Gossip-based Search Selection in Hybrid Peer-to-Peer Networks

M. Zaharia and S. Keshav
School of Computer Science, University of Waterloo, Waterloo, ON, Canada
matei@matei.ca, keshav@uwaterloo.ca

SUMMARY

We present GAB, a search algorithm for hybrid P2P networks, that is, networks that search using both flooding and a DHT. GAB uses a gossip-style algorithm to collect global statistics about document popularity to allow each peer to make intelligent decisions about which search style to use for a given query. Moreover, GAB automatically adapts to changes in the operating environment. Synthetic and trace-driven simulations show that compared to a simple hybrid approach that always floods first, trying a DHT if too few results are found, GAB reduces the response time by 25-50% and the average query bandwidth cost by 45%, with no loss in recall. GAB scales well, with only a 7% degradation in performance despite a tripling in system size.

KEY WORDS: Peer-to-peer systems; search; global statistics; gossip.

1. INTRODUCTION

A hybrid peer-to-peer search network combines an unstructured flooding network with a structured Distributed Hash Table (DHT)-based global index [1,2]. In such networks, partial keyword queries can either be flooded to all peers, or the set of peers storing documents corresponding to each keyword can be looked up in the DHT with the results intersected in-network [3] or at the initiator. Which search method should a query use? Flooding is efficient for popular (i.e well-replicated) documents but inefficient for rare documents. Therefore, a reasonable solution is to first flood a query to a limited depth, and, if this returns no results, suggesting that the document is rare, submit the query to the DHT [1,2]. This allows cheap and fast searches for popular documents and simultaneously reduces the flooding cost for rare documents. However, this comes at the expense of an increase in the response time for rare documents and wasted bandwidth due to unfruitful floods.

As an alternative, we present GAB (**G**ossip **A**daptive Hy**B**rid), a gossip-based approach to collect global statistics that allows peers to predict the best search technique for a query. GAB dynamically adapts to changes in the operating environment, making it relatively insensitive to tuning parameters. Its design does not depend on the choice of the DHT or of the unstructured flooding network: any of the solutions described in the literarture [4], for example, are adequate.

Compared to a non-adaptive hybrid approach [1,2], GAB achieves a 25-50% smaller response time, and reduces mean query bandwidth usage by 45%. GAB scales well, with only a 7% degradation in performance despite a threefold increase in system size.

Section 2 presents GAB and Section 3 evaluates it using simulations. We describe related work in Section 4 and conclude in Section 5.

2. ADAPTIVE SEARCH ALGORITHM SELECTION

This section outlines our search algorithm. Section 2.1 is an overview of hybrid search systems and Section 2.2 highlights the search-selection problem in such networks. Section 2.3 shows how we collect global statistics, and Section 2.4 how we use these statistics. Finally, Section 2.5 presents an algorithm to self-tune the flooding threshold control parameter.

2.1 Hybrid peer-to-peer systems

A hybrid search network [1,2] has four classes of nodes: end nodes, local index nodes, global index nodes, and bootstrap nodes (Figure 1). An end node has a set of documents that it shares with other end nodes. It publishes its title list to one or more local index nodes that are assigned to it by a bootstrap node.

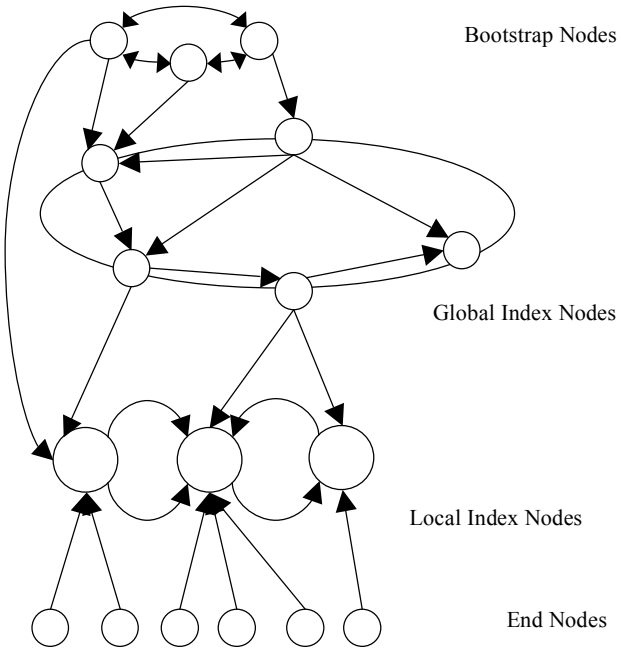

Figure 1. Hybrid P2P search network.

Local index nodes, also called ultrapeers, store the lists of documents present at a certain number (typically 30-100) of end nodes and execute search queries on their behalf. Every local index node participates in an unstructured flooding network. It can also forward queries to a DHT maintained at global index nodes. A small fraction of nodes with good connectivity and long uptimes are promoted to *global index nodes* by the bootstrap nodes, forming a DHT that maps each keyword to a list of local index nodes that index documents with that keyword [3]. Having several node classes leverages heterogeneous peer capabilities: measurements indicate that although most peers have low bandwidth connectivity and small connection durations, some have "server-like" characteristics and can be used as local or global index nodes, or even bootstrap nodes. For instance, recent work shows that 6% of Napster users and 7% of Gnutella users reported bandwidths of T1 or greater, and that about 10% of sessions last more than 5 hours each [12]. Such nodes would be ideally suited for promotion [14].

2.2 The search algorithm selection problem

An ideal search algorithm should return no more than some desired number of results, R_{max}, while minimizing both the query response time and the mean bandwidth cost per query. In a hybrid search network, these criteria are met if a peer can, by looking at the query keywords, decide if these keywords match a widely replicated document or not, using this information to either flood the query or look up the keywords in a DHT. In past work, observations of result size history, keyword frequency, keyword pair frequency, or sampling of neighboring nodes have been used to determine document rarity to reduce publishing costs [2]. However, this only uses local or neighbor information. Instead, GAB collects global statistics about document availability and keyword popularity using gossip to make the search selection decision.

We describe GAB in the context of a Gnutella-like network where *ultrapeers* index content stored at *end nodes* [5]. With GAB, an ultrapeer uses histograms of (a) the number of other ultrapeer nodes whose indices contain a given keyword and (b) the fraction of ultrapeer nodes that index a copy of a given document to predict the search technique that is best for a given query. It then compares this prediction to an actual measurement of search effectiveness, and uses the error to adapt future predictions. We describe collection of global statistics in Section 2.3, use of these statistics in Section 2.4, and adaptation in Section 2.5.

2.3 Gathering global statistics

GAB combines two algorithmic techniques: gossip-based computation and probabilistic counting. We first describe gossip-style algorithms and the challenge in using gossip for computing statistics. We then describe duplicate-insensitive probabilistic counting. Finally, we combine these two ideas and describe GAB's operation in detail. Note that a gossip-based approach to collecting global statistics is ideally suited to P2P networks because it is decentralized, robust, and results in every peer learning of the global state.

GAB is a gossip-based algorithm. A typical gossip algorithm proceeds in rounds, where a round has two phases. In the first phase, each participating node, in parallel, chooses a random other node and sends it some information. Then, in the second phase, each node combines the information it receives (from zero or more other nodes) with information it already has. Each node propagates this combined information in the next round.

The main challenge in using a gossip-based algorithm to collect global statistics is that information propagates in random node order during a gossip. Therefore, the algorithm must work correctly independent of communication order, and even if a node participates in a gossip multiple times. For example, if one wanted to count the number of nodes in an ensemble, simply gossiping a count is insufficient, because of the potential for double-counting. Instead, we use the order and duplicate-insensitive probabilistic counting technique for multisets pioneered by Flajolet and Martin [9].

The intuition behind probabilistic counting is that if a random event, such as a series of coin tosses, is carried out by many nodes in parallel, the probability that *some* node tosses a long sequence of consecutive heads (or tails) increases. Hence, if we find the *largest* number of consecutive heads (or tails) tossed by an ensemble of nodes, we can estimate the size of the ensemble. The maximum value in an ensemble is order and duplicate-insensitive, allowing the use of gossip amongst the ultrapeers to compute the maximum count value.

Our goal is to estimate the popularity of a particular document. Therefore, each ultrapeer conducts a coin-tossing experiment for each distinct document that it indexes: the greater the number of the ultrapeers that index a particular document, the larger the maximum count of consecutive heads (or tails) that we can expect in the set of coin toss experiments corresponding to that document.

More precisely, when an ultrapeer sees a document title it hasn't indexed already, it tosses a coin up to k times and counts the number of heads it sees before the first tail. It saves this result in a variable called *CT*. The ultrapeer then gossips its *CT* values for all titles it indexes with the other ultrapeers. During gossip, for each title, each ultrapeer computes the maximum value of *CT,* i.e. *maxCT*. If a document is widely replicated, its expected *maxCT* value will be larger than the expected *maxCT* value for a rare document. Moreover, the number of ultrapeers with the document is roughly 2^{maxCT}. Using this intuition, each ultrapeer can get an approximate count of the number of other ultrapeers that have that document title, which it maintains in a histogram.

We now state our algorithm in more depth. GAB consists of three distinct components (1) a *synopsis structure* that approximates the true histogram of document and keyword popularity (this

generalizes an approach proposed by Nath et al [6]) (2) a synopsis *fusion* algorithm to merge synopses, and (3) a randomized *gossip* algorithm to disseminate synopses among ultrapeers [7, 8].

The synopsis generation algorithm uses a duplicate-insensitive counting technique for multisets [9]. Consider the synopsis corresponding to a histogram of document (or, more precisely, document title) popularity. To create this synopsis, each ultrapeer, for each unique title in its index, does a coin tossing experiment that returns a value *CT(title, k)*, defined as: for the given title, toss a fair coin up to *k* times, and return either the index of the first time 'tails' occurs or *k*, whichever is smaller[1]. We require $k > 1.5\ logN$, where *N* is the maximum expected number of ultrapeers. The result of a coin toss experiment is represented by a bit vector (abbreviated 'bitvec') of length *k* with the *CT(title, k)*th bit set. The theory of probabilistic counting tells us that if the first 0 bit counting from the left in *max(CT(title,k))*, where the maximum is computed over the matching titles at all ultrapeers, is at position *i*, then the count associated with that bitvec is, with high probability, $2^{i-1}/0.77351$: the 'magic number' in the denominator comes from the mathematical principles described in [9][2].

A GAB synopsis actually has three parts: a *document title* synopsis, a *keyword* synopsis, and a *node count* synopsis. The title synopsis is a set of tuples {(*title*, *bitvec*)}, where *title* is a list of keywords that describe a document, and *bitvec* is a bit vector representing a coin tossing experiment. The keyword synopsis is similar. Finally, the node count synopsis is a bit vector counter that counts the *number* of nodes represented by that complete synopsis. The complete synopsis therefore is of the form {node_count_bitvec, {(title, bitvec),…, (title, bitvec)}, {(keyword, bitvec),…, (keyword, bitvec)}}.

The synopsis fusion algorithm is: (a) If two tuples in the combined title or keyword synopses have the same title or keyword, then take the bitwise-OR of the two corresponding bitvecs. This has the effect of computing the max of the two bitvecs in an order-insensitive manner. (b) Update the node count synopsis using the local value of the node_count bitvec. (c) To keep a synopsis from growing too large, if the size of the fused synopsis exceeds a desired limit *L*, discard tuples in order of increasing bitvec value (treating the bitvec as a *k*-bit integer) until the limit is reached. Note that at start time, synopsis counts are small, and it is possible that a popular document (with a small bitvec count) may be accidentally pruned. To compensate, a ultrapeer can skip the pruning step if the value of the node counter in the union of the synopses is smaller than some predefined threshold.

During initialization, each ultrapeer generates a synopsis of its own document titles and keywords and labels it as its 'best' synopsis. In each round of gossip, it chooses a random neighbor and sends the neighbor its best synopsis. When a node receives a synopsis, it fuses this synopsis with its best synopsis and labels the merged synopsis as its best synopsis. As with any other gossip algorithm, this results in every ultrapeer, with high probability, getting the global statistics after (*log N*) rounds of gossip.

Note that the gossip adds a bandwidth overhead to the system, which is the price to pay for getting global statistics. However, this cost is paid rarely since global statistics change rarely. So, running the gossip protocol as little as once a day is likely to be adequate in practice: this frequency can be tuned to be as low as necessary to control the overhead. Moreover, the cost is amortized over all the queries in the system, so, as the search load increases, the amortized cost for gossip actually decreases.

---

[1] *k* limits the size of the bit vector. If k=32, generation of such a vector is fast, requiring only one multiplication and addition operation for linear congruential random number generation [10], followed by the x86 Bit Scan Forward instruction and a lookup in a 32-element pre-computed bit vector array.

[2] Note that a synopsis is approximate: the number of documents can be estimated only to the closest power of 2, so the estimate may have an error of up to 50%.

2.4 Search selection using global statistics

Given a synopsis and a set of query keywords, an ultrapeer first determines if it has sufficient local matches. If so, it is done. If not, it computes the expected number of results for that set of keywords as follows: it adds the approximate counts for the titles in its 'best' synopsis that contain all the query keywords. We denote by r the sum of these approximate counts divided by the approximate number of ultrapeers N. For a given query, r represents the expected number of matching titles at any ultrapeer. The greater the number of titles in the P2P network that match a particular set of keywords, the larger the value of r. If r exceeds a threshold t, many matches are expected, so the ultrapeer floods the query. If the flood returns no results after a conservative timeout, it uses the DHT to search for each keyword, requesting an in-network join, if that is possible.

If $r < t$ and *any* keyword in the query is *not* in the common keyword synopsis, the ultrapeer uses the DHT because a join is cheap if it is initiated using this keyword [3]. Otherwise, the document is both rare and has common keywords, so the only option is to flood the query. This is done, if possible, with an indication that this query has low priority. The idea is that flooding will need a large flood depth. The low priority ensures that the request is handled only if there is adequate search capacity.

2.5 Adaptation of flood threshold

An important parameter in our system is the flooding threshold, t, expressed in terms of the expected number of matching titles at any ultrapeer. If t is too small, that is, we flood even when r is 'small' then too many queries will be flooded and *vice versa*. In either case, the system will be inefficient.

Unfortunately, it is hard for a system administrator to choose a threshold value that is optimal for all operating environments. Worse, the optimal threshold can change over time depending, among other things, on the number of end nodes, the number of documents they store, the search load, and the available bandwidth to each ultrapeer. Therefore, GAB adjusts t over time, instead of using a fixed value. Adaptive thresholding also makes GAB more robust: in case of failure of the DHT, all queries would eventually be flooded because t would rapidly decrease.

Intuitively, an ultrapeer should choose a search algorithm that maximizes a search's utility. For widely-replicated 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. GAB adapts the flooding threshold by computing the utility of *both* flooding and DHT search for a randomly chosen set of queries. If the current threshold is correct, then when GAB chooses to flood, the utility from flooding that query should be greater than the utility of using a DHT and vice versa. Otherwise, the threshold should be modified so that future queries make the right choice.

Adapting the threshold therefore requires us to define a utility function quantitatively. We base our measure of utility on four considerations. First, getting at least one result is a lot better than getting none. So, the first term represents the benefit from this. Second, because there is little use in finding thousands of results if a user is just searching for one particular document, the marginal utility per extra result should decrease sharply beyond some point. We approximate this by assuming that users require no more than some maximum number of results, R_{max}, beyond which each further result contributes zero utility. Therefore, the utility of receiving R results is proportional to min (R, R_{max}). Third, since only the first *Rmax* results are useful to the user, the utility should be proportional to the response time T of the min(R, R_{max})'th result, which we call the *last response time*. Finally, the cost of a query should include its bandwidth cost B. Assuming linearity, we denote the utility function U:

$$U = (R>0\ ?\ 1\ :\ 0) + w_1 * min(R, R_{max}) - w_2 * T - w_3 * B,$$

where w_1, w_2, and w_3 are normalized weights (with respect to the first term) chosen by a user. We choose to add and subtract the components of utility instead of multiplying or dividing them to prevent large fluctuations when one of the numbers is very small or very large. Note that R, R_{max}, T and B can be computed for each search if a partial bandwidth cost is carried with each search request and returned in the reply.

The optimal value for t is the point of indifference, where flooding and DHT search provide equal utility. This motivates the following technique for adapting t: we compute the utility from *both* flooding and using the DHT for a certain number of queries (doing it for all queries would be too expensive). We then compare the two values. If the current value of r is less than the threshold (suggesting that we use the DHT), but flooding turned out to have more utility, then t is too large, so we need to make it smaller. Symmetrically, if r exceeds the threshold, but the utility from the DHT exceeds that from flooding, then the threshold has to be made larger. In making these adjustments, we do not want to be sensitive to the results from a single query. Therefore, we average the adjustments from a set of queries as follows:

1. For each query, compute r, the expected number of results per ultrapeer. Recall that t is the threshold, in terms of expected number of matching titles at any ultrapeer, beyond which we opt to flood instead of using the DHT. Because we expect variations in t to be small, we obtain more measurements around the current value of the threshold by choosing p, the probability that this query will be used for adaptation, 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 results in the computation of two 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 (refer to Figure 2). If $r > t$, we expect $u_f > u_d$, otherwise, $u_f < u_d$.

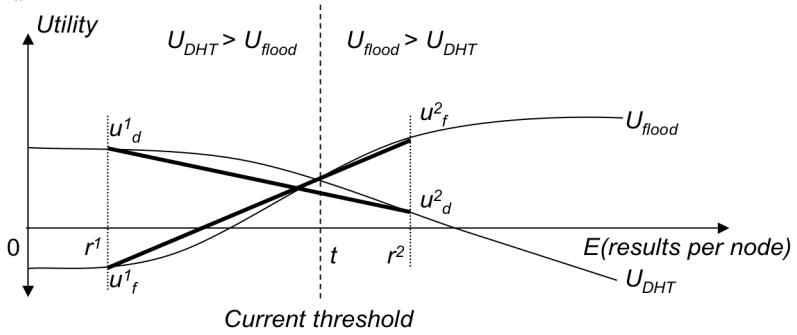

Figure 2: Adapting the flood threshold.

5. After Q queries, we need to update t. To first order, we approximate the utility from flooding and from a DHT as lines, so that their intersection is a reasonable estimator for t. Exponential averaging of this estimate allows us to deal with noisy estimates. Thus, for every two pairs of points $\{(r^1, u_f^1), (r^1, u_d^1)\}$ and $\{(r^2, u_f^2), (r^2, u_d^2)\}$ such that $r^1 < t < r^2$ let x be the abscissa 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). We set the new estimate of t to be the median x value from these pairs and use this to update t using an exponential averager with a forgetting factor of 0.05. In other words $t \leftarrow 0.95 * t + 0.05 * median\_abscissa\_of\_intersections$.
6. Optionally, the value of t computed at each ultrapeer can be gossiped so that every node is aware of the average value of t and uses this in its prediction.

3. EVALUATION

3.1 Simulator

We wrote a custom simulation system in Java to compare GAB with other well-known search techniques [11]. It accurately models end-node lifetimes and link capacities using empirically observed distributions [12,13], as well as a flooding network and a Chord-like DHT. To save time and because DHT performance is well-studied, we do not simulate interaction between DHT nodes. Instead, we set the delay for DHT searches to $b \log n$, where b is a constant and n is the number of DHT nodes. However, we do account for the bandwidth cost and delay in querying each ultrapeer in the result set returned by the DHT.

When an end node leaves, we model the deletion of its index both from ultrapeers as well as from the DHT. By modeling end-node churn, which is an important factor in real peer-to-peer systems, we capture the costs of DHT and ultrapeer updates both on node arrival and on departure. Because node *lifetimes* are chosen from a fixed distribution, we can increase the number of documents in the system, the total node population, as well as the number of simultaneously active nodes simply by modifying the node arrival rate.

We outline three optimizations that make our current simulator several orders of magnitude more efficient than the original implementation:

- Fast I/O routines. To our surprise, we found that logging an event may take longer than simulating it, because Java creates temporary objects during string concatenation. Using a custom, large StringBuffer for string concatenation greatly improves performance.
- Batch database uploads. Even prepared statements turn out to be much less efficient in updating the results database than importing a table into it from a tab-separated text file.
- Avoiding keyword search for exact queries. If we know a query to be exact, we match queries using an index on document IDs. This makes the simulator about 10 times faster on synthetic runs. This technique can also be used to simulate inexact queries if we were to pre-calculate the set of document ID's matching each subset of keywords.

3.2 Parameters

New end nodes join the system at a rate of one every 0.7 seconds, each bringing an average of 20 documents into the system, randomly chosen from a dataset of 20,000 unique documents, when they register with an ultrapeer. These documents are then are indexed by the DHT. The node lifetime distribution is from measurements of Gnutella in the literature [12] with a mean lifetime of 1.9 hours. End nodes emit queries on average once every 240 seconds, requesting at most 25 results (R_{max}). Documents and keywords are assumed to have a Zipfian popularity distribution with a Zipf parameter of 1.0.

We simulated about 1.9 million queries over a 22 hour period. We observed that a stable online population of about 10,000 active end nodes and 500 ultrapeers was achieved after about 20,000 simulated seconds (~6 hours) (Figure 3). Therefore, results are presented only for the queries made between 40,000 and 80,000 seconds.

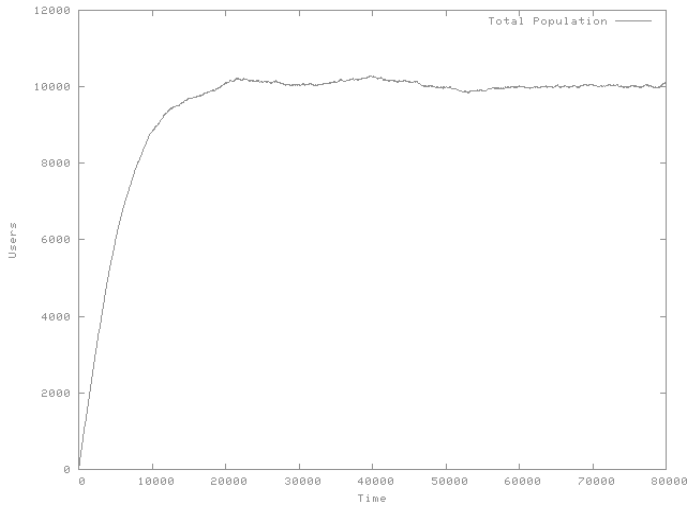

Figure 3: Active population size vs. time (s).

With these parameters, the total population over the simulation lifetime was about 114,000 end nodes. Although these numbers are still about an order of magnitude smaller than a real system, we believe that it is large enough for us to get meaningful comparisons between various search approaches. In a realistic system, the bandwidth costs will be about ten times larger. However, we expect the relative costs and benefits to be roughly the same as in our simulations.

Queries and document/keyword sets were generated in two different ways: (a) random exact search requests[3] according to the fetch-at-most-once model [13] (for results in 3.A-3.C) ; (b) by playing back partial keyword searches from the Gnutella data set [2] (for results in 3.D).

We compared GAB with the algorithms presented in Table I below.

Pure DHT	All queries are looked up in a DHT with in-network adaptive joins [3].
Simple Hybrid	Models the hybrid search algorithm [2]. Queries are first flooded to low depth then looked up in a DHT if fewer than 25 results are received from the flood after 2s. We studied both depth 2 and depth 3 floods and found that depth 3 outperformed depth 2. All comparisons are, therefore, with respect to a depth 3 flood.
Central Server	An ideal central server with zero request service time.

Table I: Algorithms evaluated in our simulations

To implement GAB itself, we used the gossip algorithm described in this paper to select between flooding and using a DHT search. For the flooding algorithm, we used one of two techniques. With GAB-Dynamic, when flooding is selected, the ultrapeer floods a query to a depth of 2, and, if insufficient results are found, floods to depth 4. This is the flooding technique used in P2P search networks such as Gnutella. GAB-Adaptive is a new flooding technique that limits the flood depth without requiring any re-flooding [11]. In adaptive flooding, a node receiving a flooded query estimates the number of results likely to have been already found by the other nodes in the flood as a function of its depth in the flood DAG, the mean fan out of the DAG, and the mean number of results found at each node of the DAG. If this number exceeds the number of desired results, the node does not flood the query any further. Otherwise, it propagates the query, passing on

[3] We only generated exact queries, since it is difficult to generate realistic partial queries

the depth, fan-out, and result cardinality values to its neighbors on a stack carried as part of the flooded search request message [11].

We compared the results for each approach using metrics in Table II.

Recall	Percentage of queries that found a matching document, given that there existed at least one matching document at some peer when the query was issued.
FRT	Mean first response time for successful queries.
LRT	Mean last response time, i.e response time for R_{max}th query, for successful queries.
BWC	Bandwidth cost in kilobytes per query; the cost of publishing and gossiping is also included in this cost.

Table II: Metrics used to compare the algorithms

To save space, we only report means. Also, for ease of comparison, we present **normalized** results for FRT, LRT, and BWC, where the value with respect to which normalization is being performed is shown in bold font. Standard deviations, which are all well under 5% of the mean value, are reported in an extended version of this paper [11].

3.3 Search approaches compared

System	Recall	FRT	LRT	BWC
Pure DHT	99.9%	1.18	0.62	1.63
Simple Hybrid	99.9%	**1.00**	**1.00**	**1.00**
GAB-Adaptive	99.9%	0.70	0.57	0.51
GAB-Dynamic	99.9%	0.89	0.52	0.47
Central Server	100.0%	0.41	0.21	0.22

Table III: Performance comparison of the algorithms

The search approaches are compared in Table III. The pure DHT approach has poor (and almost identical, though this is not apparent from the table) first and last response times because it cannot exploit document popularity to reduce response times. It also has the highest bandwidth cost. The simple hybrid approach [1,2] when compared to a pure DHT, reduces the first response time by about 20%. It also uses far less bandwidth because it avoids DHT lookups for popular documents. Unfortunately, it has a higher average last response time because rare documents must be both flooded and looked up.

Both GAB variants performs much better than Simple Hybrid. Their first and last response times are both lower than Simple Hybrid (and Pure DHT) because queries for known rare documents are sent directly to the DHT rather than being "tested" using a flood. Moreover, bandwidth costs are approximately halved because GAB saves doing a flood for queries that are sent directly to the DHT. This validates the gain in performance by the use of global statistics.

GAB-Dynamic has a much higher FRT than GAB-Adaptive, though it has a slightly *smaller* LRT. To understand this, we first compared adaptive and dynamic flooding in the absence of GAB. We found that adaptive flooding reduces the first response time by 21%, and the last response time by 34% [11]. This is because when an ultrapeer selects dynamic flooding, but finds no results at depth 2, then the subsequent depth-4 flood incurs a much higher FRT than an adaptive flood. Given this, we would expect GAB-Adaptive to have both a lower FRT and a lower LRT. However, when we introduced GAB, we found that the adaptive thresholding algorithm, in the steady state, directed nearly 50% of queries to the DHT with GAB-Dynamic, instead of only about 30% with GAB-

Adaptive. This is because when a query for a rare document is flooded using dynamic flooding, it is very likely to incur a high cost of a depth-4 flood. This causes a GAB-Dynamic ultrapeer to select the DHT more often than with Adaptive flooding, which has lower flood costs. Thus, GAB-Dynamic uses the DHT more often (which is high-delay), and when it uses flooding, may pay the cost of a depth-2 flood in addition to a depth-4 flood. Together, these effects substantially increase the FRT. However, the increased use of the DHT tends to pull in the cost of finding a rare document, which reduces the LRT slightly. This explains why the FRT with GAB-Dynamic is much higher than GAB-Adaptive, but the LRT is slightly lower.

A central server has perfect recall, 55% lower FRT and 70% lower LRT than even GAB. Moreover, the bandwidth cost is also 57% lower. We conclude that the price to pay for decentralization is roughly a doubling of every performance metric.

3.4 Adaptive thresholding

Figure 4 shows the times series of flooding threshold at two ultrapeers for a particular simulation run. For this environment, the optimal value of t is around 1.0E-4. The two ultrapeers being observed chose initial values of 1.0E-3 and 1.0E-5. Over time, both independently adapted the threshold and converged to the optimal value, illustrating GAB's adaptive behavior. The convergence time is about two hours, which is smaller than the lifetime of a typical ultrapeer. In practice, we expect t to be a rather slowly changing function of time, and a node could be given a hint about the current value of t by other ultrapeers when it is promoted to being an ultrapeer. This would greatly reduce the convergence time.

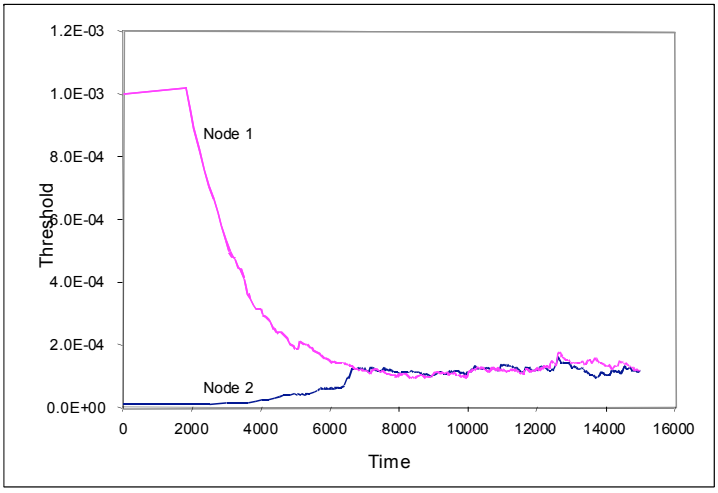

Figure 4: Threshold value t *(results/node) vs. time (s) for nodes with different initial values.*

3.5 Scalability

Intuitively, GAB should scale well with increases in end node population because DHT and gossip costs increase logarithmically with system size, and flooding costs, for a fixed flood depth and node degree, are constant. We validated this intuition by choosing three different values for the mean end node inter-arrival time: 1.0s, 0.4s, and 0.3s. As described earlier, this changes the mean number of end nodes in the system. The approximate stable active populations were, respectively, 7000 end nodes, 17,500 end nodes, and 23,300 end nodes, corresponding to total end node populations of roughly ten times this size. The normalized results are shown in Table IV.

Active Population	FRT	LRT	BWC
7000	**1.00**	**1.00**	**1.00**

17,500	1.00	0.85	1.06
23,300	1.03	0.80	1.07

Table IV: Evaluation of scalability

We observe that as the population more than triples, the first response times increase by 3% due to the need to consult larger DHT indices for rare items. However, the DHT is used only about 20% of the time, so its effect on the overall average is negligible. Note that LRT actually *decreases* slightly with increase in population size because, with more nodes, sufficient numbers of results are found with shallower floods. Bandwidth costs increase by about 7%, again mostly due to larger DHTs but also because as the number of users increases while keeping node degree and flood depth constant, the fraction of non-back edges increases and a flood is more widely propagated.

3.6 Trace-based simulations

To validate the conclusions from synthetic-workload based simulation, we also ran our experiments on a trace-based workload. The traces use the Planetlab-based monitoring infrastructure [2], and were obtained by simultaneously monitoring the queries and the results of these queries at 50 ultrapeers for 3 hours on Sunday October 12, 2003. This represents 230,966 distinct queries, 199,516 distinct keywords and 672,295 distinct documents. Newly joining nodes select a random subset of these documents to add to the system and emit queries randomly selected from the list of queries. Note that the traces do not allow us to model the effect of node churn. The results from this evaluation are show in Table V.

System	Recall	FRT	LRT	BWC
Simple Hybrid	87.0%	**1.00**	**1.00**	**1.00**
GAB	87.3%	0.59	0.45	0.67

Table V: Results from a trace-based simulation

For this more realistic trace-based workload, our results show that GAB has a 41% lower FRT than a simple hybrid peer-to-peer search network (compared to 30% lower with a synthetic workload); 55% lower LRT (43% with a synthetic workload), and 33% lower bandwidth cost (49% with a synthetic workload). Both systems have a lower recall than in synthetic-workload simulations.

Although differences in the workload make an apples-to-apples comparison between trace-based and synthetic-workload simulations impossible, we believe that the discrepancy in the reductions in FRT and LRT between these workloads is because trace-based queries are partial-keyword queries compared to the full-keyword queries in the synthetic workload. GAB's reduction in FRT and LRT compared to a simple hybrid is due to lower delays in finding rare documents, and the proportional gain for these documents can be expressed as (Flood delay)/(Flood delay + DHT lookup delay), a decreasing function of the number of DHT lookups. So, with fewer keywords, the trace-based workload shows higher proportional gains.

The recall is lower with the trace-based workload than in the synthetic workload because the trace has many more distinct documents, but in our simulation, each newly joining peer still selects only 20 of these documents to host. This results in fewer documents being stored per peer, increasing the probability that a document correctly located by the DHT becomes unavailable due to peer departure, and therefore decreasing effective recall.

4. RELATED WORK

Numerous DHT-based search systems have been proposed in the literature [4], including hybrid systems that combine DHT and flooding networks [1,2]. Extensions to DHTs to allow searches using only a subset of document title's keywords are also well-known [3, 14]. GAB builds on and extends this work by proposing gossip-based algorithms for search selection.

Gossip systems have been widely described in the literature, where they are also called epidemic algorithms [7, 8, 15]. We refer interested readers to [16] for an overview and survey of recent work in this area.

5. CONCLUSIONS AND FUTURE WORK

Our work makes two main contributions. First, we show how gossip-based computation of global statistics improves search efficiency, reducing both response time and bandwidth costs. Second, we show how to adapt a critical tuning parameter, the flood threshold, to changes in the operating environment. The use of a decentralized gossip-style state computation, combined with a DHT removes all centralized elements from our system, which permits good scalability. The adaptation process uses user utilities, and this allows system behavior to be controlled by intuitive 'control knobs'. We believe that the use of gossip to compute global state and the explicit use of utility functions to modify system behavior, are applicable to any large-scale distributed system.

We quantified gains from GAB using simulation on both synthetic and trace-based workloads. We found that, compared to a simple hybrid approach, our search algorithm can roughly halve the last response time and bandwidth use, with no loss in recall. Our algorithm scales well, with only a 7% degradation in performance with a threefold increase in system size.

We have implemented our system by modifying the Phex Gnutella client to use the OpenDHT framework. In current and future work, we plan to quantify the benefits from our algorithms for more realistic workloads.

Acknowledgements

We would like to gratefully acknowledge Boon Loo, then at UC Berkeley, for his traces of the Gnutella workload. Comments from the anonymous reviewers also helped to improve the quality and precision of this work. This research was supported by grants from the National Science and Engineering Council of Canada, the Canada Research Chair Program, Nortel Networks, Sun Microsystems Canada, Intel Corporation, and Sprint Corporation.

REFERENCES

[1] B.T. Loo, R. Huebsch, I. Stoica, and J.M. Hellerstein, "The Case for a Hybrid P2P Search Infrastructure," *Proc. IPTPS,* 2004.

[2] B.T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker and I. Stoica, "Enhancing P2P File-Sharing with an Internet-Scale Query Processor," *Proc. 30th VLDB Conference,* 2004.

[3] P. Reynolds, and A. Vahdat, "Efficient Peer-to-Peer Keyword Searching," *Proc. Middleware*, 2003.

[4] H. Balakrishnan, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking Up Data in P2P Systems," *Comm. ACM*, Vol. 46, No. 2, Feb, 2003.

[5] Gnutella, `http://www.gnutella.com`

[6] S. Nath, P. Gibbons, S. Seshan, and Z. Anderson, "Synopsis Diffusion for Robust Aggregation in Sensor Networks," *Proc. SenSys*, Nov. 2004.

[7] D. Kempe, A. Dobra, and J. Gehrke, "Gossip-Based Computation of Aggregation Information," *Proc. IEEE FOCS*, 2003.

[8] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Gossip Algorithms: Design, Analysis, and Applications," *Proc. INFOCOM 2005*, March 2005.

[9] P. Flajolet and G.N. Martin, "Probabilistic Counting Algorithms for Database Applications," *J. Computer and System Sciences*, Vol. 31, 1985.

[10] D. Carta, “Two fast implementations of the "minimal standard" random number generator,” *Comm. ACM*, Vol. 33, No. 1, pp.87-88, 1990.
[11] M. A. Zaharia and S. Keshav, “Efficient and Adaptive Search in Peer to Peer Networks,” U. Waterloo Technical Report 2004-55, 2004.
[12] S. Saroiu, K.P. Gummadi, S.D. Gribble, “Measuring and Analyzing the Characteristics of Napster and Gnutella Hosts,” *Multimedia Systems Journal*, Vol. 9, No. 2, pp. 170-184, August 2003.
[13] K.P. Gummadi, R.J. Dunn, S. Saroiu, S.D. Gribble, H.M. Levy, and J. Zahorjan, “Measuring, Modeling and Analysis of a Peer-to-Peer File-Sharing Workload,” *Proc. 19th SOSP*, October 2003.
[14] S. Dwarkadas, and C. Tang, “Hybrid Global-Local Indexing for Efficient Peer-to-Peer Information Retrieval,” *Proc. NSDI*, 2004.
[15] A. Demers et al “Epidemic algorithms for replicated database maintenance,” *PODC*, 1987.
[16] S. Keshav, “Efficient and Approximate Computation of Global State,” *ACM Computer Communication Review*, Jan. 2006.