Improving Hybrid Keyword-Based Search

Matei A. Zaharia and Srinivasan Keshav

Abstract: We present a hybrid peer-to-peer system architecture for keyword-based free-text search in environments with heterogeneous document popularities and user lifetimes, such as file-sharing applications. Our system incorporates several novel design elements that increase its effectiveness. These include the use of central servers to collect global statistics, a search algorithm that uses these statistics to decide whether to flood or use a DHT, adaptive flooding, and delayed publishing. The gain due to these techniques is quantified by simulation: compared to a simple hybrid approach where queries are sent to the DHT if a flood doesn't return enough results, our system can achieve 2.8 times smaller first response time, 6.8 times smaller last response time, and 2.6 times smaller bandwidth use, while receiving at least as many results as are desired for most queries and maintaining a similar success rate.

I. Introduction

Consider searching a large distributed peer-to-peer system for a document described by keywords {A,B,C}. If you wanted to search for this document by specifying *all* three keywords, Distributed Hash Tables (DHTs) provide a scaleable, efficient, and elegant solution. But they are poor at searches where only one or two of the keywords form the query. This is because such *partial keyword* searches require the DHT to maintain inverted indices per keyword and join the search results. For searches involving two or more common keywords, with correspondingly large inverted indices, this join is expensive. Surprisingly, well-known flooding techniques pioneered by Gnutella and Kazaa are far more efficient at such searches, provided the documents being searched for are highly replicated (i.e. are 'popular').

Based on this insight, and similar to the work described in Reference [15], we propose a hybrid architecture that combines DHTs and flooding to efficiently support partial keyword searches. Unlike previous work, we use centralized servers to determine and disseminate *global statistics* on keyword and document popularity. Using these statistics, an efficient search algorithm can be selected for each query: If the query's keywords match the keywords for a popular document, it is flooded among *local index nodes* (high-capacity nodes each storing the list of documents available at 50-100 *end nodes*). Otherwise, a *global index* DHT that maps from keywords to local index nodes is used to determine the set of local index nodes matching each keyword in the query; the query is forwarded to the nodes in intersection of these sets.

Motivated by existing measurement studies, we make the following assumptions:

1. Documents are identified by a *title* consisting of a small number of *keywords*.
2. Queries are of the form "find all documents having a given set of keywords in their titles" (AND queries).
3. Document availability and keyword popularities follow a Zipfian distribution [17]. We call a highly replicated document *popular,* and a keyword that appears in many titles *common*.
4. Document popularities are global: i.e. we do not exploit regional variations in the popularity of a specific document, such as those considered in the work on interest-based locality [27].
5. User capabilities are heterogeneous: some users have "server-like" characteristics (high bandwidths and long uptimes) [20].
6. The numbers of documents and keywords are small enough that statistics about document and keyword popularity can be collected and distributed to users.
7. Users are satisfied with a limited number of results for each query [8].

Our system optimizes two metrics: (a) the number of results, up to number of results desired [8] and (b) response time. Analysis of the system shows that costs of operations grow at most logarithmically in the number of users, making it highly scalable, and simulations show that our optimizations have a significant effect.

II. Related Work

There are several peer-to-peer systems designed specifically for efficient partial-keyword search, based on DHTs [5, 6, 8, 9, 10, 12, 13, 14]. Our system incorporates several ideas presented in past work, such as partitioning of global indices by keyword instead of by document [8], incremental set intersection [8], node promotion [13], compressed Bloom filters to represent set membership [9, 24, 25] and compression of partial results [9]. However, these systems do not take into account the dichotomy between popular and unpopular documents. To our knowledge, this dichotomy was first explored in the work that most closely matches ours [15] where a hybrid search engine using both a DHT and flooding is proposed. This system first floods each query, then sends it to a DHT-based search engine (PIER [14]) if too few results are found.

Our system makes several improvements over this design. We propose the use of global statistics collected by centralized elements to determine whether or not to flood a query based on a *flooding threshold*, low priority flooding for unpopular documents with popular keywords, adapting the flooding threshold, and adaptive flooding. We also propose three other optimizations: delayed publishing to

decrease the impact of short-lived peers, maximally different neighbor set selection to increase flood effectiveness, and efficient batch updates to the Chord DHT.

III. ARCHITECTURE

Our hybrid design (Figure 1) incorporates a DHT on keywords as well as an unstructured flooding network between local index nodes. The flooding network structure and global index implementation are left unspecified: a Gnutella-like flooding system and the Chord DHT (with the join algorithms described in [8]) are assumed for some of our optimizations, but most of them will work with a random-walk flooding network such as Gia [16] or one that organizes neighbors based on interests [27], or with a more complex global index like PIER [14].

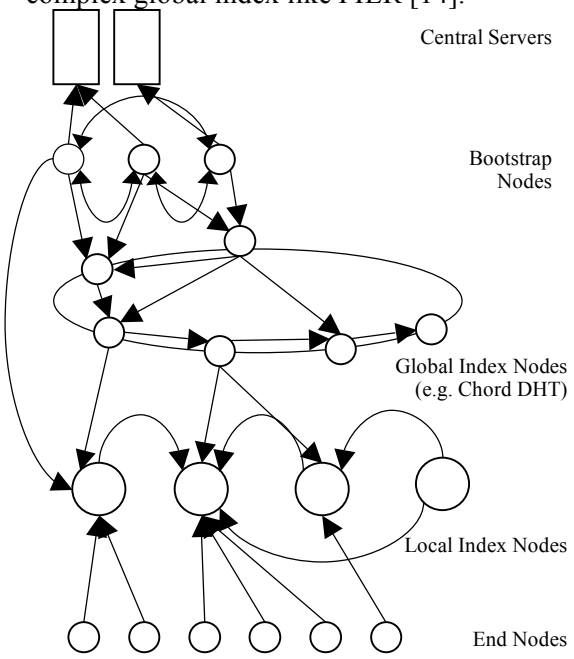


Figure 1. System Architecture.

A set of replicated *central servers* compute global statistics as described in Section IV.A. *Bootstrap nodes* that participate in a group communication protocol keep track of global and local index nodes, choose which end nodes to promote to higher levels in the system, and assign new end nodes to local index nodes.

A small fraction of nodes (say 0.1% to 0.2%) with high bandwidths and long uptimes are promoted to *global index* nodes by the bootstrap nodes, forming a Distributed Hash Table that maps each keyword to a list of local index nodes having documents with that keyword. They use search algorithms such as the ones in [8] can to execute queries.

Another fraction of nodes (say 1% to 2%) are promoted to *local index* nodes that store the lists of documents owned by a certain number (50-100) of other *end nodes*. Each local index node participates in a flooding network of neighboring local index nodes (with unspecified structure), and can flood queries to them. Local index nodes execute queries on behalf of the end nodes connected to them, choosing a search algorithm as explained in section III.A. They process queries directed to them by other local index nodes. They also regularly ping their end nodes to check whether they're alive, and report this information to bootstrap nodes for use in node promotion. Finally, they regularly report document and keyword frequencies among their end nodes to the central servers, and receive back the resultant global statistics.

Finally, each *end node* has a set of documents that it shares with other end nodes. It publishes its title list to several local index nodes when it joins the system (for duplication in case one disconnects), and, if sufficiently powerful and long-lived, it may be promoted by a bootstrap node to higher levels in the hierarchy.

IV. OPTIMIZATIONS

A. Search Algorithm Selection Using Global Statistics

A query can be performed in two ways in our system:

1. *Flooding*: An end node sends the query to a local index node, with a "time-to-live" (TTL) hop count. The local index node checks for matches among its list of documents and sends these to the originator. If the TTL is larger than zero, it floods the query to all its neighbors with a decremented TTL.
2. *DHT Search*: A local index node uses the DHT to find the intersection of the lists of local index nodes matching each keyword in the query. It then forwards the query to all nodes in the intersection.

How should a local index node decide which algorithm to use? In [15] the node always floods first, invoking the DHT only if no results are found. This can be inefficient. Instead, we propose that a set of central servers periodically compute the most common keywords and the relative popularities (average number of copies of the document per end node), of the most popular (well-replicated) documents in the system and disseminate this information to all local index nodes. Given a set of keywords in a query and these statistics, local index nodes then go through the following decision process:

1. If the sum of the relative popularities (i.e. expected responses per node) of all the popular documents that match the query's keywords is higher than some threshold *t*, flood with high priority.
2. If *any* keyword is *not* in the list of common keywords, use the DHT (since a join is cheap if it is initiated at this keyword).
3. Otherwise, flood with low priority.

Step 3 handles the case of an unpopular document with common keywords. In reality, this case might not occur very often, because most users realize that certain keywords are very common and avoid them. Nevertheless, if it does occur, neither search algorithm is efficient: flooding might not locate the document without using a large TTL, and DHT search will yield a large list of local index nodes that have each of the keywords in the query

but not the document itself. We propose flooding with a low priority and high TTL: using the DHT would provide no advantage, because it would require the query to be sent to many local index nodes anyway. We believe the introduction of priorities when flooding to be novel.

We note in passing that our use of centralization in what is, in all other respects, a decentralized system, is justifiable because of its simplicity and effectiveness. Replication of statistics across multiple central servers and participation in a group communication protocol adds reliability, so that the overall system fails only if *all* the central servers fail, an unlikely event. In any case, failure of these servers only affects performance, not correctness. We believe that central servers do not reduce scalability because they only collect aggregate results from local index nodes at large time intervals; the time interval can be tuned to reduce the load on the central servers, if necessary. We envisage the central servers to be managed by an administrative entity, because they provide a convenient point of management and control for such an entity.

B. Adaptive Flooding

Due to the Zipfian distribution of document popularities, flooded queries are likely to return very large result sets. For instance, Reference [15] shows that 29% of queries in Gnutella receive more than 100 results, and some queries receive as many as 1500 results. However, most users require only a few results, especially in file-sharing systems where they usually have a small set of files (e.g all files by some author) in mind. Thus, bandwidth is wasted when flooding queries for very popular documents. In our system, local index nodes only flood queries for popular documents, so we could address this by using a fixed, small TTL. However, this introduces a 'magic number' that we would prefer to avoid.

Gnutella addresses this issue using dynamic querying [26], where a query is re-flooded with a larger TTL if too few results are found. Instead, we propose an *adaptive* flooding algorithm. Essentially, a node guesses how many results are being found in parallel by its peers based on its depth in flooding tree and the known average number of results per end node so far. If this exceeds the needed number of results, flooding stops. Our method only performs one flood, so it generates less traffic and has shorter response times than dynamic querying.

Specifically, for a node at depth N in the flood, let $d_1,\ldots,d_N$, be the degrees of its ancestors (itself included), $r_1,\ldots,r_N$ be the numbers of results found by each ancestor, $e_1,\ldots,e_N$ be the numbers of end nodes at each ancestor, and let M be the number of results desired. Then,

1. Calculate the average number of results per end node, $\bar{r} = \sum_{i=1}^{N} r_i / \sum_{i=1}^{N} e_i$.
2. Estimate the number of results found in total, $R = \bar{r} \cdot \sum_{i=1}^{N} \prod_{j=1}^{i-1} d_j$ (assuming isotropicity).
3. If $kR>M$, do not flood further. (k is a tuning parameter to correct for overestimation, which we set to ~0.8).

Note that this algorithm is also applicable in a random-walk flooding network, like [16]: R will then be known precisely, and can be passed down along the search path.

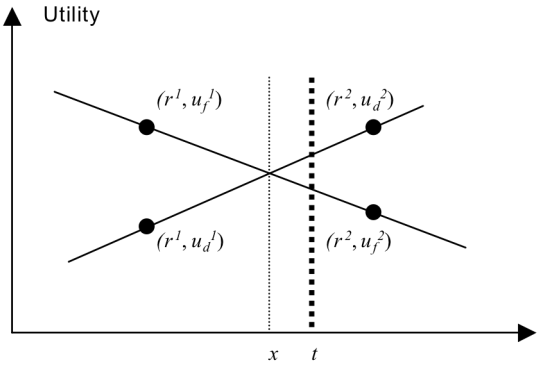
C. Adaptive Flood Threshold

An important parameter in our system is the flooding threshold, t. Let the *utility* of a search be a function directly proportional to $\min(r, R)$ where r is the number of results found and R is the number of results desired, and inversely proportional to both the response time and the amount of bandwidth used for the query. For popular documents, where the expected number of results per node is large, flooding provides more utility than DHT search, and for unpopular ones, DHT search provides more utility. Clearly, it is optimal for t to be chosen as the *point of indifference*, where flooding and DHT search provide equal utility.

Note that this point can change over time depending, among other things, on the number of end nodes, the number of documents they store, and the number of nodes promoted to global index status. Therefore, the system should adjust t over time, instead of using a fixed value. Also, adaptive thresholding makes the system more robust. For example, in case of failure of the DHT, all queries would be flooded because t would decrease rapidly.

As a simplification, assume that the utility from flooding or DHT search can be modeled as a *linear* function of r, where r is the expected number of responses per node for that query. r can be estimated by the sum of popularity values for all the popular documents matched by a query. We propose the following algorithm at each local index node for adjusting t over time:

1. For each query, assign a probability p that it will be chosen as a data point for adaptation. For simplicity, we choose p to be a linear function of $|r - t|$.
2. With probability p, use both flooding and DHT for the query and carry out steps 3 and 4.
3. Compute the utilities of each type of search.
4. Each query will result in the computation of two data points (r, u_f) and (r, u_d), where u_f is the utility from flooding, and u_d is the utility from a DHT search. If $r < t$, we expect $u_f > u_d$, otherwise, $u_f < u_d$



5. After determining Q data point queries, for every two pairs of points $\{(r^1, u_f^1), (r^1, u_d^1)\}$ and $\{(r^2, u_f^2), (r^2, u_d^2)\}$ let x be the X coordinate of the intersection of the

line passing through (r^1, u_f^1) and (r^2, u_f^2) and (r^1, u_d^1) and (r^2, u_d^2). Set the new value of t to be the median x value from all $C(Q,2)$ pairs. Essentially, each intersection point is an estimate of the point of equal utility, or point of indifference. The median value therefore is a good estimate of the new value for t.

As an extension, local index nodes can report their t to a bootstrap node, which can notify all local index nodes when the average t reported changes considerably from the previous one. This allows the system as a whole to adapt very quickly to changes in population.

D. Delayed Publishing

A significant number of users participating in P2P systems are *butterflies*, who log on for a very short time to download a file then disconnect without providing any files to other users. Publishing a butterfly's list of titles to a local index node and (more importantly) to the DHT is a waste of resources if the user will disconnect several minutes later, since it is unlikely that any other user will manage to download any file from the butterfly in this time. Therefore, we suggest only publishing a user's title list to the local and global indices after a certain delay (longer for the DHT than for the local index).

Note that some P2P systems, such as Kazaa [3] and BitTorrent [23], split each file into fixed-size "chunks". This would reduce the publishing delays necessary, since even a butterfly might be able to share some chunks during its lifetime to make publishing its titles worthwhile.

E. Efficient Batch Updates to the Chord DHT

If the Chord DHT [4] is used for the global index, then a simple optimization can be applied to reduce the network traffic required for the batch updates performed whenever a new end node connects to the system, or an end node disconnects. Suppose that a local index node wishes to perform an update involving M keywords on a DHT with N nodes. The naïve algorithm for doing the search is to independently find the DHT node for each keyword and perform the update, which requires $O(M \log N)$ hops. Our batch update algorithm is as follows:

1. Sort the keywords in increasing order of their images under the hash function.
2. Starting from any DHT node, find the node responsible for the first keyword and perform the update required.
3. Find the node responsible for each subsequent keyword *starting at the node responsible for the previous keyword*, and perform the update.

Because there are M keywords, the average number of nodes between the nodes responsible for successive keywords is N/M, so the node responsible for the next keyword can be found in $O(\log(N/M))$ hops. Therefore, the total number of messages is $O(M \log(N/M))$.

If the Chord nodes are relatively long-lived, we can go further and cache the DHT nodes we found responsible for each keyword, then perform each update starting from the closest known node behind it in the ring (either the node for the previous keyword or a cached node). Effectively, each local index node keeps its own finger table pointing to nodes all around the Chord ring.

F. Maximally Different Neighbor Set Selection

While most flooded queries will be for popular documents, some of these documents will be less popular than others, and some floods will be for rare documents with popular keywords. We propose a simple optimization to improve the effectiveness of flooding in finding rarer items: have each local index node select its neighbor set so as to maximize the number of distinct documents they have. After enough end nodes connect to it, a new local index node should obtain addresses of more candidates for neighbors than it requires, and query them for their lists of document titles that it doesn't already have, using Bloom filters [24] for compression. Then, it should check each combination of possible neighbors and select the one that will provide the largest set of distinct documents that the node doesn't already index. This way, there will be more queries for which the node will to find a result nearby.

In interest-based flooding networks where neighbors are selected to be maximally similar [27], we propose instead that some fraction of neighbors be maximally different, to exploit the well-known small-world phenomenon.

V. Analysis and Simulation

A detailed analysis of the costs of searches and node joins in our system can be found in [28]. Here, we will merely summarize our results by stating that with adaptive flooding and incremental DHT search, the costs of DHT/flood searches and node join/leave operations are either not proportional to the number, N, of nodes in the system, or at worst $O(\log(N))$. Therefore, the system scales well to large numbers of users.

We have also used discrete event simulation to compare our peer-to-peer design with others and observe the effect of each optimization. We simulate Zipfian document and keyword frequencies, as well as flooding fully. But for simplicity, we do not simulate incremental DHT search algorithms such as [8] in detail; instead, we set the delay for DHT searches to $b \log(g)$, where b is a constant base delay and g is the number of DHT nodes; we then simulate querying each local index node in the intersection returned by the DHT. Also, we simulate only exact queries for specific documents (using the fetch-at-most-once model in [17]), because it is difficult to generate realistic inexact queries given sets of documents and keywords.

We used the following base values for simulation parameters for all the tests reported here:

TABLE I
BASE VALUES FOR SIMULATION PARAMETERS

Argument	Base Value
Total simulation time	80000 s (about 2.2 hours)

Number of documents	1500
Zipf parameter for document popularity	1.0
New end node creation interval	4.0 s
End node lifetime distribution	The distribution observed in [20] for Gnutella hosts
Average end node query interval	120 s
Average initial number of documents per end node	20
Fraction of nodes promoted to local index	5%
Local index node degree	3
Maximum flood TTL	3
Maximum results desired	25
Number of local index nodes per user (replication factor)	2

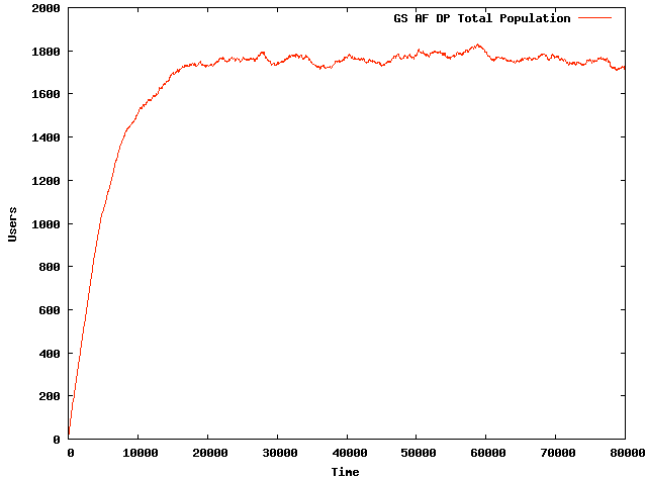
We use several query performance metrics to present our results. They are defined in Table II:

TABLE II
QUERY PERFORMANCE METRICS

Name	Meaning
Recall	Percentage of queries that found a matching document, given that at least one available document matched the query
Results	Average number of results returned per query, for queries that found more than zero results.
BWC	Bandwidth cost in kilobytes per query; the cost of publishing is also included in the total cost
FRT	Average first response time for queries that found results.
LRT	Average last response time for queries that found results.

A. System Stability

With the parameters in Table I, Figure II shows that a stable population of about 2200 users is achieved after about 10 hours. Therefore, results are presented only the queries made between times 40,000 and 80,000.



B. Effect of Optimizations

We compared six systems against each other:

TABLE II
SYSTEMS COMPARED

Name	Description
Pure Flooding	Flooding among local index nodes, with fixed TTL of 3.
Pure DHT	All queries looked up in a DHT.
Simple Hybrid	Similar system to [15]: queries are first flooded with TTL of 2, then looked up in a DHT if fewer than $R_{max}/2$ results are received from the flood.
GS Hybrid	Hybrid with algorithm selection based on global statistics and adaptive thresholding.
GSH-AF	GS Hybrid with adaptive flooding.
GSH-AF-DP	GS Hybrid with adaptive flooding and delayed publishing; the local index publishing delay is 60.0 s, and the global index publishing delay is 600.0s.

TABLE III
PERFORMANCE RESULTS

Search Method	Recall	Results	BWC	FRT	LRT
Pure Flooding	95.4%	63.1	1.60	1.50	3.47
Pure DHT	99.9%	262.6	11.57	3.12	3.32
Simple Hybrid	99.8%	43.1	1.93	4.34	14.37
Hybrid with global stats	98.6%	66.6	1.65	1.53	2.81
Hybrid with global stats and adaptive flooding	98.6%	26.6	0.73	1.48	2.17
Hybrid with global statistics and adaptive flooding and delayed publishing	98.6%	26.5	0.73	1.38	2.15

We now study effect of each of the optimizations presented in Section IV. Consider each column in turn:

In terms of recall, DHT-based systems perform better, since the DHT provides perfect recall for rare queries that might find no matches when flooded. Our results actually overestimate the recall of flooding because the number of total documents is small and the flood depth is large for the number of users present: with the full flood depth of 3, 1+3+27=31 local index nodes are "hit" in a flood, and they store the title lists of about 31/0.05 = 620 users, or 28% of the total number of users in the system. In a real system with 1,000,000 peers online, this would be inefficient, and much a smaller percent of the peers would be "hit" by each flood. The number of documents would also be larger by about 100 times, and although there might be 2-5 times more users per local index node, flooding would be far less effective at finding rare items.

In the second column, flooding returns well above the required number of results (25). This is because it is difficult, if not impossible, to choose a flooding threshold that returns just the right number of results. Even with dynamic querying (which we did not simulate), it is quite likely that increasing TTL by 1 would return more results than required, and that too, at the expense of an increase in the response time. The simple hybrid scheme reduces the number of results since it floods to a TTL of only two, instead of the depth of four chosen with pure flooding. However, it too returns more results than desired. In contrast, with adaptive flooding (the last two rows), the number of results closely matches the number required. Note that the number of results returned with a pure DHT is also large, because we do not implement the incremental intersection algorithms in Reference [8] where the DHT search is abandoned after the requisite number of results have been obtained.

The bandwidth cost of a pure DHT solution exceeds that of flooding because joining inverted indices for common keywords is expensive. We see that the simple hybrid has an intermediate bandwidth cost because it floods with a small TTL but uses a DHT to search only for rare documents. Our system improves this performance in two ways. First, adaptive flooding makes the cost of flooding much lower for popular documents. Second, with delayed publishing, we do not incur publishing costs for transient

nodes. Thus, the full system (last row) has the least bandwidth cost.

The first response time is low for flooding and high for DHTs. We see that the hybrid system's mean FRT is actually higher than that of a flood, because for a rare document, it pays both the flooding cost as well as the DHT cost. In contrast, our system uses global statistics to *avoid* flooding for popular documents. This makes our system's performance as good as flooding.

The last response time is the time taken for the last response or the R_{max}'th response (25^{th} response). We see that both pure flooding and the DHT have poor LRTs. Flooding incurs a high LRT for rare documents, and a DHT incurs a high LRT for all documents, because of having to navigate potentially very long hops in the underlying topology. In contrast, the LRT for our schemes are pulled in because of efficient decision making: we only use the DHT for rare documents, avoiding the large LRT from flooding, and the DHT cost for popular documents.

C. System Scalability

We also tested system scalability by varying the number of end nodes. We ran tests with three different end node creation intervals: 2.0s, 1.0s, and 0.5s. The approximate stable populations of these tests were, respectively, 3600 end nodes, 7200 end nodes, and 14,400 end nodes. Table V shows the results:

TABLE VI
QUERY METRICS FOR DIFFERENT SIZE SYSTEMS

Population	Recall	#Results	FRT	LRT
3600 users	97.8%	63.5	4.5	6.1
7200 users	97.7%	74.9	4.8	6.6
14000 users	98.4	98.9	5.2	7.8

The average number of results and average first and last response times for flood queries were similar in all three cases, because flood search efficiency does not depend on the total number of users. However, the average numbers of results for DHT searches grew linearly with the number of users: from 28.7 for 3600 users, to 58.1 for 7200 users, to 120.6 for 14,400 users. In the largest test, DHT searches returned more results on average than flood searches. These results show the need for incremental DHT search [8], which was not simulated, in large systems.

VI. CONCLUSIONS

We have presented a number of techniques for improving the performance of peer to peer networks that support hybrid search, including centralized elements to collect global statistics to choose a search algorithms for each query, adaptive techniques that avoid the use of 'magic numbers', efficient batch updates for Chord, and delayed publishing. We have evaluated these techniques through simulation and shown their effectiveness: compared to the simple hybrid approach, our system can achieve 2.8 times smaller first response time, 6.8 times smaller last response time, and 2.6 times smaller bandwidth use, while receiving at least as many results as are desired for most queries and maintaining a similar success rate. In future work, we plan to implement our results in a real system. The system also scales well with user population.

VII. REFERENCES

[1] Napster, http://www.napster.com

[2] Gnutella, http://www.gnutella.com

[3] Kazaa, http://www.kazaa.com/us/index.htm

[4] Stoica, I, Morris, R, Karger, D, Kaashoek, M.F, and Balakrishnan, H. "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications" ACM SIGCOMM 2001.

[5] Dwarkadas, S. and Tang, C, "Hybrid Global-Local Indexing for Efficient Peer-to-Peer Information Retrieval," NDSI 2004.

[6] Harren, M, Hellerstein, J. M, Huebsch, R, Loo, B. T, Shenker, S, and Stoica, I, "Complex Queries in DHT-based Peer-to-Peer Networks," IPTPS 2002.

[7] Suel, T, Mathur, C, Wu, J, Zhang, J, Delis, A, Kharrazi, M, Long, X, and Shanmugasundaram, K, "ODISSEA: A Peer-to-Peer Architecture for Scalable Web Search and Information Retrieval," *6th International Workshop on the Web and Databases (WebDB)*, June 2003.

[8] Reynolds, P, and Vahdat, A: "Efficient Peer-to-Peer Keyword Searching," unpublished.

[9] Li, J, Loo, B.T, Hellerstein, J.M, Kaashoek, M.F, Karger, D.R, and Morris, R, "On the Feasibility of Peer-to-Peer Web Indexing and Search," IPTPS 2003.

[10] Lopes, N.A.F, "A Peer-to-Peer Inverted Index Implementation for Word-Based Content Search," *Simposio Doutoral do Departamento de Informatica da Universidade do Minho*, October 2003.

[11] Schmidt, C, and Parashar, M, "Flexible Information Discovery in Decentralized Distributed Systems," *Twelfth IEEE International Symposium on High-Performance Distributed Computing*, June 2003

[12] Dwarkadas, S, Tang, C, and Xu, Z, "Peer-to-Peer Information Retrieval using Self-Organizing Semantic Overlay Networks," ACM SIGCOMM 2003.

[13] Mahalingam, M, Tang, C, and Xu, Z: "pSearch: Information Retrieval in Structured Overlays," *First Workshop on Hot Topics in Networks* October 2002.

[14] Huebsch, R, Hellerstein, J.M, Lanham, N, Loo, B.T, Shenker, S, and Stoica, I: "Querying the Internet with PIER," VLDB 2003.

[15] Loo, B.T, Hyebsch, R, Stoica, I, and Hellerstein, J.M: "The Case for a Hybrid P2P Search Infrastructure," IPTS 2004.

[16] Chawathe, I, Ratnasamy, S, Breslau, L, Lanham, N, and Shenker, S: "Making Gnutella-like P2P Systems Scalable," ACM SIGCOMM 2003.

[17] Gummadi, K.P, Dunn, R.J, Saroiu, S, Gribble, S.D, Levy, M, and Zahorjan, J, "Measuring, Modeling and Analysis of a Peer-to-Peer File-Sharing Workload", SOSP 2003.

[18] Gribble, S.D, Gummadi, K.P, Levy, H.M, and Saroiu, S, "An Analysis of Internet Content Delivery Systems," OSDI 2002.

[19] Dictionary Facts - Oxford English Dictionary. http://www.oed.com/about/facts.html

[20] Gribble, S.D, Gummadi, K.P, and Saroiu, S, "Measuring and Analyzing the Characteristics of Napster and Gnutella Hosts," *Multimedia Systems Journal*, vol. 8 issue 3, November 2002.

[21] Garcia-Molina H, Kamvar, S.D, and Schlosser, M.T, "The EigenTrust Algorithm for Reputation Management in P2P Networks," *Proceedings of the Twelfth International World Wide Web Conference*, May, 2003

[22] Golle, P, Leyton-Brown, K, and Mironov, I, "Incentives for Sharing in Peer-to-Peer Networks," *Proceedings of Electronic Commerce'01*, 2001.

[23] BitTorrent, http:// bitconjurer.org/BitTorrent/

[24] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, July 1970.

[25] Mitzenmacher, M, "Compressed Bloom Filters," Twentieth ACM Symposium on Principles of Distributed Computing, August 2001.

[26] Gnutella Developer Forum: Gnutella Dynamic Query Protocol 0.1. http://www.limewire.com/developer/dynamic_query.html

[27] Sripanidkulchai, K., Maggs, B., and Zhang, H., ``Efficient Content Location Using Interest-Based Locality in Peer-to-Peer Systems", IEEE INFOCOM'03, April 2003.

[28] Zaharia, M.A. and Keshav, S., "Efficient Search Algorithms for Hybrid Peer-to-Peer Systems," University of Waterloo Technical Report UW-CS-2004-55.