Adaptive Peer-to-Peer Search

M.A. Zaharia and S. Keshav
School of Computer Science
University of Waterloo, Waterloo, ON, Canada
{mazahari, Keshav}@cs.uwaterloo.ca

Abstract: Peers in a peer-to-peer (P2P) system have widely differing performance characteristics and are subject to time-varying workloads. However, once a peer has been deployed, reconfiguring it is difficult, if not impossible. It is, therefore, important for a peer to adapt its behavior to variations in network and workload characteristics. In this paper, we present adaptive mechanisms for two specific problems in peer-to-peer search systems: efficient flooding in unstructured P2P systems and search algorithm selection in hybrid P2P systems. Both algorithms use a mathematical model to predict an expected result, compare the actual result to the prediction, and feed back the error to adapt future responses. This allows the system to automatically adapt to changes in the operating environment. Synthetic and trace-driven simulations show substantial gain due to these adaptive techniques. Adaptive flooding provides the same recall as dynamic querying, but with 22% lower bandwidth cost. Similarly, compared to a non-adaptive hybrid approach our adaptive algorithm can achieve a 51% smaller last response time, and 1.9 times smaller bandwidth use, with no loss in recall. Our algorithms scale well, with only a 3-9% degradation in performance with a 3x increase in system size. These lead us to conclude that adaptation can greatly improve performance of both existing and future peer-to-peer systems.

I. INTRODUCTION

The success of peer to peer (P2P) networks such as Kazaa, Gnutella, and eDonkey has underlined the need for good design principles for such highly distributed systems. We believe that algorithms for P2P systems must exhibit *adaptation, scalability,* and *intuitive control*.

Peers in a peer-to-peer (P2P) system have widely differing performance characteristics and are subject to time-varying workloads. Moreover, once a peer has been deployed, reconfiguring it is difficult, if not impossible. It is, therefore, important for a peer to *adapt* its behavior to variations in network and workload characteristics. In general, systems that adapt their behavior based on feedback control deliver good performance despite potentially poorly chosen initial values of tuning parameters. This allows such systems to be effective and efficient even when users know little about the operating environment.

P2P systems are highly distributed and must *scale* to very large sizes. Consequently, they should have the fewest possible number of centralized elements. Moreover, the scaling behavior of a P2P algorithm should be *mathematically* modeled in order to explore parts of the state space that cannot be reached through simulation or small prototype deployments.

Finally, P2P systems must expose 'control knobs' that are understandable by the lay user. These *intuitive controls* should adapt the system behavior to fit user needs. Essentially, this means that users should describe their utility function quantitatively, and the system should maximize this utility.

In this paper, we design adaptive, scalable, and intuitively controllable search algorithms for a file sharing system where peer nodes contain documents that they wish to share with other peers. Documents are identified by a *title* consisting of a small number of textual *keywords*. The search system supports *partial keyword* implicit AND searches of the form "find all documents having a given set of keywords in their titles".

We focus on two problems that arise in this context: search algorithm selection in hybrid P2P systems (Section II) and flooding in unstructured P2P networks (Section III). For both problems, we design adaptive algorithms using a 'predict-measure-adapt' design pattern, that mimics classical feedback flow control. We motivate these algorithms and understand their behavior through mathematical modeling. We also show how they can be controlled through intuitive control interfaces by explicit consideration of user utilities. Although we developed these algorithms in the context of P2P search systems, we believe that they embody general principles that are widely applicable in all highly distributed environments.

We evaluate our algorithms using an event-driven simulator (Section IV). Synthetic and trace-driven simulations show that our algorithms substantially improve over the best-known current techniques.

II. Adaptive Search Algorithm Selection

A. Overview

Hybrid peer-to-peer search networks (Figure 1) were recently introduced by Loo et al [1,2]. Such a network combines an unstructured flooding network of *local index nodes* with a structured Distributed Hash Table (DHT)-based *global index*. Searches can be performed either by flooding the unstructured network of local index nodes, or by doing a keyword-based lookup in the DHT with a subsequent join either in-network as in [3] or at the initiator. The problem of interest is: given a query, which search method should the system choose? References [1,2] propose to first flood a query, and, if this fails, submit the query to the DHT. Instead, we propose to collect *global statistics* to optimally select the best search technique for a query, adapting the search selection algorithm itself to changes in the network. The use of global statistics makes query optimization straightforward, and the use of adaptation makes our system relatively insensitive to tuning parameters.

We outline the hybrid peer-to-peer search system in Section II.B and present a decentralized algorithm for collection of global statistics in Section II.C. Our statistics collection algorithms leverage recent innovations in Order and Duplication Insensitive synopsis collection and randomized gossip originally introduced in the context of sensor networks. We conclude with adaptive techniques to dynamically modify the search selection algorithm by feeding back the prediction error in Section II.D.

We note that our algorithms do not depend on the choice of the DHT or of the unstructured network: any of the solutions described in Reference [4], for example, are adequate. Moreover, although presented in a limited context, our approach to distributed decision making using synopsis collection is quite general and can be applied to a variety of other distributed algorithms. Similarly, our adaptive algorithm for search algorithm selection that dynamically updates a system parameter based on observed utilities, is applicable to any decision making process under uncertainty and in a dynamic environment.

B. Hybrid P2P search networks

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.

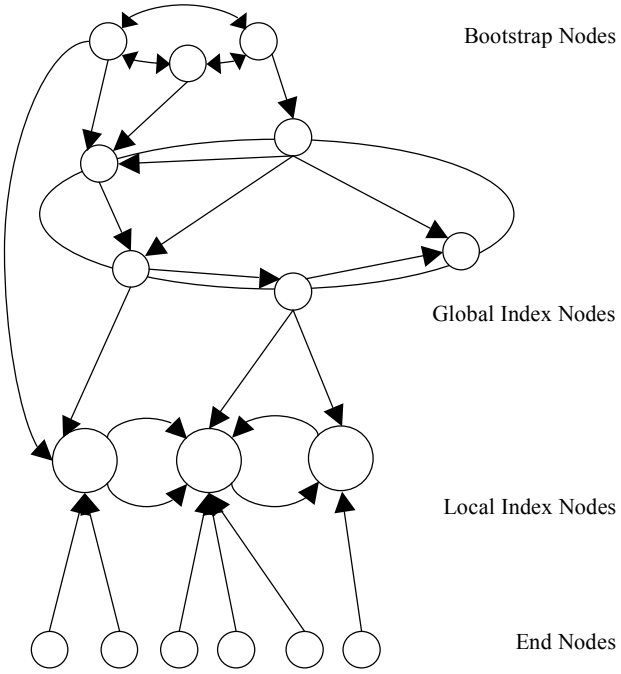

Figure 1. A Hybrid P2P Search Network

Local index nodes store the lists of documents present at a certain number (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. Finally, it participates in the collection of global statistics, as explained in Section II.D.

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 having documents with that keyword [5]. They use search algorithms such as the ones in [3,5] to execute queries. 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, Reference [6] found that 6% of Napster users and 7% of Gnutella users reported bandwidths of T1 or greater (which is probably an underestimate), and that about 10% of sessions last more than 5 hours each. Such nodes would be ideally suited for promotion [5].

C. The search algorithm selection problem

A hybrid system can use both flooding and DHT to perform searches as follows:

1. *Flooding*: A user on an end node sends a query with a globally unique ID to a local index node which floods to other local index nodes using the algorithm described in Section III.

2. *DHT Search*: A local index node uses the DHT to find the list of local index nodes corresponding to each keyword in the query. It then forwards the query to local index nodes in the intersection of these lists. As described in [3], to limit query costs, the size of lists corresponding to popular keywords are truncated during the execution of the query to a pre-determined maximum.

How should a local index node decide whether to use flooding or a DHT for a query? Ideally, a search system should optimize four metrics: (a) return all the results before the user's timeout, but (b) not more results than the user desires; (c) minimize response time and (d) minimize the bandwidth cost per query. Clearly, it is better to use the flooding network to search for popular documents and the DHT for unpopular documents. The problem is how to decide whether a query is for a popular or rare document.

Instead of always flooding first, as in [1,2] one could imagine that search selection could use heuristics similar to those advocated in [2] for *publishing* rare documents to the DHT. The four schemes proposed in [2] use observations of query result size, term frequency, term pair frequency, or a sampling of neighboring nodes to determine which documents are rare. However, they use only local (or neighbor) information. Instead, we propose to use *global statistics* about document availability and keyword popularity to make the search decision.

Specifically, in our approach, a local index nodes uses histograms of (a) the number of local index nodes that match *common* keywords (b) the fraction of local index nodes that index a copy of the *most well-replicated* documents 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. Our algorithm therefore reduces to three components: collection of global statistics (II.D), use of these statistics (II.E), and adaptation to errors in prediction (II.F).

D. Gathering global statistics

The key to good prediction is the use of accurate global statistics. We now present a distributed algorithm for gathering these statistics. It leverages the fact that the global statistics are slowly changing (on the time scale of days to weeks), and that optimal choice of search technique is weakly dependent on the accuracy of the statistics. Therefore, infrequent and approximate computation of these statistics is sufficient for good performance.

We compute global statistics using a variant of the distributed algorithm for finding 'Most Popular Items' outlined in [7]. Our algorithm has three components (1) a synopsis generation algorithm that approximates the true histogram and is both order and duplicate insensitive (ODI) (2) a synopsis fusion algorithm to merge two synopses to create a new synopsis and (3) (unlike [7]) a randomized gossip algorithm to disseminate synopses among the local index nodes [8,9] We describe each component in turn.

The synopsis generation algorithm at a local index node uses the duplicate-insensitive counting technique for multisets pioneered by Flajolet and Martin [10]. Consider the synopsis to describe the well-replicated document histogram. To create this synopsis, each local index node, for each title in its collection, does a coin tossing experiment *CT(k)*, defined as: toss a fair coin up to *k* times, and return either the index of the first time 'heads' occurs or *k*, whichever is smaller. Intuitively, the more popular a title is, the more coin tossing experiments will be done for it, and the higher the likelihood that at least one of the local index nodes obtained a long initial string of 'tails', increasing the largest value of *CT(k)* in the system. This allows us to estimate the number of local index nodes with that title without having to explicitly count them, and, in particular, in a manner that is insensitive to duplicate updates in the synopsis. The synopsis itself is represented by a bit vector length *k* with the $CT(k)^{th}$ bit set and $k > 1.5 \log N$, where *N* is the maximum number of local index nodes[1]. We will regard bit vector *a* to be greater than bit vector *b* if the first 0 in *a*, counting from the left, occurs at a greater index than in *b*.

Note that, these desirable properties of a synopsis come at a cost: the number of nodes can be estimated only to the closest power of 2, so the estimate may have an error of up to 33.3%. Nevertheless, this is adequate for our purposes.

A complete synopsis in our algorithm has three components: a *title* synopsis, a *keyword* synopsis, and a *node count* synopsis. The title synopsis at a local index node is a set of tuples (*title*, *bitvec*), where *title* is a list of keywords, 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 counts the *number* of nodes represented by that complete synopsis. Each local index, during initialization, computes and saves an experiment *CT(k)* and uses this to update the node counter synopsis as described below.

How large is a synopsis? To get a sense for this, note that Oxford English Dictionary Second Edition contains about 615,100 words [11] but of these only about 1,500 form the entire vocabulary of the Voice of America [12]. If we add another 1000 proper nouns as potential common keywords, we expect to see a total of about 2500 common keywords. Our Gnutella query traces (described in Section IV) had a mean keyword length of 6.5 bytes and a mean title length of 50.2 bytes. With a 4-byte synopsis and assuming no compression, a 2500 keyword synopsis takes 26.2 KB and 5,000 title synopsis takes 271 KB. Text compression using the Burrows-Wheeler Transform can reduce this (using measured compression ratios of 2.95 [13]) to 8.9 KB and 91.9 KB respectively, for a total of roughly 100 KB. Considering that the median object size in a file-sharing system is more than 3 Mb [14,15], we feel that a synopsis of ~100 KB is acceptable.

The synopsis fusion algorithm is: (a) take the component-wise union of the tuples in the three component synopses (b) if two tuples in the union of the title (keyword) synopses have the same title (keyword), then take the bitwise-OR of the two bitvecs (c) update the node count synopsis using the local value of *CT(k)* (d) if the size of the union exceeds a desired limit *L*, discard tuples in order of increasing bitvec value until the limit is reached.

Synopses are disseminated using randomized gossip using a variant of the schemes described in [8,9]. During initialization, each local index node generates a synopsis of its own titles and keywords and labels it as its 'best' synopsis. It then periodically chooses a random neighbor and sends it 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. Kempe et al [8] have shown that their Push-Synopsis randomized gossip scheme causes synopses to converge exponentially fast. Although the synopsis structure described here does not meet the linearity assumption in their work, we expect similar performance due to this scheme, and we verify this through simulation in Section IV.J.

Keyword and title popularities at a local index node are estimated from its current best complete synopsis as follows: if the first 0 bit counting from the left in a bitvec is at position *i*, then the count associated with that title (or keyword) is, with high probability, $2^{i-1}/0.77351$. This allows us to directly compute the number of nodes contributing to the synopsis and the number of local index nodes indexing a particular keyword. It is also trivial to compute the fraction of local index nodes that index a given document title.

One complication with this approach is that at start time, histogram counts are small, and it is possible that a popular document (with a small bitvec count) may be accidentally pruned. To compensate for this, *L* can be chosen to be large enough (say, 5-10 times the expected number of documents at any local index node).

[1] Generation of such a 32-bit vector requires only one multiplication and addition operation for linear congruential random number generation [34], followed by the x86 Bit Scan Forward instruction and a lookup in 32-element pre-computed bit vector array.

Alternatively, a local index node can skip the pruning step if value of node counter in the union of the synopses is smaller than some predefined threshold. This automatically prevents pruning in the absence of adequate information at the cost of some longer synopsis messages.

We also must deal with the fact that the set of documents at a local index node changes over time. Note that a synopsis with a node count much larger than N almost surely contains data from every local index node. However, a particular local node has no way of knowing whether it has, in fact, contributed to that synopsis, and, for that matter, *what* it has contributed to that synopsis. Therefore, when a node receives a synopsis with a node counter such that this counter, when added to the node counter of its current 'best' synopsis would be larger than some maximum value V, it simply drops both the incoming and the 'best' synopsis. It then re-computes a local synopsis based on its *current* document set and marks it as the 'best' local synopsis. This periodic purging of old synopses guarantees that stale information will eventually leave the system. V has to be chosen to balance between staleness and computational cost. As a rule of thumb, a value of $V = 3N$ appears to be adequate. We intend to study the optimal choice of V in future work.

Note that global statistics (and for that matter, the searches themselves) can also be handled by centralized servers. This is, in fact, more efficient, although it introduces a single point of failure. In Section IV.E.5, we study the relative costs of the centralized and distributed solutions.

E. Search selection using global statistics

Given a set of keywords in a query and the 'best' complete synopsis, local index nodes go through the following decision process:

1. Use the title and node counter synopses to determine the set of well-replicated documents that match all the keywords in the query, and the fraction of local index nodes that have each of these titles. If the sum of these fractions, r, exceeds a threshold t, the document is popular, so flood the query with high priority. If the flood returns no results, use the DHT.
2. If *any* keyword in the query is *not* in the common keyword synopsis, use the DHT because a join is cheap if it is initiated using this keyword [3].
3. Otherwise, flood with low priority and a long time to live value.

Step 3 handles the case of an unpopular document whose title contains common keywords. In practice, this case might not occur very often, because most users realize that certain keywords are common and avoid them. Nevertheless, if it does occur, neither search algorithm is efficient: flooding might not locate the document without using a large flood depth, 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 (defined in Section III.C) and a long timeout value. The long timeout ensures that the request is flooded throughout the network. The low priority ensures that the request is handled only if there is adequate search capacity. Note that using the DHT for such queries would provide no advantage, because it would require the query to be sent to many local index nodes anyway.

F. Adaptation to prediction error

An important parameter in our system is the flooding threshold, t, defined as the threshold below which search requests are sent to the DHT instead of being flooded. Intuitively, a local index node should choose a search algorithm that maximizes a user'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. Clearly, it is optimal to choose t as the *point of indifference*, where flooding and DHT search provide equal utility.

It is hard for a system administrator to choose a threshold value that is optimal for all operating environments. Worse, the 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 peer. Therefore, the system should adjust t over time, instead of using a fixed value. Adaptive thresholding also makes the system more robust: in case of failure of the DHT, the utility from the DHT would be zero, and all queries would eventually be flooded because t would increase rapidly.

Adapting the threshold requires us to define the 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 one is just searching for

one particular document, the marginal utility per extra utility will decrease sharply beyond some point. We approximate this by choosing some maximum number of results requires, R_{max}, beyond which each further result as contribution 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. Finally, the cost of a query is its bandwidth cost B. Assuming linearity in the function, for simplicity, we can denote the utility function U as:

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

where $w_{1\text{-}3}$ are normalized weights 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 B is stored and updated in each search request and reply.

Given this utility function, we use the following algorithm at each local index node for adjusting t over time:

1. For each query, compute r, the expected number of results per node as in Step 2 in Section II.E. Assign a probability p that it will be chosen as a data point for adaptation. Because we expect most variations in t to be small, we obtain more measurements around the operating point by choosing 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 (refer to Figure 2). 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 $\binom{Q}{2}$ pairs. 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.

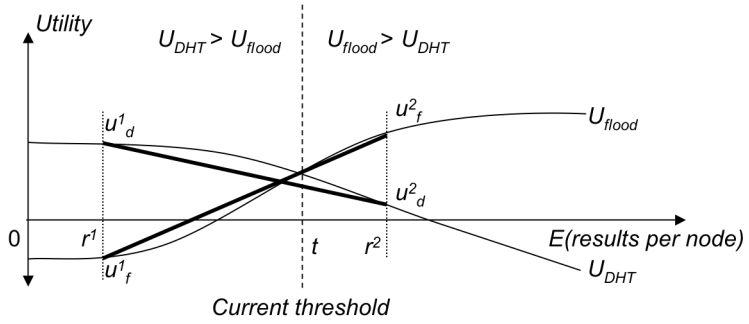

Figure 2: Adapting the flood threshold using utility functions

We find that dynamically adapting the threshold allows the system to be insensitive to the initial threshold value, and to quickly respond to changing environmental conditions. We feel that explicitly using utility functions to adapt system behavior is an interesting and widely-applicable approach in the design of highly distributed systems because it allows users to control system performance using intuitive control knobs instead of obscure tuning parameters.

III. EFFICIENT FLOODING

A. Overview

We now turn our attention to the question of flooding. Our work, though done in the context of hybrid search networks, is applicable generally to all flooding networks.

Our point of departure is to observe that flooding imposes a heavy and sometimes unnecessary burden on search networks. Recall that our search selection algorithm causes searches for popular documents to be flooded. However, flooded queries for popular documents are likely to return large result sets. For instance, in the study in Reference [1] 29% of queries in Gnutella received more than 100 results, and some queries received as many as 1500 results. However, most users require only a few results. Thus, bandwidth is wasted when flooding queries for very popular documents.

Besides network bandwidth, flooding also affects the load on the search subsystem at a local index node. Given that search, even local search, take time, excessive flooding can cause large request queues to build up at the search subsystem, causing large delays. To our knowledge, this problem has not been studied in the literature. We therefore first present a classical queueing analysis of the search load at a local index node due to flooding (Section III.B). This exposes two specific problems: congestion collapse due to search overload, and search overload due to excessive flood depth. We present an algorithm for search sub-system queue scheduling to avert congestion collapse in Section III.C and a technique to limit flood propagation that reduces search load in Section III.D. Simulations show that these improvements allow excellent performance with little overhead.

B. A queueing analysis of flooding in unstructured P2P networks

Consider a peer with d neighbors. A 1-hop flood from any of these neighbors will result in local search at this peer. Similarly, assuming no breadth-first-search back-edges, a 2-hop flood from any of its $O(d^2)$ neighbors will result in a local search on this peer. This shows that the load on the search subsystem at each peer grows nearly exponentially with flood depth. This has an unexpected side effect: as the flooding depth increases, the search load at a peer grows, increasing its search request queue length and mean search delays. In fact, some requests are delayed so long that the initiator times out and simply ignores the results. In a heavily loaded system, where service queues at a peer are almost always full, most requests that are accepted to long request queues time out, so that the search efficiency actually *diminishes* with load. This is very similar to congestion collapse, where the network fills up with retransmitted packets and the carried load diminishes with increasing offered load.

We can get a quantitative sense of the congestion collapse phenomenon by modeling the search subsystem at a peer as a classical queueing system. With any reasonable indexing scheme, a search request takes approximately constant time, so we model searches as deterministic (as opposed to Markov). We model the stream of search requests generated by an individual peer as a Poisson stream, akin to the call generation processes in a classical telephony system. Consequently, the set of search requests coming to any peer is simply a superposition of Poisson processes, which is itself Poisson (in any case, the superposition of a large number of requests from *any* generation process is well-modeled as a Poisson process [16]). Therefore, we can model the search subsystem as a M/D/1/N queue.

We are interested in the 'carried load', the number of search requests that succeed within the timeout interval, as a function of the 'offered load', the search load coming to each peer. We compute the carried load by observing that each arriving request is either rejected because the request queue is full, or accepted. If it is accepted, then it may still time out because of excessive queueing delays. Therefore, if the offered load i.e. the superposed Poisson search request arrival rate, at a peer is λ, the carried load is:

$$\textit{carried load} = \lambda * (1 - (P(\text{request is rejected}) + P(\text{request accepted}) * P(\text{service\_delay} > \text{timeout})) \quad (1)$$

Therefore, to compute the carried load, we need to compute (a) λ as a function of the per-local index search generation rate and (b) the probability that service delay exceeds the timeout value M for this value of λ.. For simplicity, we will ignore the propagation delay in the network, assuming that all delays are delays in the search queue. This is reasonable because propagation delays are on the order of a few seconds, whereas search delays can be on the order of minutes. We also assume that all search requests have the same timeout value.

We can estimate λ given the intrinsic search generation rate g, the maximum flooding depth f, the fraction of overlay edges that are *not* back edges χ, and the mean outdegree of the system, d as:

$$\lambda = gd + g\chi d^2 + g\chi^2 d^3 + \cdots + g\chi^{f-1} d^f \quad (2)$$

For an M/D/1/N server with deterministic service time T, the load factor ρ is given by λT, and the probability that there are j elements in the queue, corresponding to a queueing delay of jT, is given by [17]:

$$P_o(N) = \frac{1}{1+\rho b_{N-1}}; P_N(N) = 1 - \frac{b_{N-1}}{1+\rho b_{N-1}}$$

$$P_j(N) = \frac{b_j - b_{j-1}}{1+\rho b_{N-1}} \quad 1 \le j < N$$

where

$$b_0 = a_0; \quad b_j = \sum_{i=0}^{j} a_i$$

and

$$a_0 = 1; \; a_i = e^{\rho}(a_{i-1} - \sum_{k=1}^{i-1} \alpha_k a_{i-k} - \alpha_{i-1}) \quad \text{for } 1 \le i < N$$

where

$$\alpha_k = \frac{\rho^k}{k!} e^{-\rho}$$

For a timeout value of M, the probability that the queueing delay exceeds M can be approximated by

$$P(delay > M) = \sum_{i=\left\lfloor \frac{M}{T} \right\rfloor}^{N} P_i(N) \tag{3}$$

The probability that the system is full (because Poisson arrivals always see time averages) is simply $P_N(N)$, so, we obtain the carried load from (1) and (3) as:

$$carried\ load = \lambda(1 - (P_N(N) + (1 - P_N(N))(\sum_{i=\left\lfloor \frac{M}{T} \right\rfloor}^{N} P_i(N)))$$

This allows us to compute Figure 3 that shows the offered load and carried load as a function of the request generation rate for M = 20s, f = 5 hops, χ = 0.4, d = 7, T = 250ms and various values of N the request queue length. (These values are chosen for illustrative purposes only and are not necessarily representative of a real system.)

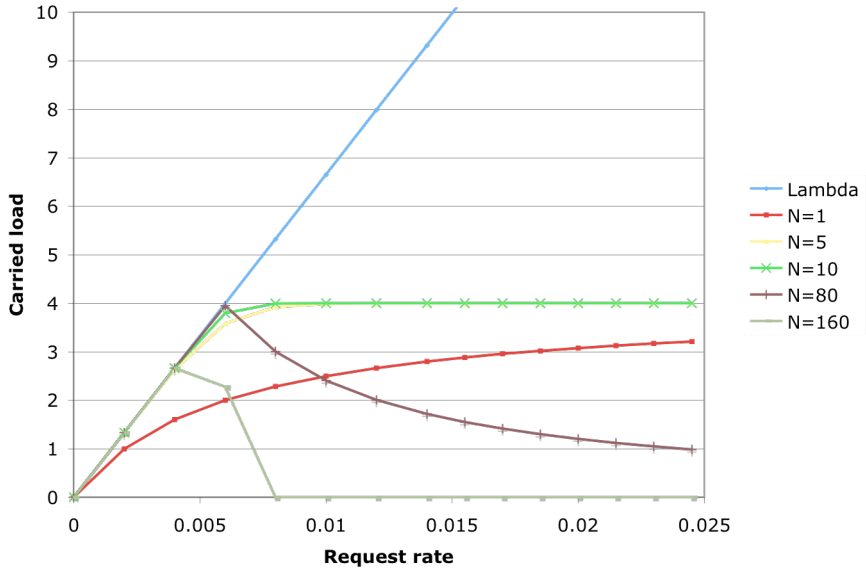

Figure 3: Carried load vs. Request rate, varying N

We see that the offered load λ increases linearly with the request rate g. The carried load tracks the offered load for small values of g for all values of N. However, it undershoots λ for small N and drops off sharply with increasing g for large values of N. The dependency of the carried load on N can be understood as follows:

- When N is small (<5), as g increases, $P_N(N)$ increases, so more requests are dropped and the carried load asymptotically approaches the service rate (4.0 in the figure).
- When N is not too large ($5 \le \left\lfloor \frac{M}{T} \right\rfloor \le N$), even when g is large, $P_N(N)$ is not too large, so the asymptote is quickly reached. However, note that when g is large enough that λ exceeds 4, the system saturates.
- When N is larger than $\lfloor M/T \rfloor$, at high load, all requests are accepted, but many time out. Indeed, due to this effect, the actual carried load is *lower* for large N than for small N. This leads to the paradox where a system administrator who is trying to improve system performance by adding more memory and increasing the request queue size can actually make matters worse!

This analysis motivates two methods to improve flooding performance. Recall that the carried load is less than the offered load due to three reasons: (a) some requests get dropped due to insufficient queue length (b) some requests wait in queue for so long that they time out and (c) because the offered load exceeds the request service rate (in which case any finite length queue will eventually fill up).

The first factor is not very important: as Figure 3 shows, merely increasing the queue length from 1 to 10 allows almost all incoming requests to be queued even at high load. The second factor is quite important: if the queue is too long, requests in the queue that are too delayed to be useful nevertheless get serviced, inconveniencing other requests. This suggests that a node should *pre-emptively* discard requests that will time out, in order to make way for requests that arrive later. We discuss this in Section III.C. The third factor--whether the offered load exceeds the service capacity--is also important, and we discuss this in more detail in Section III.D.

C. Adaptive and pre-emptive request dropping

To avoid congestion collapse, a node has to maintain the invariant that no request that has the potential to be served before its timeout is ever dropped due to service of a request that does in fact time out. This guarantees that the search subsystem always makes progress. To maintain this invariant, the initiator of search request sets a 'timeout' value in the request. A node drops an incoming request if it predicts that the request cannot be serviced before its timeout.

The simplest way to achieve the invariant is to keep the queue in FIFO order and simply skip over a request if the current time exceeds the timeout. While correct, this may result in some requests being served well before their timeout, while other requests, with short timeouts, arriving to a long queue, get dropped. If we rearrange such requests, a node can serve more requests than with strict FIFO ordering, while still maintaining the invariant. Specifically, we would like to arrange that if request B arrives after request A, but with a shorter timeout, the node should service B before A as long as this will not cause A to time out. Note that this preserves the invariant. This also suggests that the right way to organize the request queue is as a priority queue.

Several priority queues have been studied in the literature. We choose a variant of a calendar queue [18] originally presented in [19] to implement non-work-conserving scheduling. In this scheme, time is bucketed, and all requests with timeouts in the same bucket ('day') are linked together. The node measures the mean time it takes to service each search request and ensures that the time taken to serve all requests in a day, i.e. the product of the service time and the maximum number of requests in a bucket, does not exceed a day's duration. With this data structure, requests can enqueued and dequeued in O(1) time.

Unlike the scheme in [19], the priority queue in our scheme is work conserving, that is, if there aren't enough tasks in a day, the scheduler moves on to the next day. This introduces the problem that although a 'day' bucket may be full, it may start service earlier than expected, so that requests arriving to a 'full' day bucket may be unnecessarily dropped. To deal with this, we extend the scheme in [19] to maintain a guaranteed (high priority) and a best effort (low priority) linked list per 'day'. The guaranteed list has a fixed length maximum. Requests that arrive to a full guaranteed list (and all low-priority search requests) are placed in the low priority queue. Low

priority requests are served if and only if serving them does not cause the start time of the next day to be delayed: low priority requests not served by the end of a day are dropped.

An important parameter in this system is the time taken to serve one request. This determines the number of service requests that can be stored in a guaranteed service list of a bucket. A node must continually update the value based on the current request stream. Because the system is insensitive to over-estimates of this value, and to account for variations in the service rate, we estimate this as the mean service rate + three times the standard deviation as measured over a moving time window.

If a new service time estimate is lower than an older estimate, then the queue invariant continues to hold for all accepted requests, because the only effect is to increase the maximum size of the 'guaranteed' list in each day's bucket. However, if a new service time estimate is higher, then some existing 'guaranteed' service requests may not be serviced in time. We need to renormalize the queue after such an increased estimate. We do so by first re-computing the new maximum number of guaranteed requests in a day. Service requests are re-inserted into the guaranteed service queue in order of their arrival time, up to the new maximum. Any overflow requests are moved to the low priority queue. This may result in some formerly guaranteed requests to move to a best-effort list (though no requests move from one day to the next, because the length of the day does not change). Note that the queue invariant is preserved after the renormalization. The renormalization cost is proportional to the number of entries in the guaranteed service list. This can be reduced by carrying out renormalization only when a search request in a guaranteed service list times out, indicating that the current estimate of service rate is stale.

This priority queue allows a node to carry out insertions and deletions in O(1) time. Moreover, it is easy to see that a service request is never dropped unless it can be served *only* at the expense of another request that can also be served within its timeout period. This shows that the scheme is both correct, in that it maintains the desired invariant, and also optimal, in that no other scheduling discipline that maintains the invariant could possible serve any more requests. We demonstrate the benefit of the priority queue through simulations in Section IV.D.

D. Adaptive flooding

We now turn our attention to reducing the flooding load. We can gauge the need for adaptive flooding by looking at the sensitivity of λ to changes in the service generation rate g, the flooding depth f, the fraction of overlay edges that are *not* back edges χ, and the mean outdegree of the system, d. Taking partial derivatives with respect to g, f, χ, d :

$$\frac{\partial\lambda}{\partial g} = O(1); \quad \frac{\partial\lambda}{\partial f} = O(d^f); \quad \frac{\partial\lambda}{\partial \chi} = O(\chi^{f-2}); \quad \frac{\partial\lambda}{\partial d} = O(d^{f-1})$$

we see that f plays a critical role in determining λ, exponentially influencing the gradient with respect to three variables. To illustrate this, we choose the same operating point as before, i.e. $f = 5$, $\chi = 0.4$, $d = 7$, and set g to 0.0075 and study the sensitivity of λ to variations in these values:

Vary g	λ	*Vary* f	λ	*Vary* d	λ	*Vary* χ	λ
0.005	3.33	3	0.6	5	1.16	0.2	0.57
0.00625	4.16	4	1.76	6	2.52	0.3	1.90
0.0075	**4.99**	**5**	**4.99**	**7**	**4.99**	**0.4**	**4.99**
0.0875	5.82	6	14.03	8	9.12	0.5	11.01
0.10	6.65	7	39.3	9	15.7	0.6	21.4

It is evident from the table that, at this operating point, variations in *f, d,* and χ all influence λ. However, χ and d are intrinsic properties of the overlay network topology and are chosen for robustness as well as for search efficiency. So, they are relatively hard to vary. In contrast, f is both easy to modify and a significant determinant of λ. This motivates us to study techniques to reduce the flooding depth in an attempt to reduce λ.

A simple way to reduce λ is by using a fixed and small maximum flood depth. However, choosing too small a depth results in poor recall of unpopular documents; with a *fixed* flood depth, there is a tradeoff between the flood depth and the degree of recall.

Gnutella addresses this issue by using *dynamic querying* [20], where a query is re-flooded with a increasingly larger flood depths if fewer than 150 responses are returned. This approach addresses recall, but at the cost of query bandwidth and an increase in the response time.

Our approach instead is to use an *adaptive* flooding algorithm. Essentially, a node *predicts* how many results are being found in parallel by its peers based on a *measurement* of its depth in flooding tree and the average number of results in the path so far. If this exceeds the needed number of results, the node *adapts* by stopping flooding. Our algorithm avoids re-flooding, so it generates less traffic and has shorter response times than dynamic querying.

To predict the expected number of results, we assume that the network of peers is *isotropic*, that is, the distribution of documents matching a particular keyword is identical in every subset of peers. This is similar to the ergodicity assumption in queueing systems. Although this is a strong assumption, it considerably simplifies the mathematical model. Simulations confirm that this assumption appears to be reasonable.

We now present the prediction model. Consider a flood tree where every node has degree d and a node at depth D in it. D. Let $r_1,\ldots,r_D$ be the numbers of results found at each peer in the path where the root of the flood tree is assigned index 1 and the node has index D, and let $e_1,\ldots,e_D$ be the numbers of documents found at each peer. As before let the fraction of overlay edges that are *not* back edges be χ. Finally, let M be the number of results desired.

To implement this algorithm, a query carries with it a stack of *measured e, r,* and d values as well as χ, set by the query initiator, as a tuning parameter. Then a node:

1. Calculates the average number of documents at a peer, $\bar{e} = \sum_{i=1}^{D} e_i / D$
2. Calculates the mean number of results per document $\bar{r} = \sum_{i=1}^{D} r_i / \sum_{i=1}^{D} e_i$
3. Estimates the number of peers at level i: $p_i = \chi^{i-1} d^i$
4. Estimates the total number of peers answering the query: $p = 1 + \sum_{i=1}^{D-1} \chi^{i-1} d^i$
5. *Predicts* the number of results found by all peers in parallel, $R = \bar{r} \cdot \bar{e} \cdot p$
6. If $R > M$, or the query's timeout value has already been exceeded do not flood further. Else, *adapt* the flood by forwarding the query to no more than $(M - R)/(p \cdot \bar{e} \cdot \bar{r})$ neighbors. If this is smaller than the actual number of neighbors, flood to the least loaded neighbors as in [21].

If the actual number of results returned to an initiator exceeds (or is smaller than) M, the initiating node can reduce (or increase) χ in future queries. We plan to study this additional degree of adaptation in future work. Note that adaptive flooding is also applicable in a random-walk flooding network, like [21]: R is then known precisely, and can be passed down along the search path.

IV. Simulations

We built a discrete event simulator to study the performance of our algorithms. The following sections present its design, optimizations we applied to increase its efficiency, and simulation results.

A. Simulator design

The simulation system, written in Java, consists of two modules: Simulator, which carries out an event-driven simulation and outputs the results, and Analyzer, which uploads the results to a MySQL database and generates reports and Gnuplot commands for graphs. Three design decisions proved to be very useful in practice:

- All simulation arguments are read from a file in a simple key-value text format. Such files can also "#include" each other. This eases comparison of parameter combinations.

- The Analyzer program is scriptable, allowing us to run long simulations without intervention.

- Storing results in a database gave us considerable flexibility in comparing different runs, or even creating new reports and graphs.

The simulator is highly extensible. The Simulator class contains only the necessities for an event-driven peer-to-peer system simulator and all other functionality is delegated to other classes. Four interfaces can be implemented by other classes to let them take various roles in the system:

1. `Topology` – Simulates a P2P system architecture, including search algorithms and behavior on node join and leave events.
2. `UserGenerator` – Generates user join, leave, and query events.
3. `DelayGenerator` – Simulates an underlying network by generating delays for messages of different sizes.
4. `DocumentGenerator` – Provides the set of documents and keywords.

We simulate short-term such as hops in the propagation of a single query, as well as long-term effects like node joins and leaves and the slowing of request rates as nodes age, observed in [15]. To save time and because DHT performance is well-studied, we do not simulate the detailed 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 simulate querying each local index node in the result set returned by the DHT.

B. Optimizations

We used a number of optimizations to make the simulator several orders of magnitude more efficient than the original implementation:

- Fast I/O routines. We found that logging an event sometimes takes longer than actually simulating it, because Java creates temporary objects during string concatenation. We corrected this by using a custom, large StringBuffer for string concatenation. We also developed custom number-output routines. For instance, a simple fixed-point method for printing floating point numbers was about 10 times faster than Java's Double.toString, because the latter tries to "pretty-print" the output. Our final logging code was about 10 times faster than our original, naïve, and this has a significant impact on simulation speed.
- Batch database uploads. Even prepared statements turn out to be much less efficient in updating a database than populating a table from a tab-separated text file using MySQL's `LOAD DATA INFILE` command.
- Avoiding keyword search for exact queries If we know a query to be exact, we match queries using an index on document IDs. This made the simulator about 10 times faster on synthetic runs. This technique could also be used for inexact queries if we were to pre-calculate the set of document ID's matching each possible query. However, we have not yet implemented this optimization.

In general, we found that the key considerations for optimizing simulator performance are (a) to use a profiler, because it may not be obvious where the slowdowns are occurring and (b) to simplify algorithms by taking advantage of the fact that the simulator is "omniscient".

C. Parameter values

We used the following values for simulation parameters for all the simulation results reported here, unless otherwise specified:

Argument	**Base Value**
Total simulation time	80,000 s (22 hours)
Number of distinct documents	4,500
Zipf parameter for document popularity	1.0
Mean end node arrival interval	0.7s
End node lifetime distribution	The distribution in [6] for Gnutella hosts
End node bandwidth distribution	The distribution in [6] for Gnutella hosts
Average inter-query interval	300 s
Average *initial* number of documents per end node (this grows over time)	20
Fraction of nodes promoted to local index	5%
Local index node degree	3
Maximum results desired	25

Number of local index nodes per end node (replication factor)	2

We use randomly generated user lifetimes and bandwidths, based on the distributions observed in [6]. We generate queries and document/keyword sets in two different ways:

- *Trace-driven*: We play back partial keyword searches from the Gnutella data set in [2] for documents identified by simultaneously tracing 50 Gnutella ultrapeers as described in Section IV.I.

- *Synthetic*: We generate random exact search requests according to the fetch-at-most-once model in [15] (Zipfian popularity distribution, but no user requests the same document more than once). We only generate exact queries, since it is difficult to generate realistic inexact queries.

Synthetic workloads allow us to easily vary parameters such as the end-node arrival rate, the number of documents in the system, and the inter-query time. Therefore, we use the synthetic workload for comparing the different search algorithms in Sections IV.C-IV.F, and the trace-driven workload only for validation, in Section IV.G.

D. Simulation stability

With the parameters shown above, there were 500 local index nodes, 100 global index nodes, and we simulated about 1.7 million queries over a 22 hour period. We observed that a stable online population of about 10,000 active end nodes was achieved after about 20,000 simulated seconds (~6 hours). Therefore, results are presented only the queries made between 40,000 and 80,000 seconds. Note that the total population is actually 91,000 end nodes – the number of active nodes is smaller because we simulate end nodes entering and leaving the system.

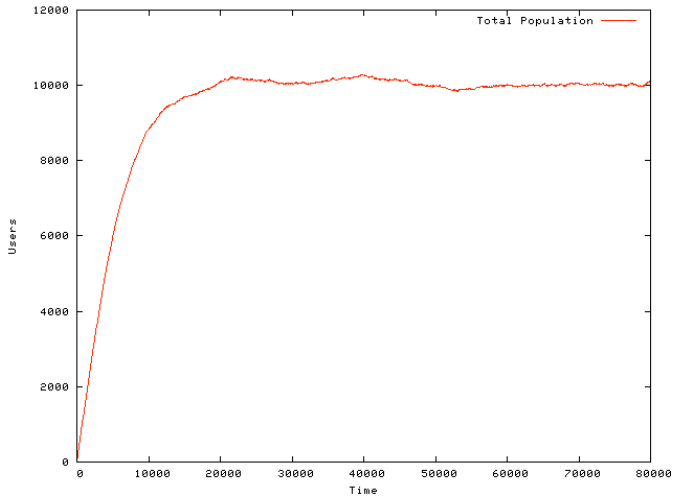

Figure 4: Active population size vs. Time

E. Search algorithm comparison

In this section, we compare our search algorithms with the following seven search techniques in an underloaded environment, which is assured by setting the service time at the search subsystem to zero:

Name	**Description**
Flooding (Depth X)	Flooding among local index nodes with fixed flood depth of X
Dynamic Flood	Initial flood to depth 2, re-flooding to depth 4 if too few results are received in 2.0s
Adaptive Flooding	Adaptive flooding up to depth 4
Pure DHT	All queries looked up in a DHT using the adaptive join method of [3]
Simple Hybrid	Similar system to [2]: queries are first flooded to depth 2 or 3, then looked up in a DHT if fewer than 10 results are received from the flood.

Our System	Adaptive thresholding with adaptive flooding to depth 4 for floods
Central Server	An ideal central server with zero request service time.

Note that we compare our algorithms not only with flooding with a fixed flooding depth, but also with Gnutella-style dynamic querying, a pure DHT search, a variant of the hybrid search proposed in [1,2], and, as a straw man, a central server. Unlike the scheme in [2] where the DHT is queried if no results are received after 30 seconds, we send a query to the DHT if fewer than 10 results are received in two seconds (which is sufficient time for a depth-3 flood). We made these changes to reflect the lower network delays in our simulation. We now analyze the results for each approach:.

We use the following performance metrics:

Name	Meaning
Recall	Percentage of queries that found a matching document, given that at least one available document that matches 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.

We report standard deviations in parentheses, where these deviations are obtained from four consecutive 10,000 second intervals of the simulation from time 40,000 to time 80,000. Most standard deviations are very small, but we show them for the sake of completeness.

1. Flooding approaches compared

System	Recall	Results	FRT	LRT	BWC
Flood (Depth 1)	56.8% (0.1%)	9.2 (0.0)	1.21 (0.0)	1.21 (0.0)	0.12 (0.0)
Flood (Depth 2)	74.4% (0.1%)	28.1 (0.1)	1.36 (0.0)	1.74 (0.0)	0.50 (0.0)
Flood (Depth 3)	87.2% (0.1%)	74.5 (0.2)	1.52 (0.0)	2.18 (0.0)	1.56 (0.0)
Flood (Depth 4)	95.6% (0.0%)	187.1 (0.7)	1.65 (0.0)	2.56 (0.0)	4.42 (0.0)
Flood (Depth 5)	99.1% (0.0%)	420.6 (3.7)	1.72 (0.0)	2.87 (0.0)	10.90 (0.1)
Dynamic Flood	95.6% (0.1%)	33.3 (0.1)	2.09 (0.0)	3.72 (0.0)	1.21 (0.0)
Adaptive Flood	95.5% (0.1%)	28.2 (0.1)	1.65 (0.0)	2.43 (0.0)	0.94 (0.0)

Flooding, even at depth 1, has a remarkably high success rate. This is because each local index node indexes about 20 end nodes, so even a depth 1 flood reaches about 80 end nodes, or 1600 instances of documents. As the depth increases, the success rate increases. However, we observe diminishing returns: few queries will be for rare enough documents that they benefit from an increase in depth. Furthermore, even at depth 5, the recall rate is noticeably less than that of pure DHT or the hybrid approaches (there are about 9 times as many unsuccessful queries even when where there was a document available). As expected, as depth increases, the bandwidth cost increases exponentially and last response time increases because we must flood to a greater depth. The increase in first response time is due to queries that were unsuccessful at lower depths finding results (albeit at a later time) at higher depths. The decline in recall with flood depth does not show up here because we kept the search time artificially small, so no search request queue was overloaded. The effect of congestion is studied below in Section IV.D.

2. Dynamic and Adaptive Flooding

We now compare dynamic flooding and adaptive flooding with flooding with a fixed depth 4 (both the dynamic flooding and adaptive flooding also had a *maximum* depth of 4). Both self-limiting algorithms preserve the depth-4 recall rate, about 95%, because they can reach the same number of nodes as fixed-depth flooding if they need to. They also return the number of results much closer to the desired value of 25 than a fixed depth flood.

As expected, dynamic flooding has a significantly lower bandwidth cost (3.7 times) than depth-4 fixed-depth flooding, because many queries receive enough matches after 1 or 2 levels of flooding (in fact, the bandwidth cost is even lower than depth 3 flooding, showing that more than half the queries find enough matches within 2 levels). However, dynamic flooding has noticeably *higher* first and last response times, because for those queries that do

not find enough results in a depth 2 flood, we must wait for some time (2 seconds in our implementation, which was chosen to be higher than the LRT for depth 2 flooding) to obtain the results and then re-flood.

Adaptive flooding makes two distinct improvements over dynamic flooding: lower bandwidth cost (22% lower), and lower last response time (35% lower). This is because dynamic flooding repeats work as it re-floods through nodes at depths 1 and 2 and loses some time waiting for results from the depth 1 and 2 floods. It is also comforting to observe that despite the limited information available and the fairly strong isotropicity assumption, adaptive flooding not only matches the recall rate of fixed-depth flooding at depth 4 but also has only slightly more results on average than the desired number (28.2 vs. 25).

3. Pure DHT vs. Flooding

System	Recall	Results	FRT	LRT	BWC
Flood (Depth 3)	87.2% (0.1%)	74.5 (0.2)	1.52 (0.0)	2.18 (0.0)	1.56 (0.0)
Flood (Depth 4)	95.6% (0.0%)	187.1 (0.7)	1.65 (0.0)	2.56 (0.0)	4.42 (0.0)
Dynamic Flood	95.6% (0.1%)	33.3 (0.1)	2.09 (0.0)	3.72 (0.0)	1.21 (0.0)
Adaptive Flood	95.5% (0.1%)	28.2 (0.1)	1.65 (0.0)	2.43 (0.0)	0.94 (0.0)
Pure DHT	99.8% (0.0%)	134.7 (0.8)	6.01 (0.0)	6.02 (0.0)	3.46 (0.0)

As expected, the pure DHT approach has shows one major advantage over flooding: (nearly) perfect recall. This is because with a DHT (nearly) every document will be found. The missing 0.2% in recall is due to race conditions such as an end node disconnecting while a query is in progress, causing a document that was available when a query is started to no longer be available. Another race condition is when multiple local index nodes disconnect in quick succession, causing some end nodes to be "stranded" and not indexed in the DHT although they have files matching the query.

Despite the advantage of perfect recall, the DHT approach has 2.5 times higher last response time, 3.7 times higher bandwidth cost than adaptive flooding, and, quite surprisingly, 2.2 times the bandwidth cost of fixed-depth 3 flooding. Note also that the number of results from the DHT, though higher than desired, is still significantly *lower* than it would be without adaptive joins as in [3]. Though not shown here, we found that without adaptive joins, the pure DHT approach returns vastly results than desired and its bandwidth cost is higher than even fixed-depth 5 flooding.

4. Hybrid approaches compared

System	Recall	Results	FRT	LRT	BWC
Pure DHT	99.8% (0.0%)	134.7 (0.8)	6.01 (0.0)	6.02 (0.0)	3.46 (0.0)
Simple Hybrid (Depth 2)	99.9% (0.0%)	63.0 (0.2)	2.93 (0.0)	6.06 (0.0)	1.91 (0.0)
Our System	99.9% (0.0%)	28.5 (0.1)	2.22 (0.0)	2.99 (0.0)	0.98 (0.0)

By combining flooding with DHT search, the simple hybrid approach of [1,2] when compared to a pure DHT, provides a 61% drop in first response time, 8% drop in average last response time (even though the LRT for queries sent to the DHT are higher because a flood is performed first), and 34% drop in bandwidth cost (because even flooding with depth 3 uses less than half the bandwidth of a DHT search, as seen above). This clearly validates the hybrid approach and demonstrates its benefits over a pure DHT approach. Nevertheless, note that the simple hybrid returns, on average 3.2 times as many results as desired. Moreover, its FRT is actually *higher* than a fixed-depth flood because the delays experienced by searches for rare documents pull out the tail of the delay distribution.

Our system improves these results even further with respect to both response time and bandwidth cost. The first response time is only slightly lower, because most queries are for popular documents that both systems will flood. However, our last response time is 51% lower because queries for known rare documents are sent directly to the DHT rather than being "tested" using a flood. Moreover, bandwidth costs are 1.9 times lower because we save doing a depth-2 flood for queries that are sent directly to the DHT. This validates the gain in performance by the use of global statistics. Meanwhile, the recall rate is still (nearly) perfect, and the number of results is only slightly above the desired number, because both the search algorithms we use (adaptive flooding and DHT) are careful to not return too many unnecessary results.

5. Centralized approach compared to our system

System	Recall	Results	FRT	LRT	BWC
Our System	99.9% (0.0%)	28.5 (0.1)	2.22 (0.0)	2.99 (0.0)	0.98 (0.0)
Central Server	100.0% (0.0%)	22.6 (0.0)	1.21 (0.0)	1.21 (0.0)	0.45 (0.0)

We noted in Section II.D that a centralized approach is optimal in terms of performance, though it lacks scalability and robustness. We now compare our system with a hypothetical centralized server in order to quantify the cost of decentralization.

We find that a central server has perfect recall, 45% lower FRT and 59% lower LRT than even our system. Moreover, the bandwidth cost is also lowered by 54%. We conclude that the price to pay for decentralization is a doubling of every performance metric. Note that the number of results returned by the centralized system is smaller than 25 on average, because it never returns more than 25 results, but will return fewer if only fewer are available.

F. Effect of preemptive dropping

System	Recall	Results
Flooding (fixed depth 4)	70.4% (0.7%)	68.2 (0.1)
Flooding with Preemptive Drop	91.8% (0.4%)	131.7 (0.1)
Adaptive Flooding with Preemptive Drop	94.1% (0.2%)	27.0 (0.0)
Upper Bound	95.6% (0.0%)	187.1 (0.7)

We now focus on the congestion collapse problem and its solution outlined in Section III.C. Recall that the collapse problem occurs in a system with long queues, deep flooding and heavy load. We set the queue length to be unbounded, the maximum flood depth to 4, and artificially reduce the search subsystem service rate by setting the per-request service time to 200 ms. We then compare the recall and number of results returned by three flood algorithms: a naïve flood with fixed flood depth, flooding with fixed depth but preemptive dropping, and adaptive flooding with preemptive dropping. To judge these algorithms, we obtained an upper bound on recall by flooding to depth 4 on an infinitely fast search subsystem (which is almost identical to Flooding (Depth 4) in IV.C.1).

In terms of recall, pure flooding does significantly worse than the theoretical maximum, failing on about 6.7 times more queries. This is because queries that time out at an early flood depth due to large queueing delays fail to find results. Preemptive drop provides a large improvement, failing on 3.5 times fewer queries than fixed-depth flooding. However, even with preemptive dropping, some queries are flooded to an unnecessarily large depth, which can reduce service to other queries. We see this by noting that the number of results returned, 131.7, far exceeds the desired 25. Adaptive flooding reduces the percentage of failed queries by an additional factor of 1.4, because it terminates queries that have found enough results, leaving room for others. In fact, it only has about 30% times more misses than the upper bound, although it is working on a heavily loaded system with an artificially reduced request service rate.

In terms of number of results, both flooding algorithms find fewer results than are available on average within a depth of 4 (i.e. 187.1). However, adding preemptive drops helps each *successful* query find about 1.9 times more results (131.7 vs. 68.2), meaning it's reaching about 1.9 times more nodes. Note that even in this very uncertain environment, adaptive flooding has slightly more than 25 results on average, as required.

G. Effect of Utility Function

# Results Desired	Recall	Results	FRT	LRT	BWC
$R_s = 25$	99.9% (0.0%)	28.5 (0.1)	2.22 (0.0)	2.99 (0.0)	0.98 (0.0)
$R_s = 50$	99.9% (0.0%)	36.0 (0.1)	2.21 (0.0)	4.59 (0.0)	1.42 (0.0)

We now turn our attention to the ability of users to control system behavior with easy-to-understand 'knobs'. We simulated two systems that differed in only one respect: in the first system, the desired number of results is 25, and in the second, the desired number is 50. The table above shows system behavior.

Note that the second system delivers about 26% more results than the first one with nearly the same first response time, but higher last response time and bandwidth cost. Essentially, the additional results are obtained by either a

deeper flood (for popular requests) or longer in-network joins (for unpopular requests). Although the system did not quite reach 50 results, mainly because most queries in the simulation didn't have that many matching documents, we see that simply changing one easily understandable parameter is sufficient to demonstrably alter system behavior. This is because of the explicit use of utility functions in adapting system behavior.

H. System Scalability

We tested the scalability of our system by varying the number of end nodes. We ran tests with three different end node arrival intervals (i.e. the mean time between the arrival of an end node into the system): 1.0s, 0.4s, and 0.3s. Because the node lifetime distribution is fixed, this increases the 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 populations end node populations of roughly ten times this size.

Popl'n	Recall	Results	FRT	LRT	BWC
7000	99.9% (0.0%)	27.4 (0.1)	2.14 (0.0)	3.66 (0.0)	1.00 (0.0)
17,500	99.9% (0.0%)	30.7 (0.0)	2.23 (0.0)	3.29 (0.0)	1.06 (0.0)
23,300	99.9% (0.0%)	31.3 (0.0)	2.23 (0.0)	3.31 (0.0)	1.08 (0.0)

First, note that as the population more than triples, the recall remains the same, the number of results increases by 9%, the first response time increases by 4%, the last response time decreases by 9.5% and the bandwidth cost increases by 8%. The number of results increases with population size (though it still remains fairly close to the 25 desired) because many smallest-possible (depth-1) floods find more than 25 results. This is unavoidable in any system that relies on flooding.

The first response times increase very slightly; this can be attributed to slightly larger DHT indices. 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 sufficient numbers of results are found with shallower floods. Finally, bandwidth costs increase slightly, 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.

Intuitively, our system scales well because DHTs costs increase logarithmically with system size, and flooding costs, for a fixed flood depth, are constant. This intuition is validated through our simulations. We note that the use of adaptive flooding and adaptive join for DHT searches [3] is critical: otherwise, unnecessarily many results would be returned in large networks, needlessly increasing response times and bandwidth costs.

I. Trace-based simulations

System	Recall	Results	FRT	LRT	BWC
Dynamic flood	81.0% (0.4%)	12.8 (1.0)	2.57 (0.0)	3.71 (0.0)	0.83 (0.0)
Simple Hybrid (Depth2)	99.9% (0.0%)	16.7 (1.0)	3.51 (0.0)	4.68 (0.0)	1.65 (0.0)
Our System	99.9% (0.0%)	12.3 (0.8)	2.44 (0.0)	2.88 (0.0)	1.26 (0.0)

To validate the conclusions from synthetic-workload based simulation, we re-ran several of our experiments on a trace-based workload. The traces use the Planetlab-based monitoring infrastructure described in [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 9.24 million queries, 199,516 distinct keywords and 672,295 distinct documents. Note that the trace-driven simulation underestimates the actual set of documents available in the system, since can only identify documents that matched some query. We compensate for this by adding 100 randomly generated additional documents with these keywords to each end node. Because most queries only matched about 10 documents, we set the max results desired to 10.

Our system has a lower FRT and LRT than simple hybrid, because it decides before sending each query whether to use the DHT. The response times are also lower than those of dynamic flooding because in this simulation we have a large number of documents so most documents cannot be found in one or two flood levels. Our system also has lower bandwidth cost than simple hybrid, but higher than dynamic flood because we must use the DHT very frequently in these simulations since there are more documents. Overall, however, these results are consistent with what we have shown with synthetic workloads in IV.C.3 and IV.C.4 .

J. Synopsis convergence time

We claimed in Section II.D that our synopsis aggregation algorithm converged in exponential time. We illustrate this behavior by studying the number of time steps it took for each of ~500 local index nodes to fuse

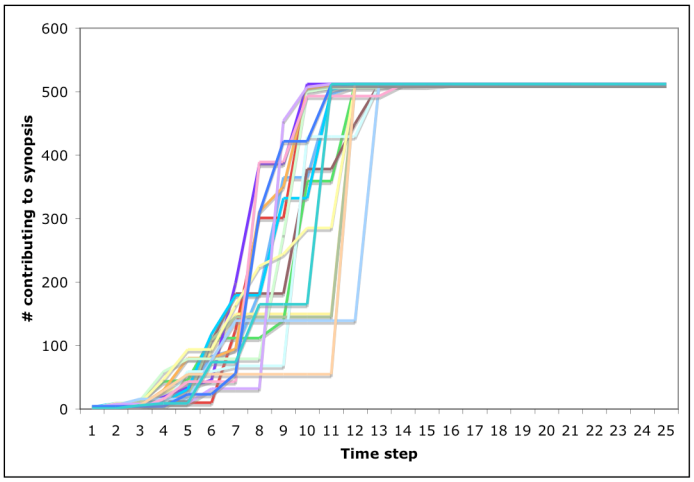

Figure 4: Synopsis convergence time

synopses generated by all the other nodes. Figure 4 shows time steps on the X axis, and the number of contributing nodes on the Y axis, for synopses arriving at a randomly chosen sample of 15 nodes. We see in Figure 4 that all 15 nodes in the sample are able to fuse synopses from all other nodes within 14 time steps, which is roughly logarithmic in the number of nodes, as claimed.

V. Related Work

A. Peer-to-Peer Search Systems

There are several peer-to-peer file-sharing/search systems that are either in use today or have been proposed in the literature. These fall roughly into three classes (a) unstructured flooding networks, such as the Gnutella network [22], and the FastTrack network originated by Kazaa [23] (b) structured DHT-based networks, such as those discussed in [4] and (c) hybrid networks that combine elements of both networks [1,2].

Unstructured flooding networks allow free-text search, because each node knows the full titles of its documents and can check whether they match the query. However, as we have shown, flooding can be expensive both in terms of network bandwidth and CPU cost. Several optimizations have been proposed to improve flooding network performance. One is to designate some peers as 'server-like' longer-lived supernodes. Queries are only flooded between supernodes, so the flood depth can be smaller. A second optimization is "dynamic querying" [20], where queries are re-flooded with increasing depth if too few results are found. The Gia system [21] introduces several other optimizations such as topology adaptation, one-hop replication of users' title lists, and random walks biased towards nodes with many neighbors. However, even with these optimizations, no flooding system can efficiently find rare documents. For example, reference [1] shows that, in Gnutella, 18% of queries return no results even though there are results available in the system in at least two-thirds of these cases.

Numerous DHT-based search systems have been proposed in the literature; an overview of these can be found in [4]. Ref. [24] presents several optimizations that can make even peer-to-peer web search feasible. DHT-based systems typically do not support free-text search. Extensions to DHTs to allow this have been proposed in [3] and [5], among others. However, approaches that use DHTs alone cannot leverage the simplicity and efficiency of flooding networks for popular documents.

Hybrid systems combine DHT and flooding networks to get the best of both worlds [1,2]. These systems first flood a query, then send it to a DHT-based search engine (PIER [25]) if no results are found. Our system makes several improvements over this design: (a) global statistics to select the search algorithm, (b) a distributed aggregate query technique to collect these statistics, (c) adaptation of the flooding threshold, (d) low priority flooding for unpopular documents with popular keywords, (e) pre-emptive dropping of queries to avert congestion collapse, and (f) adaptive flooding to reduce unnecessary search results.

B. Adaptive systems

There is an enormous literature on classical adaptive control systems that is summarized in Reference [26]. In these systems, the mathematical system model (for example, the transfer function) is precise enough to derive bounds on stability and convergence time. A typical distributed system, however, is complex enough that it is difficult, if not impossible, to use classical modeling techniques. Nevertheless, the fundamental insight in feedback control, as expressed by the 'predict-measure-adapt' pattern, still applies. This has also been observed, for, example, in Oceanstore [27] where the pattern is called 'The Cycle of Introspection'. Our collection and dissemination of synopses is similar in principle to their proposal for a common architecture for introspective systems. Adaptive control is also the basis for a recent proposal to build large-scale adaptive peer-to-peer systems using techniques from genetic algorithms and complex adaptive systems [28]. We expect adaptive behavior to be the cornerstone of the next generation of distributed and P2P systems.

C. Queueing analysis of Peer-to-peer systems

To our knowledge, the only other work on queueing analysis of a P2P system is by Ge et al [29]. In their work, the set of peers who store a particular document are treated as a single queuing server with a service rate proportional to the cardinality of the set. In contrast, we not only study the search load on a single peer due to deep flooding to mathematically quantify the congestion collapse phenomenon but also propose scheduling and adaptive flooding techniques to avert this collapse.

D. Randomized gossip and synopsis computation

Gossip systems are well known in the literature, where they are also called epidemic algorithms [30]. Recently there has been renewed interest in gossip algorithms for in-network processing of aggregate queries [8,31,32], particularly for sensor networks. Gossip is particularly well-suited to disseminate synopses (or sketches) that describe partial results of an aggregate query. Recently, Nath et al [7] introduced the class of Order and Duplicate insensitive synopses that leverage the approximate counting techniques pioneered by Flajolet and Martin [10]. Similar counting techniques have also been proposed by Consindine et al [33].

How fast do aggregate queries converge? Kempe et al have shown exponentially fast convergence for a class of synopses called *linear* synopses [8]. Boyd et al have also shown that a Semi-Definite Program approach can be used to construct optimal randomized gossip protocols [9].

Our synopsis generation algorithm is a variant of the 'Most Popular Item' query in [7]. However, unlike this work, we use randomized gossip for synopsis dissemination. Although this results in a non-linear query, so that the convergence proof in [8] does not apply, we find that we can achieve exponential convergence in practice. Note that our gossip algorithm also takes account initialization transients and the need to discard stale synopses.

VI. Conclusions

We believe that P2P systems must be designed for *adaptation, scalability,* and *intuitive control*. Our focus in this paper is on designing algorithms for search selection and flooding algorithms for P2P search networks. We now summarize our work, with a view to illustrating the effect of these design principles.

The search selection algorithm presented in Section II uses global statistics to improve efficiency, but these statistics are collected using completely distributed synopsis aggregation through randomized gossip. This allows unlimited scalability. The flood threshold is adapted depending on the actual performance of the system. The adaptation process itself uses user utilities, and, as we show in Section IV.G, this allows system behavior to be controlled by intuitive 'control knobs'. For instance, a user who wants more results can get them simply by asking for it: the system does whatever is needed to achieve this result. We believe that explicit communication of user utilities, and its use in modifying system behavior, is a generally-applicable technique for making distributed systems easier to manage.

Similarly, the adaptive flooding technique presented in Section III uses measurement of the current results in the search path to adapt the flood depth. Scaling is assured by flooding no more than necessary, and then only for popular items that we expect to find with shallow floods anyway. An interesting aspect of our flooding technique is to pre-emptively drop requests that are doomed to time out: theThe user control 'knob' here is simply the timeout value. If a user sets smaller timeouts, the system adjusts itself to prioritize this request as long as it does

not jeopardize any other requests. Naturally, we expect that control knob settings to be paid for by users, otherwise every user will want the best possible performance!

Our techniques leverage classical queueing theory, which we use for a novel analysis of search subsystem overload and congestion collapse. We also incorporate recent advances in the use of approximate counting to create order and duplication insensitive synopses.

We quantified gains from our algorithms using simulation on both synthetic and trace-based workloads. We found that adaptive flooding provides the same recall as a dynamic query, but with 22% lower bandwidth cost. Similarly, compared to a non-adaptive hybrid approach our adaptive algorithm can achieve a 51% smaller last response time, and 1.9 times smaller bandwidth use, with no loss in recall. Our algorithms scale well, with only a 3-9% degradation in performance with a 3X increase in system size. These lead us to conclude that our design principles can greatly improve performance of both existing and future large-scale distributed systems.

VII. Future Work

In future work, we plan to implement our system by modifying the Phex Gnutella client to use the OpenDHT framework [35]. We also plan to investigate using adaptive techniques to adapt the remaining tuning parameters, such as the synopsis discard time. An interesting open question is whether adaptation can be used for the challenging problem of neighbor selection. More generally, we intend to evaluate other adaptive, scaleable, and intuitively controllable algorithms for large-scale distributed systems.

VIII. Acknowledgments

We gratefully acknowledge Boon Loo's support in providing us Gnutella traces. Joe Hellerstein and Rajeev Motwani were instrumental in steering us to recent work in synopsis aggregation. The presentation of the paper was greatly improved by suggestions from Martin Karsten. The efficient computation of coin tossing synopses benefited from discussions with Ian Munro, Alex Lopez-Ortiz, and Tim Brecht. This research was supported by grants from the National Science and Engineering Council of Canada, the Canada Research Chair Program, Intel Corporation, and Sprint Corporation.

IX. References

[1] B.T. Loo, R. Huebsch, I. Stoica, and J.M. Hellerstein, "The Case for a Hybrid P2P Search Infrastructure," *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] S. Dwarkadas, and C. Tang, "Hybrid Global-Local Indexing for Efficient Peer-to-Peer Information Retrieval," *Proc. NSDI*, 2004.

[6] 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.

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

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

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

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

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

[12] Wikipedia: English Language http://en.wikipedia.org/wiki/English_language.

[13] M. Nelson, "Data Compression with the Burrows-Wheeler Transform," Dr. Dobb's Journal, September1996.

[14] S. Saroiu, K.P. Gummadi, R.J. Dunn, S.D. Gribble, H.M. Levy, "An Analysis of Internet Content Delivery Systems," *Proc. OSDI*, Dec. 2002

[15] 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.

[16] E. Cinlar, "Superposition of Point Processes," in *Stochastic Point Processes: Statistical Analysis, Theory, and Applications*, pp. 549-606, Wiley Interscience, 1972.

[17] O. Brun and J.-M. Garcia, Analytical Solution of Finite Capacity M/D/1 Queues, *J. Appl. Prob.*, Vol. 37, pp. 1092-1098, 2000.

[18] R. Brown, "Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem," *Comm. ACM,* Vol. 31, No. 10, October 1988.

[19] C. R. Kalmanek, H. Kanakia and S. Keshav, "Rate Controlled Servers for Very High Speed Networks," *Proc. Globecom 1990*, December 1990.

[20] Limewire FAQ `http://www.limewire.com/english/content/downloads/presskit/faq.pdf`

[21] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, "Making Gnutella-like P2P Systems Scalable," *ACM SIGCOMM 2003*, August 2003.

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

[23] Kazaa, `http://www.kazaa.com/us/`

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

[25] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica, "Querying the Internet with PIER," *Proceedings of 19th VLDB*, Sept. 2003.

[26] G. Goodwin, and K. S. Sin, "Adaptive Filtering Prediction And Control," *Prentice Hall*, 1984.

[27] J. Kubiatowicz, et al., "OceanStore: An Architecture for Global-Scale Persistent Storage," *ASPLOS*, December 2000.

[28] A. Montresor, H. Meling, and O. Babaoglo, "Towards adaptive, resilient and self-organizing peer-to-peer systems," *Proc. IPTPS,* 2002

[29] Z. Ge, D.R. Figueredo, S. Jaiswal, J. Kurose, and D. Towsley, "Modeling Peer-Peer File Sharing Systems," *Proc. IEEE INFOCOM 2003*, April 2003.

[30] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Stuygis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," *PODC*, 1987.

[31] D. Kempe and J. Kleinberg, "Protocols and Impossibility Results for Gossip-Based Communication Mechanisms," *Proc. FOCS*, 2002.

[32] I. Gupta, R. van Renesse, and K. Birman, "Scalable Fault-tolerant Aggregation in Large Process Groups," *Proc. Conf. on Dependable Systems and Networks*, 2001.

[33] J. Considine, F. Li, G. Kollios, and J. Byers, " Approximate aggregation techniques for sensor databases," *Proceedings of the International Conference on Data Engineering*, March 2004.

[34] D. Carta, "Two fast implementations of the "minimal standard" random number generator," *Comm. ACM*, Vol. 33, No. 1, pp.87-88, 1990.

[35] OpenDHT web site, http://www.opendht.org