each peer has an 8 MB cache for Bloom filters, and uses LRU to manage the cache. The improvement from caching depends on the nature of queries and the document set, which determine the likelihood that a previously cached Bloom filter is useful for a new query. Of course, increasing the cache size means that more filters can be cached, resulting in a potentially larger improvement.

3) Experimental Results: Fig.2 shows the performance results using Bloom filters and caching. First, Fig.2.a shows the bandwidth used as a function of the bits per entry (BPE). Note that $BPE = 0$ (on the left of the figure) actually means using inverted lists instead of Bloom filters. Increasing the BPE results in larger Bloom filters and thus more bandwidth to transmit the filters. However, increasing the BPE also reduces false positives. Since false positives need to be transmitted between peers in order to be eliminated, reducing false positives reduces the bandwidth required. As the figure shows, these two effects (increasing Bloom filter size and decreasing false positives) trade off, with the minimum bandwidth achieved at 17 bits per entry. The figure also shows that caching reduces bandwidth by about 5 percent. The real user queries we used did not have enough locality to realize a high cache hit rate and thus a significant improvement from caching.

Fig.2.b shows the response time, again as a function of the BPE. As the figure shows, using Bloom filters (e.g., $BPE > 0$) significantly increases response time, because of the need to reduce false positives. However, as BPE increases, and false positives decrease, there is a greater likelihood that a query that returns no results is detected early, terminating the search. Thus, there is a slight decrease in the number of hops as BPE increases. The figure shows that caching reduces response time by 7 percent. Again, the low cache hit rate means that few of the hops can be avoided, resulting in a small benefit in terms of response time.

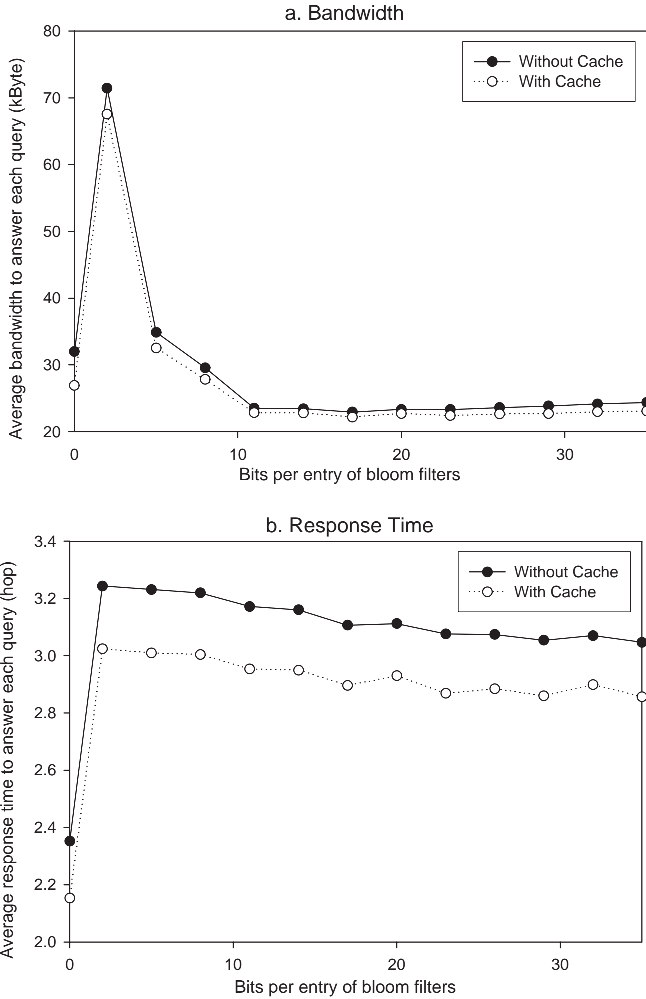


Fig. 2. Performance of answering each query

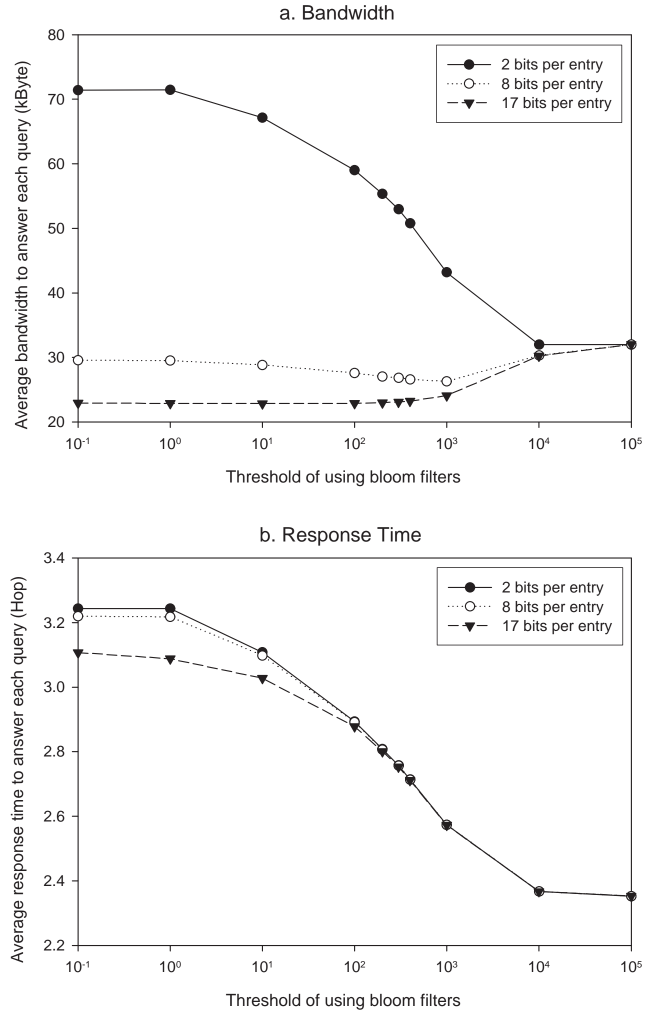


Fig. 3. Threshold of using Bloom filters

4) Threshold of Using Bloom Filters: A Bloom filter may not always be beneficial. If an inverted list is small, and the probability of false positives is high, then the cost of using a Bloom filter may actually exceed that of sending a whole inverted list. This effect was noted in [7], and we noticed it as well in our experiments for some Bloom filter sizes. As suggested in [7], we use a threshold x to determine whether to use a Bloom filter: we only use a Bloom filter if the set to be transmitted has at least x elements. Thus, some queries may be processed using a combination of bloom filters and inverted lists.

Fig.3.a and 3.b show the bandwidth and response time as function of the Bloom filter threshold x. Each graph shows three curves representing different sizes (bits per entry) of Bloom filters. First, for 2 bit-per-entry filters, increasing the threshold decreases the bandwidth used. In other words, the more we use inverted lists instead of Bloom filters, the better we do. We can conclude that 2 bit-per-entry filters do not provide any benefit over inverted lists; the false positive rate is simply too high. In contrast, larger Bloom filters do result in a bandwidth savings. For 8 bit-per-entry Bloom filters, the optimal threshold is approximately 1,000.

An interesting result is that for 17 bit-per-entry Bloom filters the optimal threshold is zero; in other words, such filters are always cheaper in terms of bandwidth than inverted lists. The false positive rate is sufficiently low that even for small sets, the Bloom filter is as small or smaller than the corresponding inverted list. For the rest of the structured network results in this paper, we assume 17 bit-per-entry Bloom filters and a threshold $x = 0$ (that is, Bloom filters are always used.)

No matter what size the Bloom filter is, the response time always decreases when the threshold increases. The final intersection list must always be sent back to the peers holding inverted lists in order to eliminate false positives, even if the false positive rate is low. As the threshold increases, Bloom filters are used less frequently, reducing the number of extra steps needed to eliminate false positives.

IV. Super-peer networks

The super-peer architecture is one of the most popular examples of an unstructured network, having been widely deployed in the FastTrack network used by Kazaa and in

TABLE IV
DOCUMENT PUBLISHING IN SUPER-PEER NETWORK

Average size of index per peer (KB)	345.3
Average response time (hop)	1
Average bandwidth per leaf peer (KB)	345.3
Average number of document per peer	89.76
Average bandwidth per document (KB)	3.85

the "ultra-peer" enhancements to Gnutella. We now examine the performance of the super-peer architecture for full text document search. We also present performance measurements using our Topaz peer-to-peer client (see Section II-B). In our experiments, we used 1,000 total peers, of which 10 percent were designated as super-peers.

A. Document Publishing

Super-peers act as searching servers, and index the documents for their leaf peers. After starting up, every leaf peer sends its local inverted lists to its super-peer. Thus, each publishing message actually contains a set of local inverted lists. This message only needs to travel one hop, from the leaf peer to the super-peer. A super-peer also has documents, and constructs an inverted index. However, unlike leaf peers, super-peers handle searches over their own content, and thus do not need to explicitly "publish" their documents.

Table IV shows the measured costs for publishing documents in our super-peer network.

B. Query Processing

When a query is issued from a leaf peer, it is first sent to the super-peer. The super-peer both processes the query and forwards it to other super-peer neighbors. In this way, the query will be forwarded among all super-peers, bounded by the TTL of the search message. If a query is issued from a super-peer, it is similarly forwarded to other super-peers for processing. Since the network of super-peers likely has cycles, super-peers discard, without processing, search messages they have previously seen (identified by the message ID).

The bandwidth required for processing queries consists of the bandwidth needed to forward search messages and return results. Unlike in the structured network, there is no need to transport inverted lists between peers; every super-peer has complete inverted indexes from a set of leaf peers and can completely evaluate the query for itself and its leaf peers.

The response time of the query depends on the structure of the connections between super-peers. Consider the time required for the search message starting at super-peer p to reach all of the super-peers it will be forwarded to. If the maximally distant super-peer (that is, the super-peer the most hops away from p) is more than TTL hops away, then the response time is bounded by the TTL; otherwise, it is bounded by the number of hops to the maximally distant super-peer. Therefore, the upper bound of the response time is controlled by the user-defined TTL.

In order to enable a fair comparison to the structured and random-walk search techniques, we use a TTL that is large enough for the search to reach all super-peers. This would be the case in a network where the super-peers had the capacity to process all searches; indeed, the primary reason to use a TTL in a super-peer network is to bound the amount of load processed by super-peers. Then, in our experiments, we can calculate the response time for a search as the number of hops from the super-peer where the query started to the maximally distant super-peer.

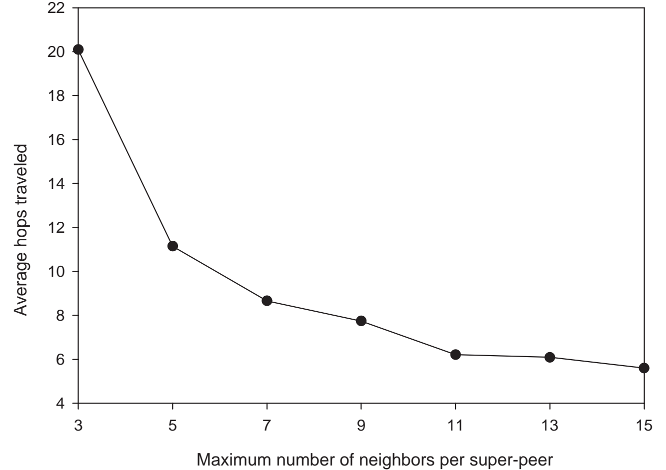


Fig. 4. Average Hops Traveled in the Super-peer Network

Fig.4 shows the average number of hops that search messages had to travel in our experiments. As the figure shows, the distance traveled varied based on the maximum number of neighbors allowed to super-peers, which is a tunable parameter. As the number of neighbors increased, the network became denser and the network paths between super-peers shortened. We could reach the ideal response time of one hop between all super-peers by constructing a fully-connected network. However, fully-connected peer-to-peer networks are not scalable; every peer must know about every other peer and any change (a peer joins or leaves the network) must be handled by every peer. The same would be true for a fully-connected super-peer network. For this reason, real networks typically limit the number of neighbors that super-peers can have. In the following experiments, we set the maximum number of neighbors to 7, since (as the figure shows) there is less improvement in hop count for larger values.

Table V summarizes the performance for a super-peer network. This table shows the bandwidth and response time (hop count) for both query and reply messages, as well as the total bandwidth and response time for the query and reply messages together. The reply message from a super-peer contains the IDs of all the documents that the super-peer indexes which match the query, along with the addresses of peers where those documents are stored. Thus, the reply message is usually much larger than the query message, which only contains the query and the address of the searching peer. The reply message can be forwarded directly (e.g. one hop) to the searching peer to

TABLE V

QUERY PROCESSING IN THE SUPER-PEER SYSTEM

Average bandwidth to forward each query (KB)	2.50
Average time to forward each query (hops)	9.56
Average bandwidth of the reply (KB)	17.59
Average time of the reply (hops)	1
Average overall bandwidth for each query (KB)	20.09
Average overall response time for each query (hops)	10.56
Average number of matches for each query	854

mitigate the bandwidth cost of the large reply message.

V. RANDOM WALK SEARCHES IN UNSTRUCTURED NETWORKS

Because searches often match many documents, it is not always necessary to search every peer's content in order to find enough results. Random walk searches [9], [10] reduce the bandwidth used for searching, compared to Gnutella's original flooding protocol, by contacting a random subset of the peers. In this section, we examine the performance of random walk searches. Our experiments were again conducted using Topaz, configured to support random walks and the optimizations we describe here. Recall that in this technique, publishing is not necessary; each peer stores, indexes and processes searches over its own content.

A. *Query Processing*

Consider an unstructured network, where all peers perform equivalent functions (that is, there are no super-peers). Each peer has some neighbors, and the number of neighbors varies from peer to peer. With a random walk search, each peer that receives a query processes it over its own content, returns any results, and forwards it to n random neighbors. In the simplest case, $n = 1$ and a single search message "walks" around the network. The walk terminates when a user-specified threshold T number of results have been found, a user-specified TTL is reached, or some other stopping condition is satisfied. If the search matches enough documents in the network the walk should terminate long before reaching every peer, reducing the bandwidth requirements compared to network flooding.

Lv et al [10] demonstrate that the efficiency of a random walk can be improved using *state-keeping*: each peer keeps track of who they forward searches to, and avoid forwarding the same search to the same neighbor repeatedly. We use state-keeping in our experiments.

It can be shown that the most efficient unstructured network structure for a random walk, in terms of messages sent, is the *square-root topology* [18], [24]. In this topology, each peer tracks the popularity of its content, and adjusts its degree (number of neighbors) so that the degree is proportional to the square root of the popularity of the content. Experiments show that this topology improves the efficiency of the search by a factor of two or more over random or power-law networks. In our experiments, peers adjust their degree continually to achieve the square root topology. The resulting distribution of degrees in our experimental network of 1,000 peers is shown in Figure 5.

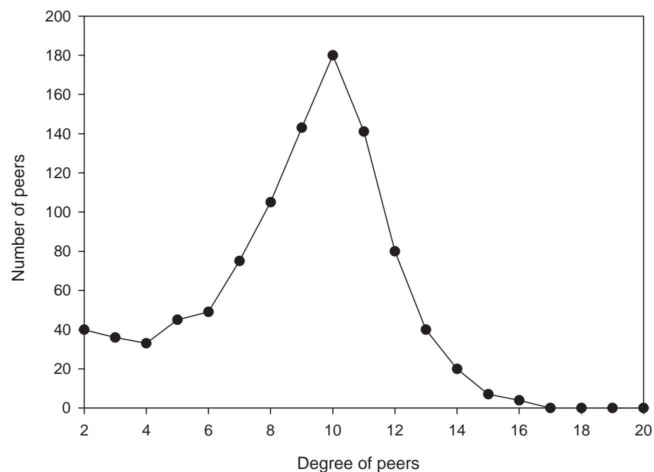


Fig. 5. Node degrees in the square root topology

One disadvantage of random walks is that the search latency can be very high. A search may walk for many hops before finding content, and since the walk proceeds sequentially, this translates into a large search latency. Lv et al [10] propose that multiple random walks for the same query should be sent out in parallel to reduce search latency. Simulations show that this approach reduces latency roughly in proportion to the number of parallel walks, although the total number of search messages sent remains the same as in the sequential case. However, each parallel walk must periodically check with the peer issuing the query to see if the threshold T number of results has been found; when the parallel walks have collectively found enough results all walks terminate. These periodic checks require extra bandwidth beyond that required just to forward the searches. The number of walks is a user-configurable parameter, and the amount of extra bandwidth required for checking the stopping condition depends on how often the checks occur, also a user-defined parameter. In our experiments, we send out a single walk to measure the performance of the basic protocol. Users can achieve their desired search latency, with a corresponding bandwidth increase, by specifying the number of parallel walks and the frequency of checking the stopping condition.

Many other optimizations and variations to random walks have been proposed [11], [12], [13], [17], [25], [29]. Some of these extensions reduce overall bandwidth usage or response time, while others focus on load balancing in the network. We have not experimented with all of the proposed optimizations and variations. However, we note that the performance results reported here can potentially be improved through some or all of these techniques.

In our experiments, we found that the choice of T made a large difference in performance. If we set T to be small, many searches terminated quickly, resulting in efficient performance. Unfortunately, a small T means that only a few results are found for all searches, reducing the amount of information returned to the user for searches that would otherwise match

TABLE VI
QUERY PROCESSING WITH RANDOM WALKS

Average bandwidth to forward each query (KB)	9.29
Average time to forward each query (hops)	440.54
Average bandwidth of the reply (KB)	14.12
Average time of the reply (hops)	1
Average overall bandwidth for each query (KB)	23.41
Average overall response time for each query (hops)	441.54
Average number of matches for each query	672

many documents. On the other hand, if we set T to be large, searches walked for a comparatively longer time, and searches that matched a total of less than T documents walked for the full TTL. This means both a long response time for the user, and an unnecessarily large amount of bandwidth used for searches that would never find T results.

As a compromise, we used the following stopping condition: if a random walk search walked for H hops without finding a match, the walk terminated. In our experiments, $H = 64$. Thus, queries that match many documents continued walking for a long time, as long as they continued to find matches. Queries that matched few documents terminated rapidly, as they quickly encountered a period of H hops without matching anything. The result was reasonable performance, without having to specify T directly.

Table VI shows the performance of random walk searches in a square-root network with statekeeping and our adaptive stopping condition. As the table shows, searches found only 672 results on average, compared to 854 results for super-peers and the structured network. The table also shows that queries had to be sent to many peers; in a network with 1,000 peers, the average walk had to travel 440 hops.

VI. PERFORMANCE COMPARISON

Now that we have examined each technique in detail, we can directly compare the performance of all three techniques.

A. Document Publishing

Fig.6.a shows the bandwidth required for document publishing. As the figure shows, document publishing in the super-peer system requires 83 percent less bandwidth than in the structured system. The random walk technique does not require explicit publishing, so the cost is zero. In the super-peer system, a leaf peer publishes its documents by sending a single message with a whole inverted index to a super-peer. In contrast, in the structured system, publishing requires sending multiple messages, one per published term per peer. Sending multiple messages means sending the peer ID many times, as well as sending multiple message headers, resulting in the higher cost of publishing in the structured system.

Fig.6.b shows the response time of the publishing phase. As the figure shows, the super-peer system requires 62 percent fewer hops than the structured system (while the random walk system again requires zero response time since no publishing is done). In the super-peer network, each publish message travels one hop from the leaf peer to the super-peer, while in the structured system messages travel $O(logN)$ hops. Thus, even though publish messages can be sent in parallel in the structured system, the response time is higher than in the super-peer system.

Both the bandwidth and response time in the super-peer system are independent of the number of peers in the system. So the experimental results for the super-peer system are not likely to be much different when the number of peers varies. On the other hand, the bandwidth and response time in the structured system will increase somewhat with increasing system size, since they are proportional to the logarithm of the number of peers. In contrast, with a larger document set, the bandwidth (but not the response time) of publishing in both the structured and super-peer system will increase in proportion to the size of the document set.

B. Query Processing

Query processing consists of two steps:

1) *Forwarding the query*. This means:
   - In the structured system, finding all of the peers holding relevant inverted lists
   - In the super-peer system, forwarding the query to all of the super-peers
   - In the random walk system, forwarding the query among peers
2) *Answering the query*. This means:
   - In the structured system, forwarding the Bloom filters between peers, checking for false positives, and returning the list of matches
   - In the super-peer system, returning the list of matches
   - In the random walk system, returning the list of matches

Fig.7.a shows the total bandwidth for forwarding and answering queries, broken down into these two phases. As the figure shows, all three techniques perform roughly the same, with the super-peer technique requiring the least bandwidth (14 percent less than the random walk technique and 11 percent less than the structured technique). In all three techniques, the cost is dominated by the *query answering phase*. For the unstructured networks (super-peer and random walks), the cost of transporting hundreds of query results is more expensive than forwarding the query. This is especially true in the super-peer network, where the query is only forwarded to a few super-peers. In the structured network, query answering is extremely expensive because of the need to forward Bloom filter intersections and check for false positives. The cost of finding peers holding relevant inverted lists (*query forwarding*) is comparatively very small.

Fig.7.b shows the response time. Note that the figure has a logarithmic vertical axis. The structured system provides the fastest response time, requiring 30 percent fewer hops in total than the super-peer system. The logarithmic properties of the Chord DHT ensure a fast response time, while the super-peer network has comparatively longer paths between

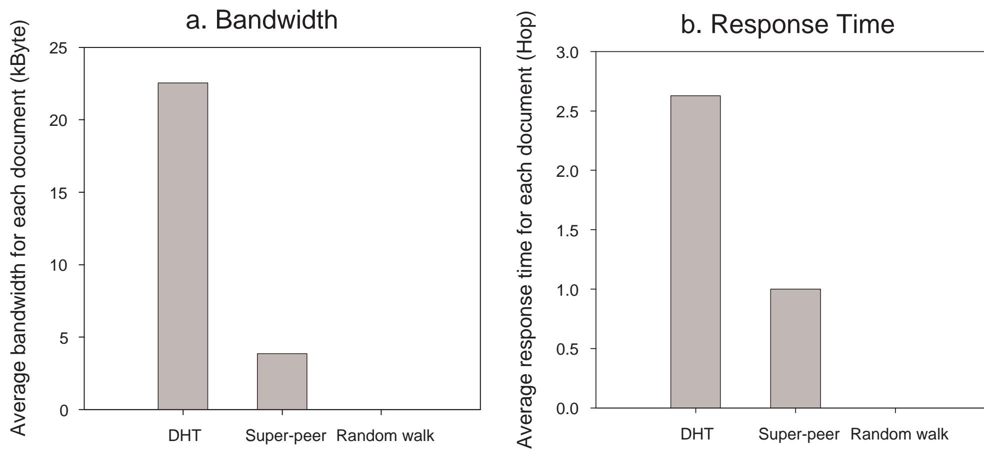


Fig. 6. Comparison of Document Publishing

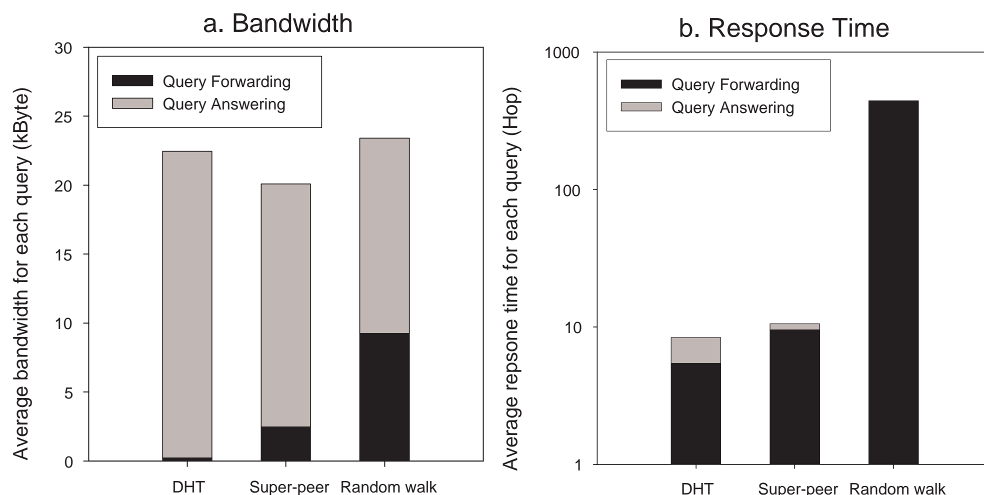


Fig. 7. Comparison of Query Processing

super-peers. The random walk requires a very large response time because of its sequential nature. However, as we noted in Section V, the response time of the random walk technique can be tuned by the user by sending out multiple random walks. Thus, response time comparable to the other techniques can be achieved for random walks by choosing the appropriate number of parallel walks, with a corresponding increase in bandwidth (as discussed above).

As the number of peers increases, the response time and bandwidth will increase in all three types of systems. For response time, the primary component is the query forwarding, which will increase logarithmically for the structured system and linearly for the super-peer and random walk networks. Thus, for very large networks, the response time is likely to be much larger in the super-peer system (and continue to be significantly large in the random walk system). Bandwidth cost is likely to be less affected by increasing system size in the structured and super-peer networks, since the primary component is query answering, which is a function of the query results and not the system size. The random walk technique is more affected by query forwarding cost; in large networks query forwarding is likely to be the dominant cost component, causing the bandwidth used to increase proportionally to the number of peers. In contrast, in all three systems, a larger document set will increase the bandwidth usage (but not the response time) for query answering, since the number of query results are likely to increase.

VII. Related Work

Although we study the performance of full text search using Chord [1] other DHTs such as CAN [2], Pastry [3], Tapestry [5], and so on can also be used. Since most DHTs require $O(logN)$ hops to route messages, we expect results to be similar in different DHTs. CAN's performance is related to the dimensionality of the CAN space, and is not necessarily $O(logN)$. Further experiments are needed to examine the efficiency of full text keyword search in CAN and DHTs routing behavior other than $O(logN)$.

Researchers proposing various searching techniques have conducted extensive simulations and experiments to quantify their performance [6], [7], [8], [10], [11], [16], [17]. In many cases, the performance studies focus on one kind of architecture; for example, comparing structured to structured or unstructured to unstructured. Zhong et al [19] conducted a performance evaluation of peer-to-peer full text keyword search techniques under different organizations of content indexes. Our focus is on the performance difference of unstructured and structured networks. In particular, we study super-peer and random walk unstructured techniques, which are not considered in [19]. Other than [19], we are not aware of any direct comparison of structured and unstructured approaches for full text search.

Various performance studies of deployed peer-to-peer systems have been conducted [30], [31], [32], [33]. These studies focus on understanding the performance of a single system, rather than comparing multiple architectures. Several analytical models of peer-to-peer systems have been developed (such as [34]). These models can be used to complement and understand the results of experimental measurements of peer-to-peer systems.

VIII. Conclusion and Future Work

We have provided a quantitative evaluation and direct comparison of structured and unstructured P2P systems for full text search. Using real web documents and user queries, we have examined the performance of the publishing and searching phases of three different techniques. Our results show that all three techniques use roughly the same bandwidth to process queries (with the super-peer technique having a slight edge). The structured network provides the best response time (30 percent better than a super-peer network), but has a high cost of document publishing, using six times as much bandwidth as the super-peer system. The random walk technique requires no publishing, but has a very long response time unless multiple random walks operate in parallel.

Peer-to-peer systems represent a huge design space, and we have only compared three techniques in that space. Other techniques may potentially improve the performance of full text search beyond the results we present here. Moreover, other types of documents or queries may result in different performance. However, our results demonstrate several of the inherent strengths and weaknesses of different peer-to-peer architectures for full text search.

References

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

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

[3] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *Proc. IFIP/ACM Int'l Conf. on Distributed Systems Platforms (Middleware)*, 2001.

[4] M. Kaasoek and D. Karger, "Koorde: A simple degree-optimal hash table," in *Proc. Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[5] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE JSAC*, vol. 22, no. 1, pp. 41–53, Jan. 2004.

[6] J. Li, B. T. Loo, J. Hellerstein, M. F. Kaashoek, D. R. Karger, and R. Morris, "On the feasibility of peer-to-peer web indexing and search," in *Proc. Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[7] P. Reynolds and A. Vahdat, "Efficient peer-to-peer keyword searching," in *Proc. ACM/IFIP/USENIX Int'l Middleware Conf.*, 2003.

[8] Y.-J. Joung, C.-T. Fang, and L.-W. Yang, "Keyword search in DHT-based peer-to-peer networks," in *Proc. Int'l Conf. on Distributed Computing Systems (ICDCS)*, 2005.

[9] L. Adamic, R. Lukose, A. Puniyani, and B. Huberman, "Search in power-law networks," *Phys. Rev. E*, vol. 64, pp. 46 135–46 143, 2001.

[10] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and replication in unstructured peer-to-peer networks," in *Proc. of ACM Int'l Conf. on Supercomputing (ICS'02)*, June 2002.

[11] B. Yang and H. Garcia-Molina, "Efficient search in peer-to-peer networks," in *Proc. Int'l Conf. on Distributed Computing Systems (ICDCS)*, 2002.

[12] A. Crespo and H. Garcia-Molina, "Routing indices for peer-to-peer systems," in *Proc. Int'l Conf. on Distributed Computing Systems (ICDCS)*, 2002.

[13] V. Kalogeraki, D. Gunopulos, and D. Zeinalipour-Yazti, "A local search mechanism for peer-to-peer networks," in *Proc. CIKM*, 2002.

[14] A. Kumar, J. Xu, and E. Zegura, "Efficient and scalable query routing for unstructured peer-to-peer networks," in *Proc. INFOCOM*, 2005.

[15] A. Carzaniga and A. L. Wolf, "Forwarding in a content-based network," in *Proc. SIGCOMM*, 2003.

[16] B. Yang and H. Garcia-Molina, "Designing a super-peer network," in *Proc. IEEE ICDE*, 2003.

[17] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, "Making Gnutella-like P2P systems scalable," in *Proc. SIGCOMM*, 2003.

[18] B. F. Cooper, "An optimal overlay topology for routing peer-to-peer searches," in *Proc. ACM/IFIP/USENIX Int'l Middleware Conf.*, 2005.

[19] M. Zhong, J. Moore, K. Shen, and A. L. Murphy, "An evaluation and comparison of current peer-to-peer full-text keyword search techniques," in *Proc. of the Int'l Workshop on the Web Databases (WebDB)*, 2005.

[20] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. New York, N.Y.: ACM Press, 1999.

[21] L. Gravano, H. Garcia-Molina, and A. Tomasic, "GlOSS: Text-source discovery over the internet," *ACM TODS*, vol. 24, no. 2, pp. 229–264, June 1999.

[22] L. Page and S. Brin, "The anatomy of a large-scale hypertext web search engine," in *Proc. WWW*, 1998.

[23] C. Gkantsidis, M. Mihail, and A. Saberi, "Random walks in peer-to-peer networks," in *Proc. INFOCOM*, 2004.

[24] B. F. Cooper, "Quickly routing searches without having to move content," in *Proc. Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.

[25] E. Cohen and S. Shenker, "Replication strategies in unstructured peer-to-peer networks," in *Proc. SIGCOMM*, 2002.

[26] "Chord," http://pdos.csail.mit.edu/chord/.

[27] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[28] L. Fan, P. Cao, J. Almeida, and A. Broder., "Summary cache: A scalable wide-area web cache sharing protocol," in *Proc. ACM SIGCOMM*, 1998.

[29] Q. Lv, S. Ratnasamy, and S. Shenker, "Can heterogeneity make gnutella scalable?" in *Proc. Int'l Workshop on Peer to Peer Systems (IPTPS)*, 2002.

[30] F. L. Fessant, S. Handurukande, A.-M. Kermarrec, and L. Massoulie, "Clustering in peer-to-peer file sharing workloads," in *Proc. Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[31] M. Ripeanu and I. Foster, "Mapping the gnutella network: Macroscopic properties of large-scale peer-to-peer systems," in *Proc. Int'l Workshop on Peer to Peer Systems (IPTPS)*, 2002.

[32] S. Saroiu, K. Gummadi, and S. Gribble, "A measurement study of peer-to-peer file sharing systems," in *Proc. Multimedia Conferencing and Networking*, 2002.

[33] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan, "Measurement, modeling and analysis of a peer-to-peer file-sharing workload," in *Proc. SOSP*, 2003.

[34] Z. Ge, D. Figueiredo, S. Jaiswal, J. Kurose, and D. Towsley, "Modeling peer-peer file sharing systems," in *Proc. INFOCOM*, 2003.