# Improving Search in Peer-to-Peer Networks

Beverly Yang Hector Garcia-Molina
{byang, hector}@cs.stanford.edu
Computer Science Department, Stanford University

## Abstract

*Peer-to-peer systems have emerged as a popular way to share huge volumes of data. The usability of these systems depends on effective techniques to find and retrieve data; however, current techniques used in existing P2P systems are often very inefficient. In this paper, we present three techniques for efficient search in P2P systems. We present the design of these techniques, and then evaluate them using a combination of analysis and experiments over Gnutella, the largest open P2P system in operation. We show that while our techniques maintain the same quality of results as currently used techniques, they use up to 5 times fewer resources. In addition, we designed our techniques to be simple, so that they can be easily incorporated into existing systems for immediate impact.*

## 1 Introduction

Peer-to-peer (P2P) systems are distributed systems in which nodes of equal roles and capabilities exchange information and services directly with each other. In recent years, P2P has emerged as a popular way to share huge volumes of data. For example, the Morpheus [8] multimedia file-sharing system reported over 470,000 users sharing a total of .36 petabytes of data as of October 26, 2001. Sharing such large volumes of data is made possible by distributing the main costs – disk space for storing the files and bandwidth for transferring them – across the peers in the network. In addition to the ability to pool together and harness large amounts of resources, the strengths of existing P2P systems (e.g., [9], [6], [5], [8]) include self-organization, load-balancing, adaptation, and fault tolerance. Because of these desirable qualities, many research projects have been focused on understanding the issues surrounding these systems and improving their performance (e.g., [13], [7], [4]).

The key to the usability of a data-sharing P2P system, and one of the most challenging design aspects, is efficient techniques for search and retrieval of data. The best search techniques for a system depends on the needs of the application. For storage or archival systems focusing on availability, search techniques such as [16, 12, 21, 11] are well-suited, because they guarantee location of content if it exists, within a bounded number of hops. To achieve these properties, these techniques tightly control the data placement and topology within the network, and currently only support search by identifier.

In systems where persistence and availability are not guaranteed or necessary, such as Gnutella [6], Napster [9] and Morpheus [8], search techniques can afford to have looser guarantees. In addition, because these systems are meant for a wide range of users from non-cooperating organizations, techniques can not afford to strictly regulate the network. Friends may want to connect to other friends, while strangers may not want to store data (potentially large amounts) on behalf of each other. Also, these systems traditionally offer support for richer queries than just identifier lookups, such as keyword search with regular expressions. Search techniques for these "loose" systems must therefore operate under a different set of constraints than techniques developed for persistent storage utilities.

Current search techniques in "loose" P2P systems tend to be very inefficient, either generating too much load on the system, or providing a bad user experience. In this paper, we design and evaluate new search techniques for loosely controlled, loose guarantee systems such as Gnutella and Morpheus. In particular, our main contributions are:

- We present several search techniques that achieve huge performance gains over current techniques, but are simple and practical enough to be easily incorporated into existing systems (Section 4).
- We evaluate our techniques using large amounts of data gathered from Gnutella, the largest open P2P network in operation (Section 5).
- We highlight the strengths of each technique as well as the weaknesses, and translate these tradeoffs into practical recommendations for today's systems (Section 6).

The basic idea behind our techniques is to reduce the number of nodes that process a query. Past work in [18], and our own experiments (Section 6), show that most queries can be answered by querying fewer nodes than the current techniques. Our first technique, *iterative deepening* (Section 4.1), iteratively sends the query to more nodes until the query is answered. The *Directed BFS* technique (Section 4.2), queries a restricted set of nodes intelligently selected to maximize the probability that the query will be answered. If we allow nodes to answer queries on behalf of other nodes, then we can still reduce the number of nodes that process a query without decreasing the number of results. In the *Local Indices* technique (Section 4.3), nodes maintain simple indices over other nodes' data. Queries are then processed by a smaller set of nodes.

## 2 Related Work

The original motivation for much P2P research was early "loose" file-sharing P2P systems such as Gnutella [6], Napster [9], Freenet [5], and Morpheus [8]. The performance of search techniques used in Gnutella and Freenet are discussed in Section 3.2. Napster is not a pure P2P system, but rather a *hybrid* one containing some centralized components; performance of hybrid P2P systems is explored in [18]. Morpheus is a newer, very popular system which has an architecture partway between Napster's and Gnutella's. "Super-peer" nodes act as a centralized resource for a small number of clients, but these super-peers connect to each other to form a pure P2P network. Our techniques are applicable to the P2P network of super-peers. Performance of "super-peer networks" is studied in [19].

Other search techniques for "loose" systems include [1], [3] and [10]. The search technique proposed in [1] is similar to our Local Indices technique (Section 4.3), but the routing policy for Query messages differ, resulting different performance and cost tradeoffs. References [3] and [10] propose that each node maintain "hints" as to which nodes contain data that can answer certain queries, and route messages via local decisions based on these hints.

Search techniques for systems with strong guarantees on availability include Chord [16], CAN [11], Pastry [12], and Tapestry [21]. With deliberately formed connections and intelligent routing, these systems can locate an object by its identifier within a bounded number of hops. These techniques perform well for the systems they were intended for ([4],[13],[7]), but we reiterate that they may not be appropriate for the type of system we are interested in studying.

Finally, one of the main contributions of our work is evaluating techniques using extensive measurements over the Gnutella network. References [2] and [15] measure user characteristics through passive observation of messages. In addition, [15] measures hardware characteristics of peers.

## 3 Problem Overview

The purpose of a data-sharing P2P system is to accept queries from users, and locate and return data (or pointers to the data). Each node owns a collection of files or records to be shared with other nodes. The shared data usually consists of files, but is not restricted to files (e.g., records stored in a relational database). Queries may take any form that is appropriate given the type of data shared. If the system is a file-sharing system, queries may be file identifiers, or keywords with regular expressions, for example.

Two nodes maintaining an open connection (or "edge") are known as *neighbors*. Messages must travel along a sequences of edges. The length of this path is the "number of *hops*" taken by the message. Two nodes are said to be "$n$ hops apart" if the shortest path between them has length $n$.

When a user submits a query, her *source* node sends the query message to a set of its neighbors. The routing policy determines which neighbors (possibly all) are in this set. When a node receives a Query message, it may forward the query to a set of its neighbors, depending on the routing policy. The node will also process the query over its local collection and produce results (such as pointers to files). If any results are found at that node, the node will send a single Response message back to the query source via the reverse path traveled by the query. The total result set for a query is the bag union of results from every node that processes it.

### 3.1 Metrics

**Cost.** When a query message is propagated through the network, nodes spend processing resources (i.e., cycles) to forward the query, process it, etc, and bandwidth to send and receive messages. The main cost of queries are therefore bandwidth and processing. Since the cost of a query is not incurred at any single node in the network, it makes sense to discuss costs in *aggregate* (i.e., over all the nodes in the network). Furthermore, we should not evaluate a policy based on the performance of any single query, so instead we measure the *average* aggregate cost incurred by a set of queries $Q_{rep}$, where $Q_{rep}$ is a representative set of real queries submitted. Our two cost metrics are therefore:

- **Average Aggregate Bandwidth**
- **Average Aggregate Processing Cost**

It is not possible to directly measure these metrics; instead, we use analysis based on measured costs of individual actions to estimate these values (Section 5.2).

**Quality of Results.** Although we seek to reduce the cost of the query, we must nevertheless ensure our techniques do not degrade the user experience. Quality of results are measured by the following metrics:

- **Number of results**: the size of the total result set.
- **Satisfaction**: Some queries may receive hundreds or thousands of results. Rather than notifying the user of every result, the clients in many systems (such as Napster and Gnutella) display the first $Z$ results only, where $Z$ is some value specified by the user. We say a query is *satisfied* if $Z$ or more results are returned. The idea is that given a sufficiently large $Z$, the user can find what she is looking for from the first $Z$ results. Hence, if $Z = 50$, a query that returns 1000 results performs no better than a query returning 100 results.
- **Time to Satisfaction**: the time that has elapsed from when the query is first submitted by the user, to when

the user's client receives the $Z$th result.

In general, we observe a tradeoff between cost and quality.

### 3.2 Current Techniques

We will look at the techniques currently used by two well-known operating P2P systems:

- **Gnutella**: Uses a breadth-first traversal (BFS) with depth limit $D$, where $D$ is the system-wide maximum time-to-live of a message in hops. Every node receiving a Query will forward the message to all of its neighbors, unless the message has already traveled $D$ hops.
- **Freenet**: Uses a depth-first traversal (DFS) with depth limit $D$. Each node forwards the query to a single neighbor, and waits for a definite response from the neighbor before forwarding the query to another neighbor (if the query was not satisfied), or forwarding results back to the query source (if the query was satisfied).

If the quality of results in a system were measured solely by the number of results, then the BFS technique is ideal because it sends the query to every possible node (i.e., all nodes within $D$ hops), as quickly as possible. However, if satisfaction were the metric of choice, BFS wastes resources because, as we mentioned earlier, most queries can be satisfied from the responses of relatively few nodes. With DFS, because each node processes the query sequentially, searches can be terminated as soon as the query is satisfied, thereby minimizing cost. However, sequential execution also translates to poor response time, with the worst case being exponential in $D$. Actual response time in Freenet is moderate, because $Z = 1$ and intelligent routing is used.

As we can see, existing techniques fall on opposite extremes of bandwidth/processing cost and response time. Our goal is to find some middle ground between the two extremes, while maintaining quality of results.

## 4 Broadcast Policies

### 4.1 Iterative Deepening

In systems where satisfaction is the metric of choice, a good technique is *iterative deepening*. Iterative deepening is a well-known technique used in other contexts, such as search over state space in artificial intelligence [14]. In iterative deepening, multiple breadth-first searches are initiated with successively larger depth limits, until either the query is satisfied, or the maximum depth $D$ has been reached. To implement the iterative deepening technique, a system-wide *policy* is needed that specifies at which depths the iterations are to occur. For example, say we want to have three iterations: the first iteration searches to a depth $a$, the second to depth $b$, and the third to depth $c$. Our policy is therefore $P = \{a, b, c\}$. In addition to a policy, we must specify the time between successive iterations in the policy, $W$, explained in further detail below.

Under the policy $P = \{a, b, c\}$, a source node $S$ first initiates a BFS of depth $a$. When a node at depth $a$ receives and processes the message, it will store the message temporarily. The query therefore becomes *frozen* at all nodes that are $a$ hops from the source. Meanwhile, $S$ receives Response messages from nodes that have processed the query. After waiting for a time period $W$, if the query has been satisfied, then $S$ does nothing; otherwise $S$ will start the next iteration, initiating a BFS of depth $b$.

To initiate the next BFS, $S$ will send a *Resend* with a TTL of $a$. Instead of reprocessing the query, a node that receives a Resend message will simply forward the message, or if the node is at depth $a$, it will drop the Resend message and "unfreeze" the corresponding query by forwarding the Query message (with a TTL of $b - a$) to all its neighbors. To match queries with Resend messages, every query is assigned a system-wide "almost unique" identifier (which Gnutella does). The Resend message will contain the identifier of the query it is representing, and nodes will know which query to unfreeze by inspecting this identifier. Note that a node need only freeze a query for slightly more than $W$ time units before deleting it.

After the search to depth $b$, the process continues in a similar fashion to the other levels in the policy. Since $c$ is the depth of the last iteration in the policy, queries will not be frozen at depth $c$, and $S$ will not initiate another iteration, even if the query is still not satisfied.

### 4.2 Directed BFS

If minimizing response time is important to an application, then iterative deepening may not be appropriate because of the time taken by multiple iterations. A better strategy would be to send queries immediately to a subset of nodes that will return many results, and will do so quickly. The *Directed BFS* (DBFS) technique implements this strategy by having a source send query messages to just a subset of its neighbors, thereby reducing cost, but selecting neighbors through which nodes with many quality results may be reached, thereby maintaining quality of results. For example, one may select a neighbor that has produced or forwarded many quality results in the past, on the premise that past performance is a good indication of future performance. The neighbors that receive the query then continue forwarding the message to all neighbors as with BFS.

In order to intelligently select neighbors, a node maintains simple statistics on its neighbors, such as the number of results received through that neighbor for past queries, or the latency of the connection with that neighbor. From these statistics, we can develop a number of heuristics to help us select the best neighbor to send the query, such as:

- Select the neighbor that has returned the highest number of results for previous queries.
- Select neighbor that returns response messages that have taken the lowest average number of hops. A low hop-count may suggest that this neighbor is close to

nodes containing useful data.

- Select the neighbor that has forwarded the largest number of messages (all types) our client. A high message count implies that this neighbor is stable and it can handle a large flow of messages.
- Select the neighbor with the shortest message queue. A long message queue implies that the neighbor's pipe is saturated, or that the neighbor has died.

In our experiments, a query source sends the query to a single neighbor only. Surprisingly, we will see that the quality of results does not decrease significantly, provided that we make intelligent neighbor selections.

### 4.3 Local Indices

In the Local Indices technique, each node $n$ maintains an index over the data of all nodes within $r$ hops of itself, where $r$ is a *system-wide* variable known as the *radius* of the index ($r = 0$ is the degenerate BFS case, where a node only indexes metadata over its own collection). When a node receives a Query message, it can process the query on behalf of every node within $r$ hops. In this way, the data of many nodes can be searched by processing the query at few nodes, thereby maintaining satisfaction and number of results while keeping costs low. When $r$ is small, the amount of metadata a node must index is also quite small, on the order of 50 KB. As a result, Local Indices with small $r$ should be easily accepted by a loosely controlled system such as Gnutella. See Section 6.3 for details on space requirements.

The Local Indices technique works as follows: a system-wide *policy* specifies the depths at which the query should be processed. All nodes at depths not listed in the policy simply forward the query to the next depth. For example, say the policy is $P = \{1, 5\}$. Query source $S$ will send the Query message out to its neighbors at depth 1. All these nodes will process the query, and forward the Query message to all their neighbors at depth 2. Nodes at depth 2 will not process the query, but will forward the Query message to depth 3. Eventually, nodes at depth 5 will process the query, since depth 5 is in the policy. Because depth 5 is the last depth in $P$, these nodes will then drop the Query. Note the difference between a Local Indices policy and an iterative deepening policy, where depths in the policy represent the depths at which iterations should end, and nodes at *all* depths process the query.

To create and maintain the indices at each node, extra steps must be taken whenever a node joins, leaves, or updates its data. When a node $X$ joins the network, it sends a Join message with a TTL of $r$, containing metadata over its collection. When a node receives the Join message from $X$, it will send a Join message containing metadata over its collection directly to $X$ (i.e., over a temporary connection). Both nodes then add each other's metadata to their own index. When a node joins the network or a new connection is made, a path of length $r$ may be created between two nodes where no such path previously existed. In this case, the two nodes can be made aware of this path in a number of ways without introducing additional messages (see [20]).

When a node leaves the network or dies, other nodes that index this node's collection will remove its metadata after a timeout. When a user updates his collection, his node will send out a small Update message with a TTL of $r$, containing the metadata of the affected data All nodes receiving this message subsequently update their index.

To translate the cost of joins, leaves and updates to query performance, we amortize these costs over the cost of queries. The parameter `QueryJoinRatio` gives us the average ratio of queries to joins in the entire P2P network, while `QueryUpdateRatio` gives us the average ratio of queries to updates.

| Description | Value |
|---|---|
| Average files shared per user | 340 files |
| Average size of result record | 76 B |
| Average size of metadata for a single file | 72 B |
| Percentage of Query messages dropped ([20]) | 30% |

**Table 1.** General Statistics

## 5 Experimental Setup

We chose to base our evaluations on data gathered from the Gnutella network because Gnutella is the largest open P2P system in operation, with about 50000 users as of May, 2001. In evaluating our techniques, some metrics can be directly measured through experiments (e.g., satisfaction of a BFS query), while others must be indirectly measured (e.g., satisfaction for a local indices query, since we could not force all Gnutella nodes to use this technique for our experiments). For the latter type, we instead collect performance data for queries under the BFS technique, and combine this data using analysis to estimate what the performance would have been under our techniques. Experiments and analysis are described in Sections 5.1 and 5.2, respectively.

### 5.1 Data Collection

First, we needed to gather some general information on the Gnutella network and its users. For example, how many files do users share? What is the typical size of metadata for a file? To gather these general statistics, for a period of one month, we ran a Gnutella client that observed messages as they passed through the network. Based on the content of these messages, our client could determine characteristics of users' collections, and of the network as a whole. Table 1 summarizes some of the general characteristics we will use later on in our analysis. Our client also passively observed the query strings of query messages that passed through the network. To get our representative set of queries for Gnutella, $Q_{rep}$ (see Section 3.1) , we randomly selected 500 queries from the 500,000 observed queries.

| Symbol | Description |
|---|---|
| $L(Q)$ | Length of query string for query $Q$ |
| $M(Q,n)$ | Number of response messages received for query $Q$, from $n$ hops away |
| $R(Q,n)$ | Number of results received for query $Q$, from $n$ hops away |
| $S(Q,n,Z)$ | Returns true if query $Q$ received $Z$ or more results from $n$ hops away |
| $T(Q,Z,W,P)$ | Time to satisfaction of query $Q$, under iterative deepening policy $P$ and waiting time $W$ |
| $N(Q,n)$ | Number of nodes $n$ hops away that process $Q$ |
| $C(Q,n)$ | Number of redundant edges $n$ hops away |

**Table 2.** Symbols and function names of data extracted from logs for iterative deepening

#### 5.1.1 Iterative Deepening

For each query $Q$ in the representative set $Q_{rep}$, our client submitted $Q$ to the live Gnutella network $D$ times (spread over time), where $D = 7$ is the maximum TTL allowed in Gnutella. Each time we incremented the TTL by 1, so that we submitted $Q$ once for each TTL between 1 and $D$. For each Query message submitted, we logged every Response message arriving within 2 minutes of submission of the query. For each Response, we log:

- The number hops that the Response message took.
- The response time (i.e., the time elapsed from when the Query message was first submitted, to when the Response message was received).
- The IP address from which the Response message came.
- The individual results contained in the message.

As queries are submitted, our client sent out Ping messages to all its neighbors. Ping messages are propagated through the network in a breadth-first traversal, as Query messages are. When a node receives a Ping message, it replies with a Pong message containing its IP. We sent a Ping message immediately before every second query, and logged the following information for all Pong messages received in the next 4 minutes:

- The number of hops that the Pong message took.
- The IP address from which the Pong came.

From these Response and Pong logs, we can directly extract information necessary to estimate the cost and quality of results for each query, summarized in the first half of Table 2. Each of these data elements were extracted for every query $Q \in Q_{rep}$, and for every possible hop value $n$ ($n = 1$ to $D$). In the definitions of $R(Q,n)$ and $M(Q,n)$, note that a single Response message can hold more than one result.

We also estimated several values, listed in the second half of Table 2. These values could not be directly observed, but they are nevertheless carefully calculated from observed quantities (see [20] for details).

The data gathered for Iterative Deepening is also used to evaluate Local Indices. No separate experiments were run to gather data for Local Indices.

| Symbol | Description |
|---|---|
| $L(Q)$ | Length of query string for query $Q$ |
| $M(Q,n,y)$ | Number of response messages received for query $Q$, from nodes $n$ hops away, when the Query message was sent to neighbor $y$ |
| $R(Q,n,y)$ | Number of results received for query $Q$, from nodes $n$ hops away, when the Query message was sent to neighbor $y$ |
| $S(Q,n,Z,y)$ | Returns true if query $Q$ receives $Z$ or more results from $n$ hops away, when the Query message was sent to neighbor $y$ |
| $T(Q,Z,y)$ | The time at which query $Q$ is satisfied, when the query was sent to neighbor $y$ |
| $N(Q,n,y)$ | Number of nodes $n$ hops away that process $Q$ when the Query message was sent to neighbor $y$ |
| $C(Q,n,y)$ | Number of redundant edges $n$ hops away when the Query message was sent to neighbor $y$ |

**Table 3.** Symbols and function names of data extracted from logs for Directed BFS

#### 5.1.2 Directed BFS

The experiments for DBFS are similar to the experiments for iterative deepening, except each query in $Q_{rep}$ is now sent to a single neighbor at a time. That is, rather than sending the same Query message, with the same message ID, to all neighbors, our node sends a Query message with a different ID (but same query string) to each neighbor. Similarly, Ping messages with distinct IDs are also sent to a single neighbor at a time, before every other query.

For each Response and Ping received, our client logs the same information logged for iterative deepening, in addition to the neighbor from which the message is received. From our logs, we then extract the same kind of information as with iterative deepening, for each query and neighbor. The data we extract from the logs is listed in Table 3. Note that unlike with iterative deepening, the time to satisfaction in Directed BFS – $T(Q,Z,y)$ – is directly observable by our client. We also estimate several values that could not be directly extracted, listed in Table 3, in the same manner as the values in Table 2.

In addition to gathering Response and Pong information, we also recorded statistics for each neighbor right before each query was sent out, such as the number of results that a neighbor has returned on past queries, and the latency of the connection with a neighbor. Recall that these statistics are used to select to which neighbor we forward the query.

### 5.2 Calculating Costs

Given the data we collect from the Gnutella network, we can now estimate the cost and performance of each technique through analysis. The following subsections summa-

| Symbol | Value (Bytes) | Description |
|---|---|---|
| $a(Q)$ | $82 + L(Q)$ | Size of a Query message |
| $b$ | 80 | Size of a Resend message |
| $c$ | 76 | Size of a single result record |
| $d$ | 108 | Size of a Response message header |
| $e(Q, r)$ | see discussion in [20] | Size of a full Response message under Local Indices |
| $f$ | 24560 | Size of a Join message |
| $g$ | 152 | Size of an Update message |

**Table 4.** Sizes of messages

rize our calculations for cost. Calculations for time to satisfaction can be found in [20].

### 5.2.1 Bandwidth Cost

To calculate the average aggregate bandwidth consumed under a particular technique, we first estimate how large each type of message is. We base our calculations of message size on the Gnutella network protocol, and the general statistics listed in Table 1. For example, a Query message contains a Gnutella header, a query string, and a field of 2 bytes for options. Headers in Gnutella are 22 bytes, TCP/IP and Ethernet headers are 58 bytes, and the query string is $L(Q)$ bytes (Table 2). Total message size is therefore 82 + $L(Q)$ bytes. Table 4 lists the different message types used by our techniques, giving their estimated sizes and the symbol used for compact representation of the message size.

We can now use the message sizes and logged information to estimate aggregate bandwidth consumed by a single query under the various techniques. For example, aggregate bandwidth for a query under BFS is:

$$\begin{aligned} BW_{bfs}(Q) &= \sum_{n=1}^{D} \Big( a(Q) \cdot \Big( N(Q,n) + C(Q,n) \Big) \\ &+ n \cdot \Big( c \cdot R(Q,n) + d \cdot M(Q,n) \Big) \Big) \quad (1) \end{aligned}$$

The first term inside the summation gives us the bandwidth consumed by sending the query message from level $n-1$ to level $n$. There are $N(Q,n)$ nodes at depth $n$, and there are $C(Q,n)$ redundant edges between depths $n-1$ and $n$. Hence, the total number of Query messages sent on the $n$th hop is equal to $N(Q,n) + C(Q,n)$. If we multiply this sum by $a(Q)$, the size of the Query message, we get the bandwidth consumed by forwarding the query on the $n$th hop. The second term gives us the bandwidth consumed by transferring Response messages from $n$ hops away back to the query source. There are $M(Q,n)$ Response messages returned from nodes $n$, and these Response messages contain a total of $R(Q,n)$ result records. The size of a response message header is $d$, and the size of a result record is $c$ (on average), hence the total size of all Response messages returned from $n$ hops away is $c \cdot R(Q,n) + d \cdot M(Q,n)$. These messages must take $n$ hops to get back to the source; hence, bandwidth consumed is $n$ times the size of all responses.

| Symbol | Cost (Units) | Description |
|---|---|---|
| $s$ | 1 | Cost of transferring a Resend message |
| $t(Q)$ | $1 + .007 \cdot L(Q)$ | Cost of transferring a Query message |
| $u$ | .5 | Additional cost of sending a Response message when a result record is appended to the message |
| $v$ | 1.2 | Base cost of transferring a Response message with no result records |
| $w$ | 1.1 | Additional cost of processing a query per result discovered |
| $x$ | 14 | Base overhead of processing a query |
| $y(Q, r)$ | see discussion in [20] | Cost of transferring a Response message under Local Indices |
| $z(Q, r)$ | see discussion in [20] | Cost of processing a query under Local Indices |
| $h$ | 3500 | Cost of processing a Join message |
| $j$ | 160 | Cost of transferring a Join message |
| $k$ | 3500 | Cost of processing a timeout (removing metadata) |
| $l$ | 1.4 | Cost of transferring an Update message |
| $\Delta$ | 30 | Cost of processing an Update message |

**Table 5.** Costs of actions

Formulae for calculating aggregate bandwidth consumption for the remaining policies – iterative deepening, Directed BFS, and Local Indices – follow the same pattern, and include the same level of detail, as Equation 1. Due to space limitations, the formulae and derivations are not included here, but can be found in [20].

### 5.2.2 Processing Cost

To calculate processing costs, we first estimate how much processing power each type of action requires. Table 5 lists the different types of actions needed to handle queries, along with their cost in units and the symbol used for compact representation of the actions' cost. Costs are expressed in terms of coarse units, where the base unit is defined as the cost of transferring a Resend message, roughly 7300 cycles. Costs were estimated by running each type of action on a Pentium III 930 MHz processor (Linux version 2.2) While CPU time will vary between machines, the relative cost of actions should remain roughly the same.

Because of space limitations, we do not present formulae for calculating average aggregate processing cost. Please see [20] for the formulae and their derivations.

## 6 Experiments

In this section, we present the results of our experiments and analysis. As a convenience to the reader, some symbols defined in previous sections are re-defined in Table 6. Note that our evaluations are performed over a single "real" system, and that results may vary for other topologies. Nevertheless, since Gnutella does not control topology or data placement, we believe its characteristics are representative of the type of system we want to study.

| Symbol | Definition |
|---|---|
| $D$ | Maximum time-to-live of a message, in terms of hops |
| $Z$ | Number of results needed to *satisfy* a query |
| $Q_{rep}$ | Representative set of queries for the Gnutella network |
| $W$ | Waiting time (in seconds) between iterations |

**Table 6.** Definition of Symbols

Due to space limitations, we will not show figures for the processing cost metric, though we will cite specific numbers. Behavior of this metric is analogous to that of bandwidth cost; hence, the bandwidth cost figures give us the shape (i.e., illustrate the same tradeoffs) for processing cost.

### 6.1 Iterative Deepening

In order for iterative deepening to have the same satisfaction performance as a BFS of depth $D$, the last depth in the policy must equal $D$. For the sake of comparison, we evaluate only these kinds of policies. To understand the tradeoffs between policies of different lengths, we choose the following subset of policies to study:

$$P = \{P_d = \{d, d+1, ...., D\}, \text{ for } d = 1, 2, ..., D\}$$
$$= \{\{1, 2, ..., D\}, \{2, 3, ..., D\}, ..., \{D-1, D\}, \{D\}\}.$$

Since a policy $P_d$ is defined by the value of $d$, the depth of its first iteration, we call $d$ the "policy number". Recall that $P_D = P_7 = \{7\}$ is the degenerate case, a BFS of depth 7, currently used in the Gnutella network. Similarly, we looked only at a few possible $W$ values: $\{1, 2, 4, 6, 150\}$. These values are measured in seconds.

In our experiments, our client maintained 8 neighbors, and we defined the desired number of results $Z = 50$. In [20] we study the effects of varying $Z$ and the number of neighbors of our client. In general, increasing $Z$ results in a lower probability of satisfaction and higher cost, but increased number of results. Decreasing the number of neighbors results in slightly lower probability of satisfaction, but significantly lower cost.

We note that our client ran over an 10 Mb Ethernet connection. To ensure that our client would not die and interrupt the ongoing experiments, we had to ensure that the connection was stable and did not saturate. Most peer-to-peer clients will be connected via lower bandwidth connections, so we must keep in mind that the absolute numbers that we see in the following graphs may not be the same across all clients, though the tradeoffs should be comparable.

**Cost Comparison.** Figure 1 shows the cost of each policy, for each value of $W$, in terms of average aggregate bandwidth and processing cost, respectively. Along the x-axis, we vary $d$, the policy number. The cost savings are immediately obvious in these figures. Policy $P_1$ at $W = 8$ uses just about 19% of the aggregate bandwidth per query used by the BFS technique, $P_7$, and just 40% of the aggregate processing cost per query.

To understand how such enormous savings are possible, we must understand the tradeoffs between the different policies and waiting periods. First, notice that the average aggregate bandwidth for $d = 7$ is the same regardless of $W$. Since $W$ is the waiting time between iterations, it does not affect $P_7 = \{7\}$, which has only a single iteration.

Next, notice that as $d$ increases, the cost of policy $P_d$ increases as well. The larger $d$ is, the more likely the policy will waste bandwidth by sending the query to too many nodes. For example, if a query $Q$ can be satisfied at a depth of 4, then policies $P_5$, $P_6$, and $P_7$ will "overshoot" the goal, sending the query out to more nodes than necessary to satisfy $Q$. Sending the query out to more nodes than necessary will generate extra bandwidth from forwarding Query and Response messages.

Now, notice that as $W$ decreases, cost increases. If $W$ is small, there is a higher likelihood that the source will prematurely determine that the query was not satisfied, leading to the "overshooting" effect we described for large policy numbers. For example, say $W = 6$ and $d = 4$. If a query $Q$ can be satisfied at depth 4, but 8 seconds are required before $Z$ results arrive at the client, then the client will only wait for 6 seconds, determine that the query is not satisfied, and initiate the next iteration at depth 5. In this case, the client overshoots the goal. The smaller $W$ is, the more often the client will overshoot.

**Quality of Results.** Recall that one of the strengths of iterative deepening is that it can decrease the cost of queries without detracting from its ability to satisfy queries. Hence, satisfaction under any iterative deepening policy is equal to satisfaction under the current BFS scheme used by Gnutella, which, according to our data, equals .64. Because the iterative deepening technique is best applied in cases where satisfaction, and not the number of responses, is the more appropriate metric, we will not compare policies by the number of results returned per query.

The remaining quality of results metric, time to satisfaction, is shown in Figure 2 for each policy and value of $W$. We see that there is an inverse relationship between time to satisfaction and cost. As $W$ increases, the time spent for each iteration grows longer. In addition, as $d$ decreases, the number of iterations needed to satisfy a query will increase, on average. In both cases, the time to satisfaction will increase. Note, however, that delay is often caused by saturated connections or thrashing nodes. If all nodes used iterative deepening, load on nodes and connections would decrease considerably, thereby decreasing these delays. Time to satisfaction should therefore grow less quickly than is shown in Figure 2, as $d$ decreases or $W$ increases.

In deciding which policy would be the best to use in practice, we must keep in mind what time to satisfaction the user can tolerate in an interactive system. Suppose a system requires the average time to satisfaction to be no more than 9 seconds. Looking at Figure 2, we see that several combinations of $d$ and $W$ result in this time to satisfy, e.g., $d = 4$

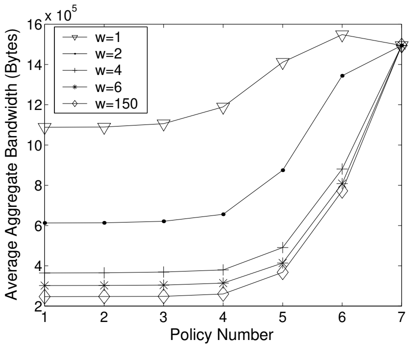


**Figure 1.** Bandwidth consumption for iterative deepening policies

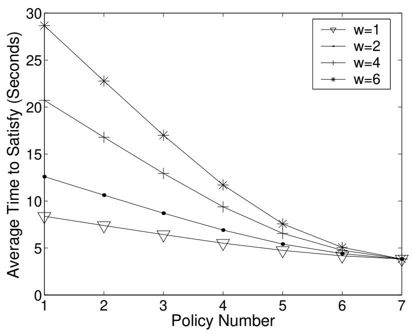


**Figure 2.** Time to satisfaction for iterative deepening policies

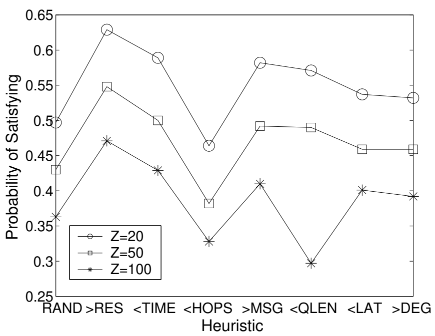


**Figure 3.** Probability of satisfaction for Directed BFS policies

| Symbol | Heuristic: Select neighbor that... |
|---|---|
| RAND | (Random) |
| >RES | Returned the greatest number of results in the past 10 queries |
| <TIME | Had the shortest average time to satisfaction in the past 10 queries |
| <HOPS | Had the smallest average number of hops taken by results in the past 10 queries |
| >MSG | Sent our client the greatest number of messages (all types) |
| <QLEN | Had the shortest message queue |
| <LAT | Had the shortest latency |
| >DEG | Had the highest degree (number of neighbors) |

**Table 7.** Heuristics used for Directed BFS

and $W = 4$, or $d = 5$ and $W = 6$. Looking at Figure 1, we see that the policy and waiting period that minimizes cost, while satisfying the time constraint, is $P_5$ and $W = 6$ (with savings of 72% in aggregate bandwidth and 53% in aggregate processing cost over BFS). We would therefore recommend this policy for our system.

## 6.2 Directed BFS

We studied 8 heuristics for Directed BFS, listed in Table 7. The RAND heuristic, choosing a neighbor at random, is used as the baseline for comparison with other heuristics.

**Quality of Results.** Figures 3 and 4 shows the probability of satisfaction and time to satisfaction, respectively, for the different heuristics and values of $Z$. All heuristics except <HOPS have a marked improvement over the baseline heuristic RAND. In particular, >RES, sending the query to the neighbor that has produced the most results in past queries, has the best satisfaction performance. It is followed by <TIME, sending the query to the neighbor that has produced results with the lowest time to satisfaction in past queries. As with iterative deepening, increasing $Z$ decreases satisfaction for all heuristics. Under the time to satisfaction metric, the <TIME heuristic has the best performance, followed by >RES. As expected, we see that past performance is a good indicator of future performance.

We were surprised to see that >DEG, sending the query to the neighbor with the highest degree, did not perform as well as most other heuristics. Past work such as [1] show that in a power-law network, sending the query to nodes with the highest degree should allow one to reach many nodes in the network, and therefore get many results back. However, this past work did not have real-time data to make decisions as we did, such as the past performance of neighbors. It is possible that with this additional real-time information, their routing algorithm might be enhanced.

We were also surprised to see that <HOPS performed so poorly. On closer examination of our logs, we found that a low average number of hops does not imply that the neighbor is close to many nodes with data, as we had hypothesized. Instead, it typically means that only a few nodes can be reached through that neighbor, but those few nodes are close to the neighbor.

**Cost.** Figure 5 shows the cost of Directed BFS under each heuristic in terms of average aggregate bandwidth. The cost of Directed BFS in unaffected by the value of $Z$, so the single curve in this figure represent all values of $Z$. We see a definite correlation between cost and quality of results. Many of our heuristics return higher quality results than RAND because they select neighbors that are directly or indirectly connected to many other nodes. Because more nodes process the query when using these heuristics, more quality results are returned, but also more aggregate bandwidth and processing power is consumed.

We feel that since users of a system are more acutely aware of the quality of results that are returned, rather than the aggregate cost of a query, the heuristics that provide the highest quality results would be most widely accepted in open systems such as Gnutella. We would therefore recommend >RES or <TIME. Both heuristics provide good

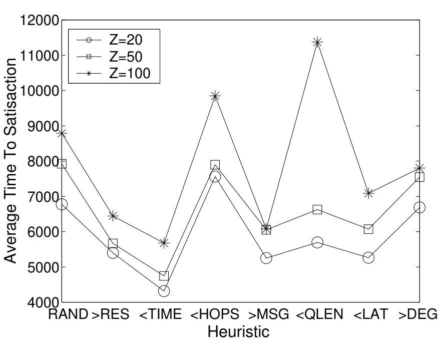


**Figure 4.** Time to satisfaction for Directed BFS policies

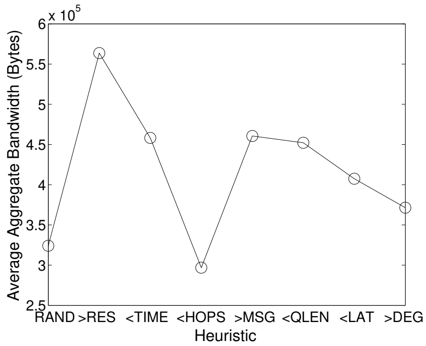


**Figure 5.** Bandwidth consumption for Directed BFS

| Policies |
|---|
| $P_0 = \{1, 2, 3, 4, 5, 6, 7\}$ |
| $P_1 = \{0, 3, 6\}$ |
| $P_2 = \{0, 5\}$ |
| $P_3 = \{4\}$ |
| $P_4 = \{3\}$ |
| $P_5 = \{2\}$ |
| $P_6 = \{1\}$ |
| $P_7 = \{0\}$ |

**Figure 6.** Policies for Local Indices

time to satisfaction, and a probability of satisfaction that is 9% and 13% lower than BFS with 8 neighbors, respectively. Furthermore, despite the fact that they are the most costly heuristics, they still require roughly 73% less processing cost than BFS, and 65% less bandwidth.

Compared with iterative deepening, the strength of Directed BFS is time to satisfaction. Comparing Figures 2 and 4, we see that Directed BFS heuristics yield times to satisfaction comparable to the best times achievable by iterative deepening. However, by sacrificing time to satisfaction, iterative deepening can achieve lower cost than any Directed BFS heuristic. Table 8 in Section 7 summarizes the comparisons between these two techniques.

### 6.3 Local Indices

Due to space limitations, here we only summarize performance for the Local Indices technique. For full experimental results, please refer to [20]. Figure 6 lists the policies we consider for each possible value of radius $r$ (assuming $r$ does not exceed $D$, the maximum depth of search). These policies were chosen to minimize the number of nodes that process the query.

Figure 7 gives us the cost of the Local Indices policies for various values of `QueryJoinRatio` (QJR) in terms of average aggregate bandwidth. Along the x-axis we vary $r$. Recall that QJR gives us the ratio of queries to joins/leaves in the network; the default value observed in Gnutella is roughly 10 [17]. We do not vary `QueryUpdateRatio` because it has the same types of effects as varying QJR, to a lesser degree. We see huge cost savings from the Local Indices technique, especially as QJR increases. At QJR = 10, policy $P_1$ uses about 39% of the bandwidth used by BFS (i.e., $r = 0$), and about 51% of the processing cost. When QJR = 100, only 28% of the bandwidth and 21% of the processing cost of the default scheme is required.

Note that cost decreases as QJR increases. Because there are more queries to joins and leaves as QJR increases, the amortized cost of joins and leaves decreases. Cost at $r = 0$ is unaffected because no indexes are being maintained at $r = 0$. Also note that as $r$ increases, the cost of the policies decrease, and then increase again. The reason for this behavior can be found in Figure 8, where we see individual costs of actions for QJR=20. When $r$ is small, the cost of queries dominates. When $r$ is large, the amortized cost of logins dominates. The minimum of the sum of all costs is found somewhere in the two extremes.

Figure 9 shows us the size of the index that our node would have as a function of $r$, if we implemented Local Indices. The size requirement for a policy is the number of nodes within $r$ hops, multiplied by the average number of files per user, and the average size of each file metadata. The number of neighbors directly affect the size of the index, so we show results for our client with 4 and 8 neighbors (average degree is roughly 3 in Gnutella). At $r = 7$ with 4 neighbors, the size of our index would have been roughly 21 MB, which is not unreasonable, but perhaps larger than many users would like to keep in memory. For $r = 1$, the size of our index would be roughly 71 KB – an index so small that almost certainly no users would object.

Local Indices has the same number of results and satisfaction as BFS. In the absence of data from a system that actually uses indices, calculating time to satisfaction for Local Indices is hard. Qualitative analysis indicates that Local Indices will have performance comparable to BFS; please see [20] for a detailed discussion.

For today's system with QJR = 10, we recommend using $r = 1$, because it achieves the greatest savings in cost (61% in bandwidth, 49% in processing cost), and the index size is so small. future, when QJR increases, the best value for $r$ will also increase.

## 7 Conclusion

This paper presents the design and evaluation of three efficient search techniques over a loosely controlled, pure P2P system. Compared to current techniques used in existing systems these techniques greatly reduce the aggregate cost

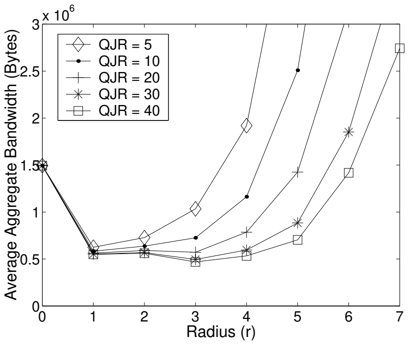


**Figure 7.** Bandwidth consumption for Local Indices

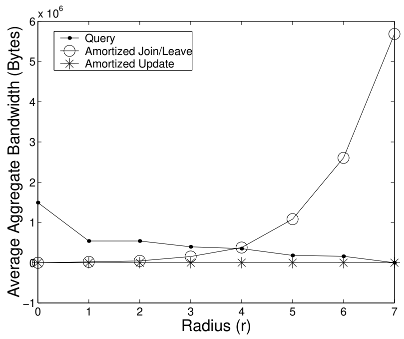


**Figure 8.** Comparison of Bandwidth Consumed by Actions

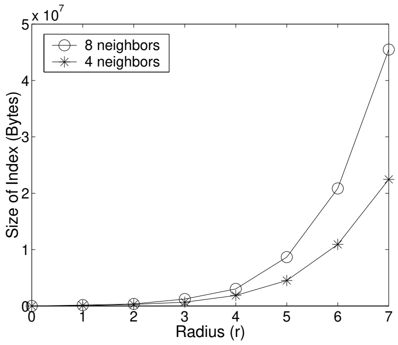


**Figure 9.** Size of the Index for different Radii(r)

| Technique | Time to Satisfy | Probability of Satisfaction | Number of Results | Aggregate Bandwidth | Aggregate Processing |
|---|---|---|---|---|---|
| BFS | 100% | 100% | 100% | 100% | 100% |
| Iterative Deepening ($d = 5, W = 6$) | 190% | 100% | 19% | 28% | 47% |
| Directed BFS (>RES) | 140% | 86% | 37% | 38% | 28% |
| Local Indices ($r = 1$) | $\approx$ 100% | 100% | 100% | 39% | 51% |

**Table 8.** Relative performance of techniques, using BFS as the baseline. For each technique, we show the performance of a single policy we recommend for today's systems.

of processing queries over the entire system, while maintaining equally high quality of results. Table 8 summarizes the performance tradeoffs among our proposed techniques. Because of the simplicity of these techniques and their excellent performance, we believe they can make a large positive impact on both existing and future pure P2P systems.

## References


[1] L. Adamic, R. Lukose, A. Puniyani, and B. Huberman. Search in power-law networks. Available at http://www.parc.xerox.com/istl/groups/iea/papers/plsearch/, 2001.

[2] E. Adar and B. A. Huberman. Free Riding on Gnutella. http://www.firstmonday.dk/issues/issue5_10/adar/index.html, September 2000.

[3] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *Proc. of ICDCS 2002*, July 2002.

[4] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proc. ACM SOSP*, October 2001.

[5] Freenet website. http://freenet.sourceforge.net.

[6] Gnutella website. http://www.gnutella.com.

[7] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Thea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. ASPLOS*, pages 190–201, November 2000.

[8] Morpheus website. http://www.morpheus-os.com.

[9] Napster website. http://www.napster.com.

[10] NeuroGrid website. http://www.neurogrid.net.

[11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, August 2001.

[12] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware 2001*, November 2001.

[13] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proc. ACM SOSP*, October 2001.

[14] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

[15] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. Technical Report UW-CSE-01-06-02, University of Washington, July 2001.

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

[17] K. Truelove. To the bandwidth barrier and beyond. Available at http://web.archive.org/web/20010107185800/dss.clip2.com/gnutella.html.

[18] B. Yang and H. Garcia-Molina. Comparing hybrid peer-to-peer systems. In *Proc. of the 27th Intl. Conf. on Very Large Databases*, September 2001.

[19] B. Yang and H. Garcia-Molina. Designing a super-peer network. Technical report, Stanford University, Feburary 2002. Available at http://dbpubs.stanford.edu/pub/2002-13.

[20] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. Technical report, Stanford University, March 2002. Available at http://dbpubs.stanford.edu/pub/2001-47.

[21] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, April 2001.