Efficient Peer-to-Peer Keyword Searching

Patrick Reynolds and Amin Vahdat

Department of Computer Science, Duke University
{*reynolds,vahdat*}@*cs.duke.edu* ⋆

Abstract. The recent file storage applications built on top of peer-to-peer distributed hash tables lack search capabilities. We believe that search is an important part of any document publication system. To that end, we have designed and analyzed a distributed search engine based on a distributed hash table. Our simulation results predict that our search engine can answer an average query in under one second, using under one kilobyte of bandwidth.
Keywords: search, distributed hash table, peer-to-peer, Bloom filter, caching

1 Introduction

Recent work on distributed hash tables (DHTs) such as Chord [19], CAN [16], and Pastry [17] has addressed some of the scalability and reliability problems that plagued earlier peer-to-peer overlay networks such as Napster [14] and Gnutella [8]. However, the useful keyword searching present in Napster and Gnutella is absent in the DHTs that endeavor to replace them. In this paper, we present a symmetrically distributed peer-to-peer search engine based on a DHT and intended to serve DHT-based file storage systems.

Applications built using the current generation of DHTs request documents using an opaque key. The means for choosing the key is left for the application built on top of the DHT to determine. For example, the Chord File System, CFS [6], uses hashes of content blocks as keys. Freenet [5,9], which shares some characteristics of DHTs, uses hashes of filenames as keys. In each case, users must have a single, unique name to retrieve content. No functionality is provided for keyword searches.

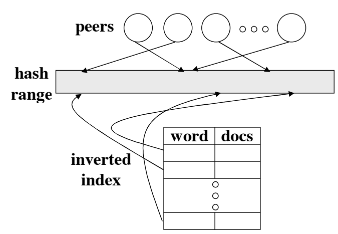


Fig. 1. Distributing an inverted index across a peer-to-peer network.

The system described in this paper provides keyword search functionality for a DHT-based file system or archival storage system, to map keyword queries to the unique routing keys described above. It does so by mapping each keyword to a node in the DHT that will store a list of documents containing that keyword. Figure 1 shows how keywords in the index map into the hash range and, in turn, to nodes in the DHT.

⋆ This research is supported in part by the National Science Foundation (EIA-99772879, ITR-0082912), Hewlett Packard, IBM, Intel, and Microsoft. Vahdat is also supported by an NSF CAREER award (CCR-9984328), and Reynolds is also supported by an NSF fellowship.

We believe that end-user latency is the most important performance metric for a search engine. Most end-user latency in a distributed search engine comes from network transfer times. Thus, minimizing the number of bytes sent and the number of times they are sent is crucial. Both bytes and hops are easy to minimize for queries that can be answered by a single host. Most queries, however, contain several keywords and must be answered by several cooperating hosts. Using a trace of 99,405 queries sent through the IRCache proxy system to Web search engines during a ten-day period in January 2002, we determined that 71.5% of queries contain two or more keywords. The entire distribution of keywords per query is shown in Figure 2. Because multiple-keyword queries dominate the search workload, optimizing them is important for end-user performance. This paper focuses on minimizing network traffic for multiple-keyword queries.

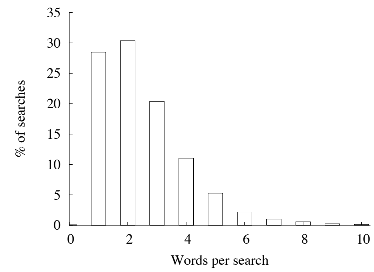


Fig. 2. Number of keywords per search operation in the IRCache for a ten-day period in January 2002.

1.1 Non-goals

One extremely useful feature of distributed hash tables is that they provide a simple service model that hides request routing, churn costs, load balancing, and unavailability. Most DHTs route requests to nodes that can serve them in expected $O(\lg n)$ steps, for networks of n hosts. They keep churn costs [11] – the costs associated with managing node joins and departures – logarithmic with the size of the network. Using consistent hashing [10] they divide load roughly evenly among available hosts. Finally, they perform replication to ensure availability even when individual nodes fail. Our design uses a DHT as its base; thus, it does not directly address these issues.

1.2 Overview

This paper describes our search model, design, and simulation experiments as follows. In Section 2 we describe several aspects of the peer-to-peer search problem space, along with the parts of the problem space we chose to explore. Section 3 describes our approach to performing peer-to-peer searches efficiently. Section 4 details our simulation environment, and Section 5 describes the simulation results. We present related work in Section 6 and conclude in Section 7.

2 System Model

Fundamentally, search is the task of associating keywords with document identifiers and later retrieving document identifiers that match combinations of keywords. Most text searching systems use inverted indices, which map each word found in any document to a list of the documents in which the word appears. Beyond this simple description,

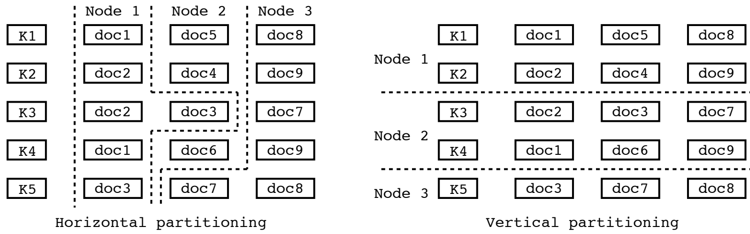


Fig. 3. A horizontally partitioned index stores part of every keyword match-list on each node, often divided by document identifiers. Here we divide the index into document identifiers 1-3, 4-6, and 7-9. A vertically partitioned index assigns each keyword to a single node.

many design trade-offs exist. How will the index be partitioned, if at all? Should it be distributed, or would a centralized index suffice? In what order will matching documents be listed? How are document changes reflected in the index? We address these questions below.

2.1 Partitioning

Although a sufficiently small index need not be partitioned at all, our target application is a data set large enough to overwhelm the storage and processing capacities of any single node. Thus, some partitioning scheme is required. There are two straightforward partitioning schemes: horizontal and vertical.

For each keyword an index stores, it must store a match-list of identifiers for all of the documents containing the keyword. A horizontally partitioned index divides this list among several nodes, either sequentially or by partitioning the document identifier space. Google [3] operates in this manner. A vertically partitioned index assigns each keyword, undivided, to a single node. Figure 3 shows a small sample index partitioned horizontally and vertically, with *K1* through *K5* representing keywords and *doc1* through *doc9* representing documents that contain those keywords.

A vertically partitioned index minimizes the cost of searches by ensuring that no more than k servers must participate in answering a query containing k keywords. A horizontally partitioned index requires that all nodes be contacted, regardless of the number of keywords in the query. However, horizontal indices partitioned by document identifier can insert or update a document at a single node, while vertically partitioned indices require that up to k servers participate to insert or update a document with k keywords. As long as more servers participate in the overlay than there are keywords associated with an average document, these costs favor vertical partitioning. Furthermore, in file systems, most files change rarely, and those that change often change in bursts and may be removed shortly after creation, allowing us to optimize updates by propagating changes lazily. In archival storage systems, files change rarely if at all. Thus, we believe that queries will outnumber updates for our proposed uses, further increasing the cost advantage for vertically partitioned systems.

Vertically partitioned indices send queries to a constant number of hosts, while horizontally partitioned indices must broadcast queries to all nodes. Thus, the throughput of a vertically partitioned index theoretically grows linearly as more nodes are added.

Query throughput in a horizontally partitioned index does not benefit at all from additional nodes. Thus, we chose vertical partitioning for our search engine.

2.2 Centralized or Distributed Organization

Google has had great success providing centralized search services for the Web. However, we believe that for peer-to-peer file systems and archival storage networks, a distributed search service is better than a centralized one. First, centralized systems provide a single point of failure. Failures may be network outages; denial-of-service attacks, as plagued several Web sites in February of 2000; or censorship by domestic or foreign authorities. In all such cases, a replicated distributed system may be more robust. Second, many uses of peer-to-peer distributed systems depend on users voluntarily contributing computing resources. A centralized search engine would concentrate both load and trust on a small number of hosts, which is impractical if those hosts are voluntarily contributed by end users.

Both centralized and distributed search systems benefit from replication. Replication improves availability and throughput in exchange for additional hardware and update costs. A distributed search engine benefits more from replication, however, because replicas are less susceptible to correlated failures such as attacks or network outages. Distributed replicas may also allow nodes closer to each other or to the client to respond to queries, reducing latency and network traffic.

2.3 Ranking of Results

One important feature of search engines is the order in which results are presented to the user. Many documents may match a given set of keywords, but some may be more useful to the end user than others. Google's PageRank algorithm [15] has successfully exploited the hyperlinked nature of the Web to give high scores to pages linked to by other pages with high scores. Several search engines have successfully used words' proximity to each other or to the beginning of the page to rank results. Peer-to-peer systems lack the linking structure necessary for PageRank but may be able to take advantage of word position or proximity heuristics. We will discuss specific interactions between ranking techniques and our design in Section 3.5 after we have presented the design.

2.4 Update Discovery

A search engine must discover new, removed, or modified documents. Web search engines have traditionally relied on enumerating the entire Web using crawlers, which results in either lag or inefficiency if the frequency of crawling differs from the frequency of updates for a given page. Popular file-sharing systems use a "push" model for updates instead: clients that have new or modified content notify servers directly. Even with pushed updates, the process of determining keywords and reporting them to server should occur automatically to ensure uniformity.

The Web could support either crawled or pushed updates. Crawled updates are currently the norm. Peer-to-peer services may lack hyperlinks or any other mechanism for enumeration, leaving them dependent on pushed updates. We believe that pushed updates are superior because they promote both efficiency and currency of index information.

2.5 Placement

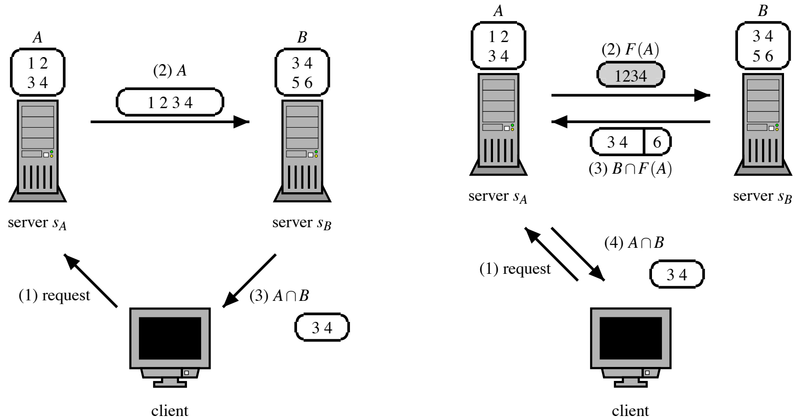
All storage systems need techniques for placing and finding content. Distributed search systems additionally need techniques for placing index partitions. We use a DHT to map keywords to nodes for the index, and we claim that the placement of content is an orthogonal problem. There is little or no benefit to placing documents and their keywords in the same place. First, very few documents indicated as results for a search are later retrieved; thus, most locality would be wasted. Second, there is no overlap between an index entry and the document it indicates; both still must be retrieved and sent over the network. A search engine is a layer of indirection. It is expected that documents and their keywords may appear in unrelated locations.

3 Efficient Support for Peer-to-Peer Search

In the previous section, we discussed the architecture and potential benefits of a fully distributed peer-to-peer search infrastructure. The primary contribution of this work is to demonstrate the feasibility of this approach with respect to individual end user requests. Conducting a search for a single keyword consists of looking up the keyword's mapping in the index to reveal all of the documents containing that keyword. This involves contacting a single remote server, an operation with network costs comparable to accessing a traditional search service. A boolean "AND" search consists of looking up the sets for each keyword and returning the intersection. As with traditional search engines, we return a small subset of the matching documents. This operation requires contacting multiple peers across the wide area, and the requisite intersection operation across the sets returned by each peer can become prohibitively expensive, both in terms of consumed network bandwidth and the latency incurred from transmitting this data across the wide area.

Consider the example in Figure 4(a), which shows a simple network with servers s_A and s_B. Server s_A contains the set of documents A for a given keyword k_A, and server s_B contains the set of documents B for another keyword k_B. $|A|$ and $|B|$ are the number of documents containing k_A and k_B, respectively. $A \cap B$ is the set of all documents containing both k_A and k_B.

The primary challenge in performing efficient keyword searches in a distributed inverted index is limiting the amount of bandwidth used for multiple-keyword searches. The naive approach, shown in Figure 4(a), consists of the first server, s_A, sending its entire set of matching document IDs, A, to the second server, s_B, so that s_B can calculate $A \cap B$ and send the results to the client. This is wasteful because the intersection, $A \cap B$, is likely to be far smaller than A, resulting in most of the information in A getting discarded at s_B. Furthermore, the size of A (i.e., the number of occurrences of the keyword k_A) scales roughly with the number of documents in the system. Thus, the cost of naive search operations grows linearly with the number of documents in the system. We propose three techniques to limit wasted bandwidth, to ensure scalability, and to reduce end-client latency: Bloom filters, caches, and incremental results. We discuss each of these approaches in turn and present analytical results showing the potential benefits of each technique under a variety of conditions before exploring these tradeoffs in more detail through simulation in Section 5.



(a) A simple approach to "AND" queries. Each server stores a list of document IDs corresponding to one keyword.

(b) Bloom filters help reduce the bandwidth requirement of "AND" queries. The gray box represents the Bloom filter $F(A)$ of the set A. Note the false positive in the set $B \cap F(A)$ that server s_B sends back to server s_A.

Fig. 4. Network architecture and protocol overview

3.1 Bloom filters

A Bloom filter [2, 7, 13] is a hash-based data structure that summarizes membership in a set. By sending a Bloom filter based on A instead of sending A itself, we reduce the amount of communication required for s_B to determine $A \cap B$. The membership test returns false positives with a tunable, predictable probability and never returns false negatives. Thus, the intersection calculated by s_B will contain all of the true intersection, as well as a few hits that contain only k_B and not k_A. The number of false positives falls exponentially as the size of the Bloom filter increases.

Given optimal choice of hash functions, the probability of a false positive is

$$p_{fp} = .6185^{m/n}, \tag{1}$$

where m is the number of bits in the Bloom filter and n is the number of elements in the set [7]. Thus, to maintain a fixed probability of false positives, the size of the Bloom filter must be proportional to the number of elements represented.

Our method for using Bloom filters to determine remote set intersections is shown in Figure 4(b) and proceeds as follows. A and B are the document sets to intersect, each containing a large number of document IDs for the keywords k_A and k_B, respectively. The client wishes to retrieve the intersection $A \cap B$. Server s_A sends a Bloom filter $F(A)$ of set A to server s_B. Server s_B tests each member of set B for membership in $F(A)$. Server s_B sends the matching elements, $B \cap F(A)$, back to server s_A, along with some textual context for each match. Server s_A removes the false positives from s_B's results by calculating $A \cap (B \cap F(A))$, which is equivalent to $A \cap B$.

False positives in $B \cap F(A)$ do not affect the correctness of the final intersection but do waste bandwidth. They are eliminated in the final step, when s_A intersects $B \cap F(A)$ against A.

It is also possible to send $B \cap F(A)$ directly from s_B to the client rather than first sending it to s_A and removing the false positives. Doing so eliminates the smaller transfer and its associated latency at the expense of correctness. Given reasonable values for $|A|$, $|B|$, the size of each document record, and the cache hit rate (see Section 3.2), the false-positive rate may be as high as 0.05 or as low as 0.00003. This means that $B \cap F(A)$ will have from $0.00003|B|$ to $0.05|B|$ extra elements that do not contain k_A. For example, if 5% of the elements of B actually contain k_A, then returning the rough intersection $B \cap F(A)$ to the client results in between $\frac{0.00003|B|}{(0.05+0.00003)|B|} = 0.06\%$ and $\frac{0.05|B|}{(0.05+0.05)|B|} = 50\%$ of the results being incorrect and not actually containing k_A, where each expression represents the ratio of the number of false positives to the total number of elements in $B \cap F(A)$. The decision to use this optimization is made at run time, when the parameters are known and p_{fp} can be predicted. Server s_A may choose an m value slightly larger than optimal to reduce p_{fp} and improve the likelihood that s_B can return $B \cap F(A)$ directly to the client.

The total number of bits sent during the exchange shown in Figure 4(b) is $m + p_{fp}|B|j + |A \cap B|j$, where j is the number of bits in each document identifier. For this paper, we assume that document identifiers are 128-bit hashes of document contents; thus, j is 128. The final term, $|A \cap B|j$, is the size of the intersection itself. It can be ignored in our optimization, because it represents the resulting intersection, which must be sent regardless of our choice of algorithm.

The resulting total number of excess bits sent (i.e., excluding the intersection itself) is

$$m + p_{fp}|B|j.$$

Substituting for p_{fp} from Equation 1 yields the total number of excess bits as

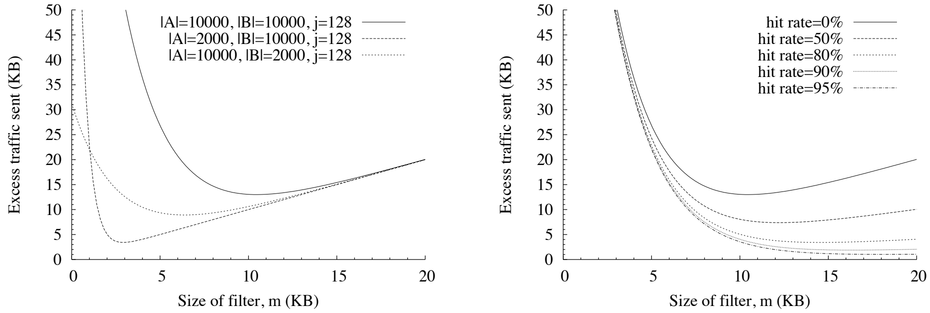
$$m + .6185^{m/|A|}|B|j. \tag{2}$$

Taking the first derivative with respect to m and solving for zero yields an optimal Bloom filter size of

$$m = |A| \log_{.6185}\left(2.081\frac{|A|}{|B|j}\right). \tag{3}$$

Figure 5(a) shows the minimum number of excess bits sent for three sets of values for $|A|$, $|B|$, and j. The optimal m for any given $|A|$, $|B|$, and j is unique and directly determines the minimum number of excess bits sent. For example, when $|A|$ and $|B|$ are $10,000$ and j is 128, m is $85,734$, and the minimum number of excess bits sent is $106,544$, representing $12.01:1$ compression when compared to the cost of sending all $1,280,000$ bits ($10,000$ documents, each with a 128-bit ID) of either A or B.

As also shown in Figure 5(a), performance is not symmetric when A and B differ in size. With j constant at 128, the minimum number of excess bits for $|A| = 2,000$ and $|B| = 10,000$ is $28,008$, lower than the minimum number for $|A| = 10,000$ and $|B| = 2,000$, which is $73,046$. $28,008$ bits represents $9.14:1$ compression when compared



(a) Expected excess bits sent as a function of m

(b) Improving cache hit rates reduces the amount of data sent and increases the size of the optimal Bloom filter.

Fig. 5. Effects of Bloom filter size and cache hit rate

with the $256,000$ bits needed to send all of A. The server with the smaller set should always initiate the transfer.

Our Bloom filter intersection technique can be expanded to arbitrary numbers of keywords. Server s_A sends $F(A)$ to server s_B, which sends $F(B \cap F(A))$ to s_C, and so on. The final server, s_Z, sends its intersection back to s_A. Each server that encoded its transmission using a Bloom filter must process the intersection once more to remove any false positives introduced by its filter. Thus, the intersection is sent to each server except s_Z a second time. As above, the expected number of excess bits is minimized when $|A| \leq |B| \leq |C| \leq \ldots \leq |Z|$.

3.2 Caches

Caching can eliminate the need for s_A to send A or $F(A)$ if server s_B already has A or $F(A)$ stored locally. We derive more benefit from caching Bloom filters than from caching entire document match lists because the smaller size of the Bloom representation means that a cache of fixed size can store data for more keywords. The benefit of caching depends on the presence of locality in the list of words searched for by a user population at any given time. To quantify this intuition, we use the same ten-day IRCache trace described in Section 1 to determine word search popularity. There were a total of 251,768 words searched for across the 99,405 searches, 45,344 of them unique. Keyword popularity roughly followed a Zipf distribution, with the most common keyword searched for 4,365 times. The dominance of popular keywords suggests that even a small cache of either the Bloom filter or the actual document list on A is likely to produce high hit rates.

When server s_B already has the Bloom filter $F(A)$ in its cache, a search operation for the keywords k_A and k_B may skip the first step, in which server s_A sends its Bloom filter to s_B. On average, a Bloom filter will be in another server's cache with probability r equal to the cache hit rate.

The excess bits formula in Equation (2) can be adapted to consider cache hit rate, r, as follows:

$$(1-r)m + .6185^{m/|A|}|B|j \tag{4}$$

Setting the derivative of this with respect to m to zero yields the optimal m as

$$m = |A| \log_{.6185}\left[(1-r)2.081\frac{|A|}{|B|j}\right]. \tag{5}$$

Figure 5(b) shows the effect of cache hit rates on the excess bits curves, assuming $|A|$ and $|B|$ are both 10,000 and j is 128. Each curve still has a unique minimum. For example, when the hit rate, r, is 0.5, the minimum excess number of bits sent is 60,486, representing 21.16 : 1 compression when compared with sending A or B. Improvements in the cache hit rate always reduce the minimum expected number of excess bits and increase the optimal m. The reduction in the expected number of excess bits sent is nearly linear with improvements in the hit rate. The optimal m increases because as we become less likely to send the Bloom filter, we can increase its size slightly to reduce the false-positive rate. Even with these increases in m, we can store hundreds of cache entries per megabyte of available local storage. We expect such caching to yield high hit rates given even moderate locality in the request stream.

Cache consistency is handled with a simple time-to-live field. Updates only occur at a keyword's primary location, and slightly stale match list information is acceptable, especially given the current state of Internet search services, where some degree of staleness is unavoidable. Thus, more complex consistency protocols should not be necessary.

3.3 Incremental results

Clients rarely need all of the results of a keyword search. By using streaming transfers and returning only the desired number of results, we can greatly reduce the amount of information that needs to be sent. This is, in fact, critical for scalability: the number of results for any given query is roughly proportional to the number of documents in the network. Thus, the bandwidth cost of returning all results to the client will grow linearly with the size of the network. Bloom filters and caches can yield a substantial constant-factor improvement, but neither technique eliminates the linear growth in cost. Truncating the results is the only way to achieve constant cost independent of the number of documents in the network.

When a client searches for a fixed number of results, servers s_A and s_B communicate incrementally until that number is reached. Server s_A sends its Bloom filter in chunks and server s_B sends a block of results (true intersections and false positives) for each chunk until server s_A has enough results to return to the client. Because a single Bloom filter cannot be divided and still retain any meaning, we divide the set A into chunks and send a full Bloom filter of each chunk. The chunk size can be set adaptively based on how many elements of A are likely to be needed to produce the desired number of results. This protocol is shown in Figure 6. Note that s_A and s_B overlap their communication: s_A sends $F(A_2)$ as s_B sends $B \cap F(A_1)$. This protocol can be extended logically to more than two participants. Chunks are streamed in parallel from server s_A to s_B, from s_B to s_C, and so on. The protocol is an incremental version of the multi-server protocol described at the end of Section 3.1.

When the system streams data in chunks, caches can store several fractional Bloom filters for each keyword rather than storing the entire Bloom filter for each keyword. This allows servers to retain or discard partial entries in the cache. A server may get a partial cache hit for a given keyword if it needs several chunks but already has some of them stored locally. Storing only a fraction of each keyword's Bloom filter also reduces the amount of space in the cache that each keyword consumes, which increases the expected hit rate.

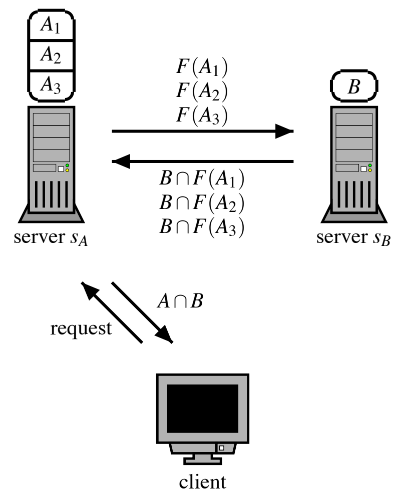


Fig. 6. Servers s_A and s_B send their data one chunk at a time until the desired intersection size is reached.

Sending Bloom filters incrementally substantially increases the CPU costs involved in processing a search. The cost for server s_B to calculate each intersection $B \cap F(A_i)$ is the same as the cost to calculate the entire intersection $B \cap F(A)$ at once because each element of B must be tested against each chunk. This added cost can be avoided by sending contiguous portions of the hash space in each chunk and indicating to s_B which fraction of B (described as a portion of the hash space) it needs to test against $F(A)$.

3.4 Virtual hosts

One key concern in a peer-to-peer system is the inherent heterogeneity of such systems. Randomly distributing functionality (e.g., keywords) across the system runs the risk of assigning a popular keyword to a relatively under-provisioned machine in terms of memory, CPU, or network capacity. Further, no hash function will uniformly distribute functionality across a hash range. Thus, individual machines may be assigned disproportionate numbers of keywords (recall that keywords are assigned to the host whose ID is closest to it in the hash range). Virtual hosts [6] are one technique to address this potential limitation. Using this approach, a node participates in a peer-to-peer system as several logical hosts, proportional to its request processing capacity. A node that participates as several virtual hosts is assigned proportionally more load, addressing heterogeneous node capabilities. Thus, a node with ten times the capacity of some baseline measure would be assigned ten virtual IDs (which means that it is mapped to ten different IDs in the hash range). An optional system-wide scaling factor for each node's number of virtual hosts further reduces the probability that any single node is assigned a disproportionately large portion of the hash range. This effect is quantified in Section 5, but consider the following example: with 100 hosts of equal power, it is likely that one or more hosts will be assigned significantly more than 1% of the hash range. However, with a scaling factor of 100, it is much less likely that any host will be assigned much more than 1% of the range because an "unlucky" hash (large portion of the hash region) for one virtual host is likely to be canceled out by a "lucky" hash (small portion of the hash region) for another virtual host on the same physical node.

3.5 Discussion

Two of the techniques described here, Bloom filters and caching, yield constant-factor improvements in terms of the number of bytes sent and the end-to-end query latency. Bloom filters compress document ID sets by about one order of magnitude, in exchange for either added latency or a configurable probability of false positives. Caching exploits re-referencing and sharing in the query workload to reduce the probability that document ID sets need to be sent. However, even together, these techniques leave both bytes sent and end-to-end query time roughly proportional to the number of documents in the system.

The third technique, incremental results, reduces the number of bytes sent and the end-to-end query latency to a constant in most cases. As long as the user wants only a constant number of results, only a constant amount of work will be done, regardless of how many possible results exist in the system. Incremental results yield no improvement in some unusual cases, however. If the user searches for several keywords that are individually popular but mostly uncorrelated in the document space, there may be a small but nonzero number of valid results.[1] If the number of results is nonzero but smaller than the number that the client requests, the system must consider the entire search space, rendering incremental results useless. In cases such as this, the entire search space must be considered, and incremental results will increase, rather than decrease, the number of bytes sent and the end-to-end query latency. However, caching may alleviate the problem if the words used are popular in search queries, and Bloom filters still yield approximately a ten-to-one compression factor.

We expect that searches containing popular but uncorrelated keywords will be rare. In our IRCache search trace, most of the queries with small numbers of results had uncommon (often misspelled) keywords. Uncommon keywords—i.e., those with few matching documents—are easy to handle, as discussed in Section 3.1. The system considers the least common keyword first, bounding the maximum size of any intersection set sent for the remainder of the query.

3.6 Ranking of Results

Two of our optimization techniques, Bloom filters and incremental results, complicate problem of ranking results. Bloom filters roughly convey membership in a set, but they do not provide the ability to order set members or to convey additional data with each member, such as a word's position in a document. The uncompressed response message containing $B \cap F(A)$ can contain document-ranking or word-position information, which would give server s_A enough information to generate rankings based on both keywords, k_A and k_B. However, in Section 3.1, we suggested eliminating this uncompressed

[1] One example of a difficult search is "OpenBSD birthday pony," suggested by David Mazières at New York University. In recent Google searches, these three keywords match two million, eight million, and two million documents, respectively. Only fifteen documents contain all three.

response message. Doing so eliminates the ability to consider k_A in any ranking techniques.

Incremental results can alleviate the problems with Bloom filters. If each chunk sent contains document IDs with strictly lower rankings than in previous chunks, then the first results returned to the client will be the best, though order within a chunk will not be preserved. However, in Section 3.3 we suggested sending contiguous portions of the hash space in each chunk to save processing time on server s_B. These two techniques are mutually exclusive.

We believe that ranking documents is more important than eliminating one additional message or saving processing time. However, this trade-off can be determined at run time according to user preference.

3.7 Load balancing

A vertically partitioned index distributes keywords randomly, resulting in a binomial (roughly normal) distribution of the number of keywords on each node. However, keyword appearance popularity (i.e., the size of the keyword's match-list) and search popularity are both roughly Zipf-distributed. Keyword appearance popularity determines the storage required, and keyword search popularity determines processing loads. Both contribute to network loads. The resulting storage, processing, and network loads are less evenly distributed than with a horizontally partitioned index. Virtual hosts alleviate the problem by assigning larger loads to more capable nodes, but they do not make load any more balanced. Increasing the size of the network and the number of documents results in somewhat more balanced load. As long as the network is over-provisioned, which many peer-to-peer networks are, we believe that load balancing will not be a problem.

4 Simulation Infrastructure

The simple analysis described above in Section 3 provides some insight into the potential benefits of our three approaches toward efficiently supporting peer-to-peer search. However, the actual benefits and tradeoffs depend heavily upon target system characteristics and access patterns. To test the validity of our approach under a range of realistic circumstances, we developed a simulation infrastructure implementing our three techniques. In this section, we discuss the details of this simulation infrastructure before presenting the results of our evaluation in Section 5.

4.1 Goals

Our goal in writing the simulator was to test the system with a realistic workload and to test the effects of parameters and features that did not lend themselves to tractable analysis. In particular, we tested the effects of the number of hosts in the network, the use of virtual hosts, the Bloom filter threshold, Bloom filter sizes, caching techniques, and the use of incremental results. We also tested the system's sensitivity to varying network characteristics.

The Bloom filter threshold refers to the document set size below which a host transmits a full list rather than a Bloom-compressed set. For small documents, the total bandwidth consumed for transmission to a remote host (for set intersection) may be so small that it may not be worth the CPU time required to compress the set. Eliminating the Bloom step further eliminates the need to return to the transmitting host to eliminate false positives from the intersection. Typically, we find that the extra CPU overhead and network overhead of returning the result is worth the substantial saving in network bandwidth realized by using Bloom filters. In Section 5, we quantify this effect for a variety of Bloom thresholds.

Bloom filter sizes affect the number of false positives transmitted during the search process. If the client is willing to accept some probability of false positives (a returned document containing only a subset of the requested keywords), sufficiently large Bloom filters can meet the client's accepted false-positive rate and eliminate the need to revisit nodes to remove false positives, as described in Section 3.1. That is, small Bloom filters result in significant compression of a keyword-set size at the cost of either generating more false positives in the result returned to the client or requiring the transmission of the intersection back to the originating host for false positive elimination.

4.2 Design

The simulator runs as a single-threaded Java application. We implement the inverted index, word-to-host mapping, and host measurement (in this case, random generation) in separate classes so that much of the simulator could be reused in a full implementation of our protocol. Our simulations use a real document set and search trace. The document set totals 1.85 GB of HTML data, comprising 1.17 million unique words in 105,593 documents, retrieved by crawling to a recursion depth of five from 100 seed URLs [4]. The searches performed are read from a list of 95,409 searches containing 45,344 unique keywords. The search trace is the IRCache log file described in Section 1. Note that the results presented in this paper are restricted to these particular traces. However, we do not expect the benefits of our techniques to differ significantly for other workloads.

Hosts in the network are generated at random based on configurable distributions for upload speed, download speed, CPU speed, and local storage capacity. We use three distributions for network speeds: one with all modems, one with all backbone links, and one based on the measurements of the Gnutella network performed by Saroiu et al [18]. This last heterogeneous set contains a mixture of modems, broadband connections (cable/DSL) and high-speed LAN connections. Our CPU speed distribution is roughly a bell curve, with a mean of 750 MIPS, and our local storage distribution is a heavy-tailed piece-wise function ranging from 1 MB to 100 MB. We experimented with a broad range of host characteristics and present the results for this representative subset in this paper. To generate random latencies, we place hosts at random in a 2,500-mile square grid and assume that network packets travel an average of 100,000 miles per second.

The time required to send a network message is the propagation time, as determined by the distance between the hosts involved, plus the transmission time, as determined by the minimum of the sender's upload speed and the recipient's download speed, and the size of the packet. The total network time for a search is the sum of the latency and transmission time for all packets sent among server nodes processing the query. We ignore the time spent by the client sending the initial query and receiving the results because these times are constant and independent of any search architecture, whether centralized or distributed.

Document IDs are assumed to be 128 bits. The time required to look up words in a local index or perform intersections or Bloom filter operations is based on the CPU speed and the following assumptions for operation costs: 1,500 simple operations per hit to look up words in an index, 500 simple operations per element to intersect two result sets, and 10,000 simple operations per document ID inserted into a Bloom filter or checked against a Bloom filter received from another host. We believe that in general, these assumptions place an upper bound on the CPU cost of these operations. Even with these assumptions, we find that network time typically dominates CPU time for our target scenarios.

We determine the number of virtual hosts to assign each simulated node based on its network and CPU speeds when compared to a baseline host. The baseline host has a 57.5 MIPS CPU and 30 Kbit/s network links. These speeds were chosen as those required to compute and transmit 5,000 Bloom operations per second. Each node is compared to the baseline host in three categories: upload speed, download speed, and CPU speed. The nodes's minimum margin over the baseline host in these three categories is rounded down and taken to be its number of virtual hosts.

To perform each query, the simulator looks up each keyword in the inverted index, obtaining up to M results for each, where M is the incremental result size. Each host intersects its set with the data from the previous host and forwards it to the subsequent host, as described in Section 3.1. Each node forwards its current intersected set as either a Bloom filter or a full set, depending on whether or not the set is larger than the Bloom threshold. After each peer performs its part of the intersection, any node that sent a Bloom filter in the first pass is potentially revisited to remove false positives. If the number of resulting documents is at least as large as the the desired number, the search is over. Otherwise, M is increased adaptively to twice what appears to be needed to produce the desired number of results, and the search is rerun.

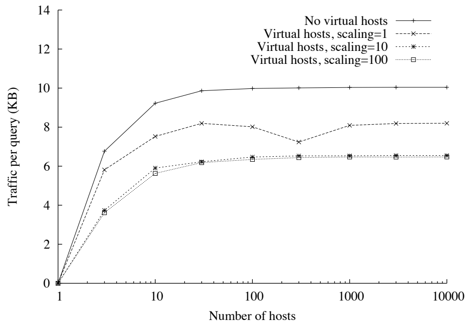
At each step, a host checks its cache to see if it has data for the subsequent host's document list in its local cache. If so, it performs the subsequent host's portion of the intersection locally and skips that host in the sending sequence.

4.3 Validation

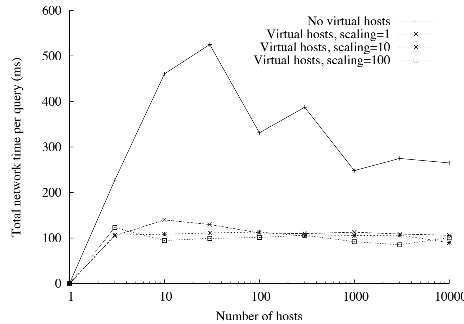
We validated our simulator in two ways. First, we calculated the behavior and performance of short, artificial traces by hand and confirmed that the simulator returns the same results. Second, we varied the Bloom filter size, m, in the simulator and compared the results to the analytical results presented in Section 3.1. The analytical results shown in Figure 5(b) closely resemble the simulated results shown in Figure 9(a).

5 Experimental Results

The goal of this section is to understand the performance effects of our proposed techniques on a peer-to-peer search infrastructure. Ideally, we wish to demonstrate that our proposed peer-to-peer search system scales with system size (total resource consumption



(a) The number of bytes sent increases very little beyond networks of 100 hosts. Enabling virtual hosts reduces the number of bytes sent by about 18%. Scaling the number of virtual hosts reduces the number of bytes sent by an additional 18%.



(b) Virtual hosts cut the amount of time spent transmitting by up to 60%. Scaling the number of virtual hosts yields a small additional improvement.

Fig. 7. Network scaling and virtual hosts

per search grows sub-linearly with the number of participating hosts) and that techniques such as Bloom filters and caching improve the performance of individual requests. Primarily, we focus on the metric of bytes sent per request. Techniques such as caching and the use of Bloom filters largely serve to reduce this metric. Reducing bytes per request has the added benefit of reducing total time spent in the network and hence end-to-end client perceived latency. We also study the effects of the distribution of network and CPU characteristics on overall system performance. One challenge with peer-to-peer systems is addressing the subset of hosts that have significantly less computation power and network bandwidth than is required to support a high-performance search infrastructure.

Finally, although we implemented incremental results, we do not present results for this technique here because our target document set is not large enough to return large numbers of hits for most queries. For our workload, this optimization reduces network utilization by at most 30% in the best case. However, we believe this technique will be increasingly valuable as the document space increases in size.

5.1 Scalability and Virtual Hosts

A key goal of our work is to demonstrate that a peer-to-peer search infrastructure scales with the number of participating hosts. Unless otherwise specified, the results presented in this section all assume the heterogeneous distribution [18] of per-peer network connectivity and the default distribution of CPU power described in Section 4. Caching and Bloom filters are both initially turned off. As shown in Figure 7(a), increasing the number of hosts in the simulation has little effect on the total number of bytes sent. With very small networks, several keywords from a query may be located on a single host, resulting in entirely local handling of parts of the query. However, beyond 100 hosts, this probability becomes insignificant, and each n-keyword query must contact n hosts, independent of the size of the system.

In addition to demonstrating the scalability of the system, Figures 7(a) and 7(b) also quantify the benefits of the use of virtual hosts in the system. Recall that when virtual hosts are turned on, each node is assigned a number of hosts based on its capacity relative to the predefined baseline described in Section 4. The virtual host scaling factor further multiplies this number of hosts by some constant value to ensure that each physical host is assigned a uniform portion of the overall hash range as discussed in Section 4. Overall, virtual hosts have a small effect on the number of total bytes sent per query. This is because enabling virtual hosts concentrates data mostly on powerful hosts, increasing the probability that parts of a query can be handled entirely locally. Virtual host scaling results in better expected load balancing, which very slightly decreases the amount of data that must be sent on average.

Although virtual hosts have little effect on how much data must be sent, they can significantly decrease the amount of time spent sending the data, as shown in Figure 7(b). By assigning more load to more capable hosts, the virtual hosts technique can cut network times by nearly 60%. Using virtual host scaling further decreases expected network times by reducing the probability that a bottleneck host will be assigned a disproportionate amount of load by mistake. Thus, while total bytes sent decreases only slightly as a result of better load balancing, total network time decreases significantly because more capable hosts (with faster network connections) become responsible for a larger fraction of requests.

5.2 Bloom Filters and Caching

Having established the scalability of our general approach, we now turn our attention to the additional benefits available from the use of Bloom filters to reduce network utilization. In particular, we focus on how large the Bloom filter should be and for what minimum data set size it should be invoked. Using Bloom filters for every transfer results in substantial unnecessary data transmissions. Any time a Bloom filter is used, the host using it must later revisit the same query to eliminate any false positives. Thus, Bloom filters should only be used when the time saved will outweigh the time spent sending the clean-up message. Figure 8 shows the total bytes transmitted per query as a function of the Bloom filter threshold, assuming the default value of 6 bits per Bloom entry. We find that the optimal Bloom filter threshold for our trace was approximately 300. Any set below this size should be sent in its entirety as the savings from using Bloom filters do not outweigh the network (not to mention latency) overhead of revisiting the host to eliminate false positives.

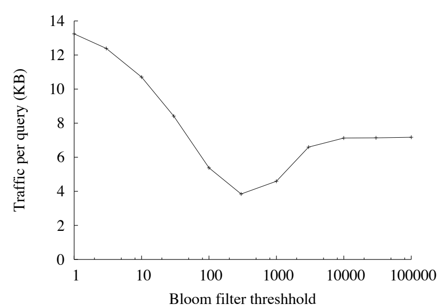
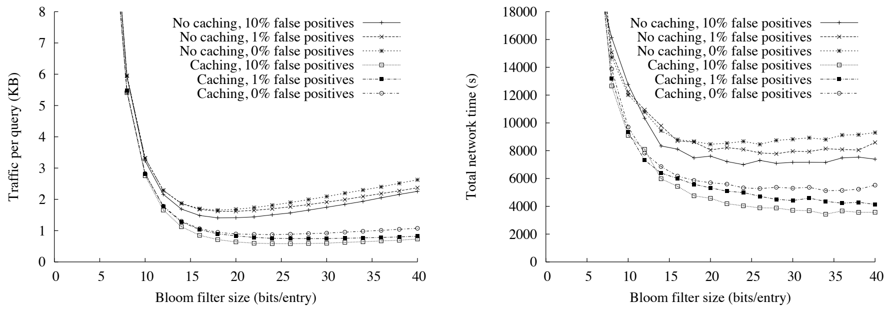


Fig. 8. Using Bloom filters less often significantly reduces the amount of data sent by eliminating the need to revisit nodes to eliminate false positives.



(a) Bytes per query (b) Latency plus transmission time

Fig. 9. Network costs as a function of Bloom filter size

Next, we consider the effects of varying the number of bits per entry in the Bloom filter and of caching on total network traffic. Figure 9(a) plots the total number of bytes transmitted as a function of the Bloom filter size. The two sets of curves represent the case when we enable and disable caching. Within each set, we set a maximum rate of allowable false positives in the set of documents returned to the user for a particular query, at 0%, 1%, and 10%. When the client allows 1% or 10% false positives, false-positive removal steps may sometimes be eliminated; increasing the Bloom filter size enhances this effect. Figure 9(b) shows that allowing false positives has significantly more effect on varying total network time than it does on bytes transferred as it eliminates a number of required message transmissions.

The effects of caching shown in Figure 9(a) are similar to those derived analytically in Figure 5(b). Caching decreases the total amount of data sent and increases the optimal Bloom filter size: in this case, from 18 bits per entry to 24 bits per entry. For optimal Bloom filter sizes of 18 and 24 bits per entry in the no-caching and caching cases respectively, our caching technique introduces more than a 50% reduction in the total number of bytes transmitted per query.

5.3 Putting It All Together

We now present the end-to-end average query times considering all of our optimizations under a variety of assumed network conditions. We break down this end-to-end time into the three principal components that contribute to end-to-end latency: CPU processing time, network transmission time (bytes transferred divided by the speed of the slower network connection speed of the two communicating peers), and latency (determined by the distance between communicating peers). Recall from Section 4 that we do not measure the time associated with either the client request or the final response as the size of these messages is independent of our optimization techniques.

Figure 10 shows three bar charts that break down total end-to-end search time under the three network conditions described in Section 4: WAN, Heterogeneous, and Modem. For each network setting there are four individual bars, representing the effects of virtual hosts on or off and of caching on or off. Each bar is further broken down into network transmission time, CPU processing time, and network latency. In the case of an all-modem network, end-to-end query time is dominated by network transmission time. The use of virtual hosts has no effect on query times because the network set is homogeneous. Caching does reduce the network transmission portion by roughly 30%. All queries still manage to complete in 1 second or less because, as shown in Figure 9(a) the use of all our optimizations reduces the total bytes transferred per query to less than 1,000 bytes for our target workload; a 56K modem can transfer 6 KB/sec in the best case. However, our results are limited by the fact that our simulator does not model network contention. In general, we expect the per-query average to be worse than our reported results if any individual node's network connection becomes saturated. This limitation is significantly mitigated under different network conditions as individual nodes are more likely to have additional bandwidth available and the use of virtual hosts will spread the load to avoid underprovisioned hosts.

In the homogeneous WAN case, network time is negligible in all cases given the very high transmission speeds. The use of caching reduces latency and CPU time by 48% and 30%, respectively, by avoiding the need to calculate and transmit Bloom filters in the case of a cache hit. Enabling virtual hosts reduces the CPU time by concentrating requests on the subset of WAN nodes with more CPU processing power. Recall that although the network is homogeneous in this case we still have heterogeneity in CPU processing power as described in Section 4.

Finally, the use of virtual hosts and caching together has the most pronounced effect on the heterogeneous network, together reducing average per-query response times by 59%. In particular, the use of virtual hosts reduces the network transmission portion of average query response times by 48% by concentrating keywords on the subset of nodes with more network bandwidth. Caching uniformly reduces all aspects of the average query time, in particular reducing the latency components by 47% in each case by eliminating the need for a significant portion of network communication.

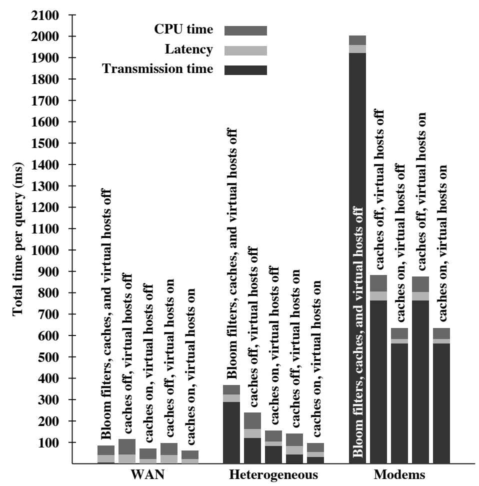


Fig. 10. Isolating the effects of caching, virtual hosts, and different network characteristics for optimal Bloom threshold (300) and Bloom filter sizes (18/24 for caching on or off).

6 Related Work

Work related to ours can be divided into four categories: the first generation of peer-to-peer systems; the second-generation, based on distributed hash tables; Web search engines; and database semijoin reductions. We dealt with DHT-based systems in Section 1. The others, we describe here.

The first generation of peer-to-peer systems consists of Napster [14], Gnutella [8], and Freenet [5, 9]. Napster and Gnutella both use searches as their core location determination technique. Napster performs searches centrally on well-known servers that store the metadata, location, and keywords for each document. Gnutella broadcasts search queries to all nodes and allows each node to perform the search in an implementation-specific manner. Yang and Garcia-Molina suggest techniques to reduce the number of nodes contacted in a Gnutella search while preserving the implementation-specific search semantics and a satisfactory number of responses [20]. Freenet provides no search mechanism and depends instead on well-known names and well-known directories of names.

Web search engines such as Google [3] operate in a centralized manner. A farm of servers retrieves all reachable content on the Web and builds an inverted index. Another farm of servers performs lookups in this inverted index. When the inverted index is all in one location, multiple-keyword searches can be performed with entirely local-area communication, and the optimizations presented here are not needed. Distributing the index over a wide area provides greater availability than the centralized approach. Because our system can take advantage of the explicit insert operations in peer-to-peer systems, we also provide more up-to-date results than any crawler-based approach can.

The general problem of remotely intersecting two sets of document IDs is equivalent to the database problem of performing a remote natural join. We are using two ideas from the database literature. Sending only the data necessary for the intersection (i.e., join) comes from work on semijoin reductions [1]. Using a Bloom filter to summarize the set of document IDs comes from work on Bloom joins [12, 13].

7 Conclusions

This paper presents the design and evaluation of a peer-to-peer search infrastructure. In this context we make the following contributions. First, we show that our architecture is scalable; global network state and message traffic grows sub-linearly with increasing network size. Next, relative to a centralized search infrastructure, our approach can maintain high performance and availability in the face of individual failures and performance fluctuations through replication. Finally, through explicit document publishing, our distributed keyword index delivers improved completeness and accuracy relative to traditional spidering techniques.

One important consideration in our architecture is reducing the overhead of multi-keyword conjunctive searches. We describe and evaluate a number of cooperating techniques—Bloom filters, virtual hosts, caching, and incremental results—that, taken together, reduce both consumed network resources and end-to-end perceived client search latency by an order of magnitude for our target workload.

Acknowledgments

We are grateful to Duane Wessels of the IRCache project (supported by NSF grants NCR-9616602 and NCR-9521745) for access to their trace data files. We would also like to thank Lipyeow Lim for access to the 1.85 GB HTML data set we used for our document trace. Finally, Rebecca Braynard, Jun Yang, and Terence Kelly provided helpful comments on drafts of this paper.

References

1. Philip Bernstein and Dah-Ming Chiu. Using semi-joins to solve relational queries. *Journal of the Association for Computing Machinery*, 28(1):25–40, January 1981.
2. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
3. Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *7th International World Wide Web Conference*, 1998.
4. Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *The VLDB Journal*, September 2000.
5. I. Clarke. A distributed decentralised information storage and retrieval system, 1999.
6. Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, October 2001.
7. Li Fan, Pei Cao, Jussara Almeida, and Andrei Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proceedings of ACM SIGCOMM'98*, pages 254–265, 1998.
8. Gnutella. http://gnutella.wego.com/.
9. T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2000.
10. David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.
11. David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of ACM Conference on Principles of Distributed Computing (PODC)*, 2002.
12. Lothar Mackert and Guy Lohman. R^* optimizer validation and performance evaluation for local queries. In *ACM-SIGMOD Conference on Management of Data*, 1986.
13. James Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558–560, May 1990.
14. Napster. http://www.napster.com/.
15. Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.
16. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM'01*, 2001.
17. Antony Rowstron and Peter 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 Systems Principles (SOSP'01)*, 2001.
18. Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN'02)*, January 2002.
19. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM SIGCOMM'01*, 2001.
20. Beverly Yang and Hector Garcia-Molina. Efficient search in peer-to-peer networks. Technical Report 2001-47, Stanford University, October 2001.