# 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 experiments over Gnutella, the largest open P2P system in operation, and analysis. We show that while our techniques maintain the same quality of results as currently used techniques, our techniques use up to 5 times fewer resources. In addition, we designed our techniques to be simple in design and implementation, so that they can be easily incorporated into existing systems for immediate impact.

**Keywords**: Peer-to-peer, distributed data, search, performance modeling, evaluation

**Technical Area(s)**: Databases, performance modeling and evaluation

## 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 given system depends on the needs of the application. For storage or archival systems focusing on availability, search techniques such as Chord [16], Pastry [12],

*Contact author.

Tapestry [19] and CAN [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 object identifier.

In systems where persistence and availability are not guaranteed or necessary, such as Gnutella [6], Freenet [5], Napster [9] and Morpheus [8], search techniques can afford to have looser guarantees. However, because these systems are meant for a wide range of users that come from non-cooperating organizations, the techniques can not afford to strictly control the data placement and topology of 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 search by identifier, 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 for a very bad user experience (see Section 3.2 for a discussion). In this paper, we present the design and evaluation of 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 receive and process each query. Doing so reduces the aggregate load generated by each query across the network. If we assume that each node can only answer queries about their own content, then naturally, the fewer nodes that process the query, the fewer results that will be returned. Our techniques will only be effective if most queries can be answered by querying fewer nodes. Indeed, past work in [18], and our own experiments in Section 6, show that most queries can be answered by querying fewer nodes than the current techniques. Hence, our first technique, *iterative deepening* (Section 4.1), reduces the number of nodes that are queried by iteratively sending 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 chance that the query will be answered.

If instead 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 relying on the ability of a few nodes to answer most queries. In the *Local Indices* technique (Section 4.3), nodes maintain very simple and small indices over other nodes' data. Queries are then processed by a smaller set of nodes than current techniques, while yielding as many results.

The rest of this paper is organized as follows. In Section 2, we give an overview of related work in the area. Section 3 gives an overview of the search problem, presents metrics with which to evaluate techniques, and discusses the shortcomings of techniques used in today's systems. Then in Section 4, we present our three search techniques, describing in detail the different variations and policies within each technique. These techniques and variations are then evaluated in Section 6, using the experiments and analysis described in Section 5.

## 2 Related Work

The original motivation for much P2P research was early file-sharing P2P systems such as Gnutella [6], Napster [9], Freenet [5], and Morpheus [8]. All four of these systems are file-sharing systems that do not provide availability guarantees. 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 then connect to each other to form a pure P2P network. Our techniques are applicable to the P2P network of super-peers.

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), where nodes index the content of other nodes in the system, thereby allowing queries to be processed over a large fraction of all content while actually visiting just a small number of nodes. However, the routing policy for Query messages differ between our work and theirs, resulting different performance and cost tradeoffs.

References [3] and [10] propose that each node maintain metadata that can provide "hints" as to which nodes contain data that can answer the current query. Query messages are routed by nodes making local decisions based on these hints. Hints in [3] are formed by building *summaries* of the content that is reachable via each connection a node has. Hints in [10] are formed by learning user preferences. By observing the behavior of the user – for example, noting which nodes the user chose to download from – a user's client can learn which nodes the user believes to be an *authority* on certain topics. Future queries on those topics can subsequently be routed to these authority nodes.

Search techniques for systems with strong guarantees on availability include Chord [16], CAN [11], Pastry [12], and Tapestry [19]. These four techniques are quite similar in concept, but differ slightly in algorithmic and implementation details. In Chord, Pastry and Tapestry, nodes are assigned a numerical identifier, while in CAN, nodes are assigned regions in a d-dimensional identifier space. A node is then responsible for owning objects, or pointers to objects, whose identifiers map to the node's identifier or region. Nodes also form connections based on the properties of their identifiers. With these deliberately formed connections and intelligent routing, a system 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 the performance of the proposed techniques using extensive measurements gathered over queries in the Gnutella network. Extensive measurements over the Gnutella network are also described in [15], but these measurements focus on the hardware characteristics of the peers (e.g., bandwidth, latency of the peer connections), whereas ours focus on query behavior. Reference [2] describes some measurements taken by passive observation of messages passing through the network, much like our general statistics described in Section 5.1. However, [2] does not actively interact with other peers in the network to gather more detailed data, as [15] and ourselves do.

## 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) to the users. 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. For example, the collection could be 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. Nodes process queries and produce results (such as pointers to files) individually, and the total result set for a query is the bag union of results from every node that processes the query.

We can view a P2P overlay network as an undirected graph, where the vertices correspond to nodes in the network, and the edges correspond to open connections maintained between the nodes. Two nodes maintaining an open connection between themselves are known as *neighbors*. Messages may be transferred in either direction along the edges. For a message to travel from node $A$ to node $B$, it must travel along a path in the graph. The length of this traveled path is known as the number of *hops* taken by the message. Similarly, 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 node becomes the *source* of the query. A source node $S$ may send the query message to any number of its neighbors. The routing policy in use determines to how many neighbors, and to which neighbors, the query is sent. When a node receives a Query message, it will process the query over its local collection. If any results are found at that node, the node will send a single Response message back to the query source. In some systems, such as Gnutella, the address of the query source is unknown to the replying node. In this case, the replying node sends the Response message along the reverse path traveled by the query message. In other systems, the replying node may know the address the query source, and will open a temporary connection with the source to transfer the

Response message. Although the first scheme uses more aggregate bandwidth than the second, it provides anonymity for the query source, and it prevents the query source from being bombarded by connection requests. In this paper, we assume the first scheme is being used.

When a node receives a Query message, it must also decide whether to forward the message to other neighbors, or to drop it. Again, the routing policy determines whether to forward the query, and to whom the query is forwarded.

## 3.1 Metrics

In the rest of the paper, we will describe techniques to address the efficiency of queries. In order to evaluate the effectiveness of these techniques, we must first define some metrics.

**Cost.** When a query message is propagated through the network, it passes through a number of nodes. Each of these nodes spends processing resources (i.e., cycles) on behalf of the query, whether it be to forward the query, to process the query, or to forward responses back to the source. Similarly, each node uses bandwidth resources on behalf of the query, as it sends and receives Query and Response messages. The main cost of queries can therefore be described in terms of bandwidth and processing cost.

Now, the cost of a given query $Q$ is not incurred at any single node in the network. It therefore makes sense to discuss costs in aggregate. Furthermore, we cannot evaluate a policy based on the performance of any single query, so instead we must measure the average aggregate cost incurred by a set of queries $Q_{rep}$, where $Q_{rep}$ is some representative set of real queries submitted to the network. Our two cost metrics are:

- **Average Aggregate Bandwidth**: the average, over a set of representative queries $Q_{rep}$, of the aggregate bandwidth consumed (in bytes) over every edge in the network on behalf of each query.
- **Average Aggregate Processing Cost**: the average, over a set of representative queries $Q_{rep}$, of the aggregate processing power consumed at every node in the network on behalf of each query.

It is not possible to directly measure these metrics because they are aggregate metrics over the entire network. Instead, we use analysis (based on measured costs of individual actions) to estimate the values for these two metrics. Please refer to Section 5.2 for a detailed description of the calculations.

**Quality of Results.** Although we seek to reduce the resources used to process the query, we must nevertheless ensure that our techniques do not degrade the user experience. Quality of results can be measured in a number of ways; we use the following metrics:

- **Number of results**: the size of the total result set.
- **Satisfaction** of the query: 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, Gnutella and Morpheus) will notify the user of

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, in terms of satisfaction.

- **Time to Satisfaction**: Another important aspect of the user experience is how long the user must wait for results to arrive. Response times tend to be slow in pure P2P networks, since the query and its responses travel through several hops in the network. Time to satisfaction is simply 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 the cost and quality metrics. In the following sections, we will apply these metrics to experimental results to determine the effectiveness of our proposed techniques.

## 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) over the network with depth limit $D$, where $D$ is the system-wide maximum time-to-live of a message, measured in hops. A source node $S$ sends the Query message with a TTL of $D$ to a depth of 1, to all its neighbors. Each node at depth 1 processes the query, sends any results to $S$, and then forwards the query to all of their neighbors, to a depth of 2. This process continues until depth $D$ is reached (i.e., when the message reaches its maximum TTL), at which point the message is dropped.
- **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, and not by satisfaction, 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 much bandwidth and processing power because, as we have argued 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 (i.e., the query message dropped) 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.

## 4 Broadcast Policies

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.

### 4.1 Iterative Deepening

In systems where satisfaction is the metric of choice, a good technique is *iterative deepening*. Iterative deepening is a well-known search technique used in other contexts, such as search over state space in artificial intelligence [14].

Over the iterations of the iterative deepening technique, multiple breadth-first searches are initiated with successively larger depth limits, until either the query is satisfied, or the maximum depth limit $D$ has been reached. Because the number of nodes at each depth grows exponentially (Section 5.1.1), the cost of processing the query multiple times at small depths is small, compared to processing query once at a large depth. In addition, if we can satisfy the query at a depth less than $D$, then we can use much fewer resources than a single BFS of depth $D$. Indeed, our experiments show (Section 5.1.1) that for a client with 8 neighbors in the Gnutella network where $D = 7$, 70% of all queries can be satisfied at a depth less than $D$, for $Z = 20$.

The iterative deepening technique is implemented as follows: first, 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 at depth $c$. Our policy is therefore $P = \{a, b, c\}$. Note that in order for iterative deepening to have the same performance as a BFS of depth $D$, in terms of satisfaction, the last depth in the policy must be set to $D$. In addition to a policy, a waiting period $W$ must also be specified. $W$ is the time between successive iterations in the policy, explained in further detail below.

Under the policy $P = \{a, b, c\}$, a source node $S$ first initiates a BFS of depth $a$ by sending out a Query message with TTL$=a$ to all its neighbors. Once a node at depth $a$ receives and processes the message, instead of dropping it, the node will store the message temporarily. The query therefore becomes "frozen" at all nodes $a$ hops from the source $S$. Nodes at depth $a$ are known as the *frontier* of the search. Meanwhile, $S$ receives Response messages from nodes that have processed the query so far. After waiting for a time period $W$, if $S$ finds that the query has already been satisfied, then it does nothing. Otherwise, if the query is not yet satisfied, $S$ will start the next iteration, initiating a BFS of depth $b$.

To initiate the next BFS, $S$ could send out another Query message with TTL$=b$, meaning every node within $a$ hops will process the query a second time. To prevent this wasted effort, however, $S$ will instead send out a *Resend* message 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 the frontier of the search, it will drop the Resend message and "unfreeze" the corresponding query by forwarding the Query message (with a TTL of $b - a$) to its neighbors. To identify queries with

Resend messages, every query is assigned a system-wide “almost unique” identifier. Gnutella already assigns such an identifier to all queries. The Resend message will contain the identifier of the query it is representing, and nodes at the frontier of a search 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 a particular application, then the iterative deepening technique may not be applicable because of the time taken by multiple iterations. A better strategy that still reduces cost 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 query source send Query messages to just a subset of its neighbors, but selecting neighbors through which nodes with many quality results may be reached. 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 will maintain statistics on its neighbors. These statistics can be very simple, such as the number of results that were received through the 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. Sample heuristics include:

- 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) since our client first connected to the neighbor. A high message count implies that this neighbor is stable, since we have been connected to the neighbor for a long time, 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.

By sending the query to a small subset of neighbors, we will likely reduce the number of nodes that receive the query by a significant amount, thereby reducing costs incurred by the query. On the other hand, by selecting neighbors we believe will produce many results, we can maintain quality of results to a large degree, even though fewer nodes are visited. 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, a node $n$ maintains an index over the data of each node within $r$ hops of itself, where $r$ is a **system-wide** variable known as the *radius* of the index ($r = 0$ is the degenerate case, where a node only indexes metadata over its own collection). When a node receives a Query message, it can then process the query on behalf of every node within $r$ hops of itself. In this way, the collections of many nodes can be searched by processing the query at few nodes, thereby maintaining a high satisfaction rate and number of results while keeping costs low.

Previously, we mentioned that data placement strategies may face a barrier to implementation in loosely controlled systems such as Gnutella, because users will not want to devote large amounts of storage to store other users' data or metadata. However, when $r$ is small, the amount of metadata a node must index is also quite small – on the order of 50 KB – independent of the total size of the network. 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 more details on space requirements.

Before we describe details of this technique, let us first point out that the concept of indexing other nodes' data has been used in several systems to date. We mentioned earlier that the Morpheus P2P system uses "super-peers" – nodes that index the collections of their clients and answer queries on their behalf, while the clients never answer any queries. Napster can be seen as using a variant of the super-peer technique, where a server containing a centralized index over every node's data is the single super-peer, and all other nodes are clients. Our Local Indices technique differs from these other techniques because all nodes under this technique still have equal roles and capabilities, i.e., it is still pure P2P. Furthermore, our technique is malleable in that we can adjust $r$ to study both systems with small indices (small $r$) such as Morpheus, and systems with large indices (large $r$) such as Napster, and many possibilities in between. As a result, it is well-suited to understanding the benefits and tradeoffs of indexing.

The Local Indices technique works as follows: a 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, since 2 is not in the policy $P$, 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. Also, because depth 5 is the last depth in $P$, these nodes will then drop the Query message. We assume that in a given system, all queries use the same policy.

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. Note also that in order for the Local Indices technique to have equal performance as a BFS of depth $D$, in terms of number of results and satisfaction, the last depth in the Local Indices policy must be set to $D - r$.

To create and maintain the indices at each node, extra steps must be taken whenever a node joins or leaves the network, and whenever a user updates his local collection (e.g., inserts a file). When a node $X$ joins the network, it sends a Join message with a TTL of $r$, which will be received by all nodes within $r$ hops. The Join message contains sufficient metadata over $X$'s collection for another node to answer queries on node $X$'s behalf. When a node receives the Join message from $X$, it will in return 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, it may introduce a path of length $r$ or less between two other nodes, where no such path previously existed. In this case, the two nodes can be made aware of their new connection in a number of ways without introducing additional messages. For example, in the Gnutella network, nodes periodically send out Ping messages to all nodes within a depth $D$, and every node receiving the Ping should respond with a Pong message. This mechanism is used for node discovery. If a node $A$ receives a Pong message from another node $B$ that is within $r$ hops, but whose metadata is not currently indexed at $A$, node $A$ may send a Join message directly to $B$, and in return, $B$ may send a Join message directly to $A$.

When a node leaves the network or dies, other nodes that index the leaving node's collection will remove the appropriate metadata after a timeout. Timeouts can also be implemented without introducing new messages. For example, if a node does not respond to Ping messages several times, it may be assumed that this node has died or left the network.

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 element (e.g., file or record), and indicates whether the element was inserted, deleted, or updated. 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.

## 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. To evaluate our techniques, we need to answer questions such as:

- When our Gnutella client submits a query, how many results are received?
- When our client submits a query, what is the time to satisfaction under the iterative deepening technique?

Some of these questions can be answered through direct experiments; for example, we can answer the first question simply by having our client parse and log each Response message it gets in response to its query. Other questions need

| 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 (Appendix A) | 30% |

Table 1: General Statistics

to answered indirectly; for example, we could not force all Gnutella nodes to use the iterative deepening technique, in order to answer our second question. Instead, we collect performance data for queries under the BFS technique, and combine this data using analysis to estimate what the time to satisfaction would have been under iterative deepening. We therefore evaluate our techniques using a combination of experiments to gather raw data from the Gnutella network, and analysis to estimate the cost and performance of queries 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. For example, Gnutella Pong messages contain the IP address of the node that originated the message, as well as the number of files stored at the node. By extracting this number from all Pong messages that pass by, we can determine the distribution of collection sizes in the network. 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.

### 5.1.1 Iterative Deepening

From the representative set $Q_{rep}$, our client submitted each query $Q$ to the Gnutella network $D$ times (spread over time), where $D = 7$ is the maximum TTL allowed in Gnutella. Each time the query was submitted we increment its TTL by 1, so that we submit the query once for each TTL between 1 and $D$. For each Query message submitted, we logged every Response message that arrived 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

| 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 |

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

| Symbol | Description |
|---|---|
| $T(Q,Z,W,P)$ | 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 3: Symbols and function names of data estimated from logs for iterative deepening

message was received).

- The IP address from which the Response message came.
- The individual results contained in the Response 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 (and other information). We sent a Ping message immediately before every second query. After a Ping message was sent, for the duration of the next two queries (i.e., 4 minutes), we logged the following information for all Pong messages received:

- 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 received for each query, summarized in 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 record. Binary function $S(Q,n,Z)$ returns true if query $Q$ was satisfied by the results received from nodes within $n$ hops.

In addition to the values we can extract directly from our logs, we also estimate several values that could not be directly observed by our client, listed in Table 3. Though these values could not be directly observed, they are nevertheless carefully calculated from observed quantities. For example, $N(Q,n)$ is calculated using three sets of observed IP addresses: (1) the IP addresses of nodes $n$ hops away that responded to the latest Ping sent out before query $Q$ was submitted, (2) the IP addresses of nodes $n$ hops away that responded to the earliest Ping sent out after query $Q$ was submitted, and (3) the IP addresses of nodes $n$ hops away that responded to query $Q$. Details on how $N(Q,n)$ and $C(Q,n)$ are estimated can be found in Appendix A, while Appendix B describes how we estimate $T(Q,Z,W,P)$, the time at which query $Q$ is satisfied under policy $P$, with waiting time $W$.

### 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

| 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 $y$ |

Table 4: Symbols and function names of data extracted from logs for Directed BFS

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. However, now we differentiate the performance of the query when sent to different neighbors. The data we extract from the logs is listed in Table 4. 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 5. These values are estimated in the same manner as described in Appendix A.

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 in heuristically selecting which neighbor to forward the query to. Please see Appendix C for a list of statistics recorded for each neighbor.

#### 5.1.3 Local Indices

No separate experiment was run to gather data for this technique. Instead, we use the data from the Iterative Deepening and Directed BFS experiments.

## 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 summarize our calculations for average aggregate bandwidth and average aggregate processing cost. Calculations for time to satisfaction can be found in Appendix B.

#### 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

| Symbol | Description |
|---|---|
| $N(Q,n,y)$ | 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 5: Symbols and function names of data estimated from logs for Directed BFS

| 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 Appendix D | Size of a full Response message under Local Indices |
| $f$ | 24560 | Size of a Join message |
| $g$ | 152 | Size of an Update message |

Table 6: Sizes of messages

2 bytes to describe the minimum bandwidth the query source requires any responding node to have (which in our experiments is set to 0). A Gnutella header holds application-specific information such as the ID of the message and the message type. Headers in Gnutella are 22 bytes, and TCP/IP and Ethernet headers are 58 bytes. The query string is $L(Q)$ bytes (Table 2). Total message size is therefore 82 + $L(Q)$ bytes. Table 6 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:

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

We can understand the above formula by thinking of the query as it travels from one depth to the next – that is, when it travels from $n$ hops away to $n+1$ hops away. First, we know that the query will reach every level from 1 to $D$, which is why we have the summation from $n = 1$ to $D$. Here, $n$ represents the depth, or number of hops the query has taken. Recall that $D$ represents the maximum TTL of a message.

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.

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 Appendix D.

| 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 Appendix E | Cost of transferring a Response message under Local Indices |
| $z(Q,r)$ | see discussion in Appendix E | 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 7: Costs of actions

### 5.2.2 Processing Cost

Calculating average aggregate processing power consumed under a particular technique is done much in the same way as calculating average aggregate bandwidth. First, we estimate how much processing power each type of action requires. We then go through the data in the query logs and use analysis to estimate how much aggregate processing power each query consumed. Because of space limitations, we do not present formulae for calculating average aggregate processing cost. Please see Appendix E for the formulae and their derivations. Here, we briefly describe how processing power costs are measured.

Table 7 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. Rather than calculating processing cost in terms of cycles, we express the cost of actions in terms of coarser units, and use the relative cost of actions in units to compare processing power consumed by a query. Costs were estimated by running each type of action on a 930 MHz processor (Pentium III, running Linux kernel version 2.2) and recording CPU time spent on each action. While CPU time will vary between machines, the relative cost of actions should remain roughly the same. We define the base unit as the cost of transferring a Resend message, which is roughly 7300 cycles.

In Table 7, the cost of transferring a message of any type (Query, Resend, Join, Update) covers the cost of sending the message from the application layer in one node, and receiving the message at the application layer of another node. This cost is a linear function in the size of the message, so the first few actions in Table 7 are calculated by simply plugging the size of each message type, estimated in Table 6, into this function. Note that the cost of transferring a Response message is broken into two sub-costs: the base cost of sending a single Response message containing only the headers, and the additional cost of sending a Response message when another result record is appended to the message. The cost of processing a query also consists of a fixed component and a linear component in the number of results discovered. The "per result" cost is the additional cost added for each result discovered, while the "base" cost

| 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 |

Table 8: Definition of Symbols

is the fixed overhead of initiating a search.

## 5.3 Averaging Results

Now that we can calculate or directly observe the performance of a single query $Q$ in terms of each metric, the performance of a technique is measured by averaging the performance of every query $Q \in Q_{rep}$. For example, average aggregate bandwidth for a breadth-first search is:

$$AvgAggBW_{bfs} = \frac{1}{|Q_{rep}|} \cdot \sum_{Q \in Q_{rep}} BW_{bfs} \tag{2}$$

## 6 Experiments

In this section, we present the results of our experiments and analysis. We discuss the performance tradeoffs between different policies, and form recommendations for existing systems such as Gnutella. As a convenience to the reader, some symbols defined in previous sections are re-defined in Table 8.

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.

## 6.1 Iterative Deepening

For the sake of comparison, we evaluate only those iterative deepening policies where the last depth is set to $D$ (see Section 4.1 for an explanation). To understand the tradeoffs between policies of different lengths, and policies that begin at different depths, we choose the following subset of policies to study:

$$\begin{aligned} P &= \{P_d = \{d, d+1, ...., D\}, \text{ for } d = 1, 2, ..., D\} \\ &= \{\{1, 2, ..., D\}, \{2, 3, ..., D\}, ..., \{D-1, D\}, \{D\}\}. \end{aligned} \tag{3}$$

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 the first experiment, our client ran for a period of 4 days, submitting twice each query in our representative set $Q_{rep}$. Throughout this period, the client maintained 8 neighbors. When analyzing the data, we defined the desired

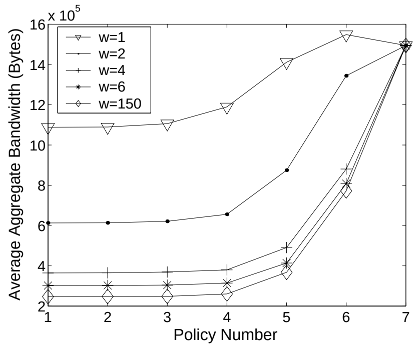

Figure 1: Bandwidth consumption for various iterative deepening policies

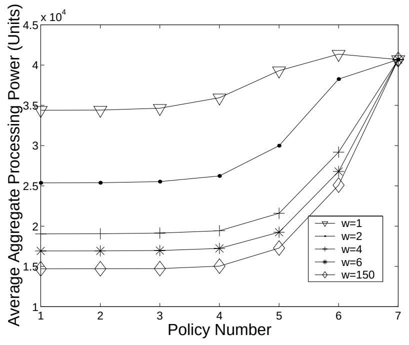

Figure 2: Processing cost for various iterative deepening policies

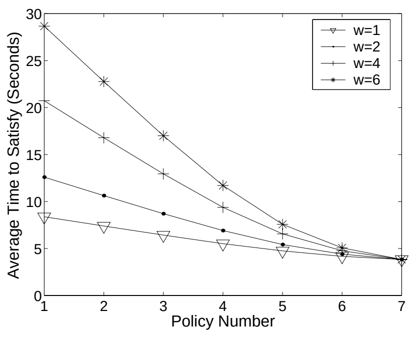

Figure 3: Time to satisfaction for various iterative deepening policies

number of results $Z = 50$. Later in this section, we will show what happens when $Z$ is varied, and when the number of neighbors of our client changes.

We note that our client ran over an 10 Mb Ethernet connection. We wanted to ensure that the connection was stable (unlike dialup modems, and sometimes even cable modem and DSL), and we wanted to ensure that the connection did not saturate. Both of these characteristics were needed to ensure that our client would not die and interrupt the ongoing experiments. 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.** Figures 1 and 2 show the cost of each policy, for each value of $W$, in terms of average aggregate bandwidth and average aggregate processing cost, respectively. Along the x-axis, we vary $d$, the policy number. Immediately obvious in these figures are the cost savings. 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. The curves in Figures 1 and 2 have roughly the same shape, for the same reasons. Let us therefore focus on bandwidth consumption, in Figure 1. First, notice that the average aggregate bandwidth for $d = 7$ is the same, regardless of $W$ (also seen in Figure 3). Since the waiting time $W$ only comes into play between iterations, it does not affect $P_7 = \{7\}$, which has only a single iteration.

Next, notice that as $d$ increases, the bandwidth consumption of policy $P_d$ increases as well. The larger $d$ is, the more likely the policy will waste bandwidth by sending the query out 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 the Query message, and transferring Response messages back to the source. Hence, as $d$ increases, bandwidth consumption increases as well, giving $P_7$ the worst bandwidth performance.

Now, notice that as $W$ decreases, bandwidth usage increases. If $W$ is small, there is a higher likelihood that the client 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; hence, bandwidth usage increases as $W$ decreases.

Note also that for $W = 1$, when $d$ is large, bandwidth usage actually decreases as $d$ increases. The reason for this anomaly is the cost of Resend messages. As we just explained, when $W$ is small, the client tends to overshoot the goal. This effect is so strong for $W = 1$ that almost every query under $P_6$ will eventually reach depth 7. Therefore, roughly the same amount of bandwidth is used for Query and Response messages for $P_6$ and $P_7$, but policy $P_6$ also has the additional cost of Resend messages for almost every query. Aggregate bandwidth consumption for $P_6$ is therefore greater than that of $P_7$.

**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 3 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 of these cases, the time to satisfaction will increase. Note, however, that delay is often caused by saturated connections or thrashing nodes. If all nodes in Gnutella were to use 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 3, as $d$ increases 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 3, we see that several combinations of $d$ and $W$ result in this time to satisfy, e.g., $d = 4$ and $W = 4$, or $d = 5$ and $W = 6$. Looking at Figures 1 and 2, 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.

**Variations.** Performance of the iterative deepening technique and its various policies rely on the value chosen for $Z$. In addition, performance depends on the number of nodes that process each query, which in turn is determined locally

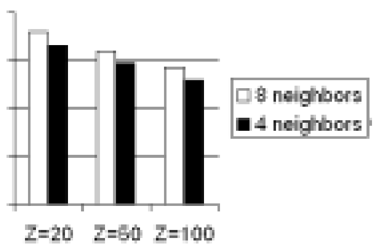

Figure 4: Probability of satisfaction for different $Z$

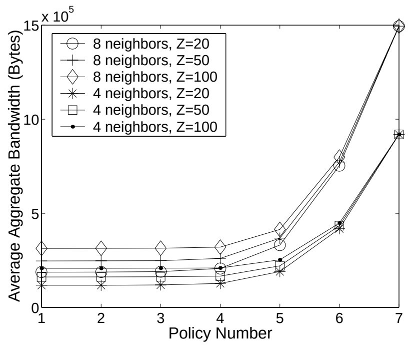

Figure 5: Bandwidth consumption for iterative deepening at $W = 150$

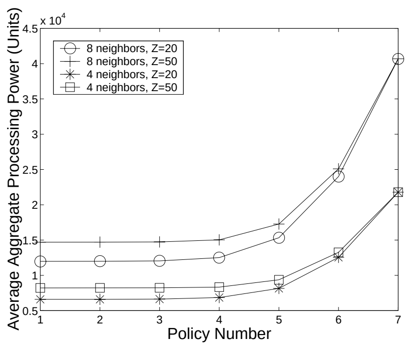

Figure 6: Processing cost for iterative deepening at $W = 150$

by the number of neighbors (degree) that a client maintains. To see the effect these two factors have on the iterative deepening technique, we ran our data collection experiment twice, first where our client maintained 8 neighbors, then where our client maintained 4 neighbors. Over each data set, we then ran analysis three times, first with $Z = 20$, then with $Z = 50$, then with $Z = 100$.

Figure 4 shows the performance of each variation in terms of satisfaction. Since choice of policy number and $W$ does not affect the satisfaction of iterative deepening, this figure represents all iterative deepening policies. As expected, when the definition of satisfying becomes more stringent (i.e., $Z$ increases), satisfaction decreases. However, it is encouraging to note that satisfaction does not drop very quickly as $Z$ increases. A value of $Z = 100$ requires that 5 times as many results need to be returned in order to satisfy the query, when compared to $Z = 20$. Yet, its probability of satisfying is only 15% less with 8 neighbors, and 14% less with 4 neighbors.

Also as expected, when the number of neighbors of our client decreases, the probability that a query is satisfied also decreases, since fewer neighbors generally translates to fewer number of results. However, we find it interesting that satisfaction with 4 neighbors is not much lower than satisfaction with 8 neighbors. Though we have half as many neighbors, our probability of satisfaction is only about 6% lower when we have four neighbors as opposed to 8. The reason for this effect is that with 8 neighbors, our client usually received far more results than was needed to satisfy the query. When the number of neighbors decreased to 4, the client received significantly fewer results, but in most cases it was still enough to satisfy the query.

The cost savings of having fewer neighbors, however, is quite significant. Figures 5 and 6 show the cost of the different variations for $W = 150$ in terms of average aggregate bandwidth and average aggregate processing cost, respectively. We chose $W = 150$ because it represents the best-case cost performance of these variations from our experimental data. Again along the x-axis we vary the policy number $d$. We see in Figure 5 that for all policy numbers, iterative deepening with 4 neighbors requires just 55% to 65% of the bandwidth required with 8 neighbors. Similarly, Figure 6 shows us that iterative deepening with 4 neighbors requires just 55% of the processing cost required with 8 neighbors. Though we do not show the cost of the policies for different values of $W$, we verify that the same

| 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 9: Heuristics used for Directed BFS

magnitude of savings occur for all $W$ we considered. Hence, decreasing the number of neighbors that the query is sent to might decrease satisfaction a little, but decrease the cost significantly. Larger values of $Z$ translate to more costly queries, because more results must be processed and returned before the query terminates.

In terms of making a recommendation for a real system, we first note that most nodes do not have many neighbors, so the option of sending the query to fewer neighbors does not exist for them. For the nodes that do have many neighbors, they would be doing a service to the network by decreasing the number of neighbors they send the query to, if they are the query source. However, it may be difficult to convince these highly connected nodes to be altruistic, especially since they already provide a service to the network by routing much traffic.

## 6.2 Directed BFS

We studied 8 heuristics for Directed BFS, listed in Table 9. The RAND heuristic, choosing a neighbor at random, is used as the baseline for comparison with other heuristics.

**Quality of Results.** Figure 7 shows the probability of satisfying, for the different heuristics, for different 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 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. We therefore see, as expected, that past performance is a good indicator of future performance. As with iterative deepening in Figure 4, increasing $Z$ decreases the satisfaction. We could not find an intuitive explanation for why the performance of <QLEN drops at $Z = 100$.

Figure 8 shows the time to satisfaction of the different heuristics. Under this metric, again most heuristics outperform the baseline RAND. The <TIME heuristic has the best performance (36% lower than the baseline), followed by >RES (20% lower than the baseline). Again, 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 the other heuristics, with the exception of <HOPS. Past work such as [1] show that in a power-law network such as Gnutella, sending the query to nodes with the highest degree should allow one to reach many nodes in the

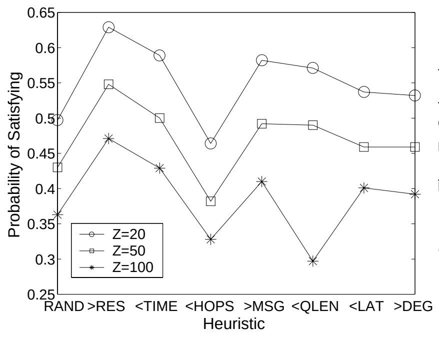

Figure 7: Probability of satisfaction for various Directed BFS policies

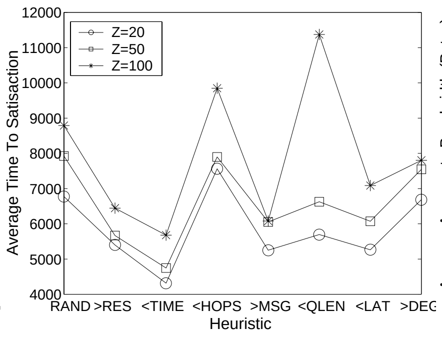

Figure 8: Time to satisfaction for various Directed BFS policies

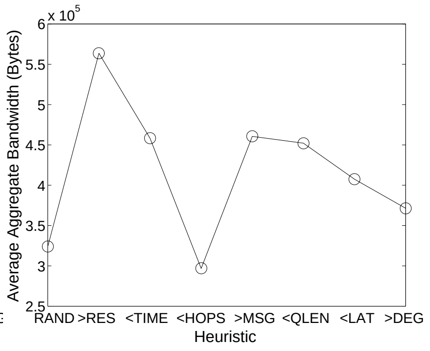

Figure 9: Bandwidth consumption for Directed BFS

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, the amount of time a neighbor has been alive, etc. It is possible that with this additional real-time information, their routing algorithm might be enhanced.

We were also initially surprised to see that <HOPS, sending the query to the neighbor that has returned results with the lowest number of hops, performed so poorly in terms of both satisfaction and time to satisfaction. On closer examination of our logs, we found that a low average number of hops does not imply that the neighbor is close to other 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. Though all results come from just a few hops away, only few results are returned through that neighbor. Since this heuristic returns few results, it also has the worst time to satisfaction. Though the average arrival time of results is lower with this heuristic than with others, the time at which the $Z$th result arrives for this heuristic is still higher than the time at which the $Z$th result arrives for other heuristics, especially if $Z$ is large.

**Cost.** Figures 9 and 10 show the cost of Directed BFS under each heuristic, in terms of average aggregate bandwidth and average aggregate processing cost, respectively. The cost of Directed BFS in unaffected by the value of $Z$, so the single curve in these figures 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 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

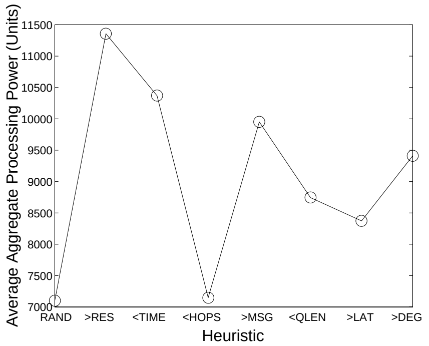

Figure 10: Processing cost for Directed BFS

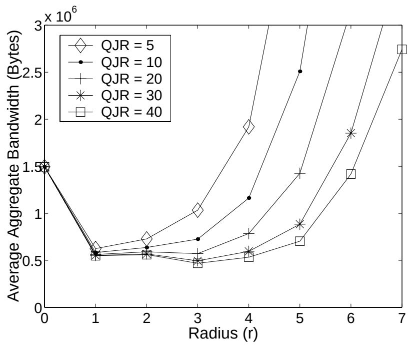

Figure 11: Bandwidth consumption for Local Indices

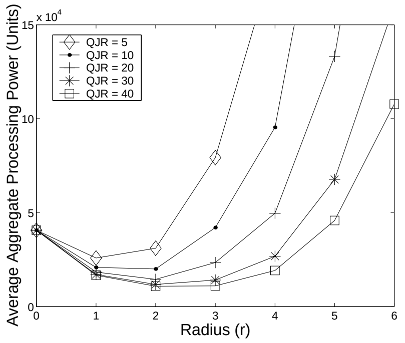

Figure 12: Processing cost for Local Indices

cost than BFS, and 65% less bandwidth.

Compared with iterative deepening, the strength of Directed BFS is time to satisfaction. Comparing Figures 3 and 8, 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. Furthermore, satisfaction for all iterative deepening policies is higher than satisfaction of any Directed BFS heuristic. Table 10 in Section 7 summarizes the comparisons between these two techniques.

## 6.3 Local Indices

To evaluate the Local Indices technique, we used data collected from the iterative deepening experiments described in Section 5.1.1, where the client was run with 8 neighbors. For the sake of comparison, we evaluate only those policies that have the same performance as a BFS of depth $D$, in terms of number of results and satisfaction. Hence we only consider policies where the last depth is set to $D - r$ (see Section 4.3 for an explanation). Since the number of nodes grows exponentially as distance from the source node increases, to minimize cost, we want the remaining depths in the policy to be as small as possible. If a query is being executed at depth $x$, then the previous depth in the policy need be at least $x - 2r - 1$ (processing at depth $x - 2r - 1$ covers all nodes up to depth $x - r - 1$, and processing at depth $x$ covers all nodes starting at $x - r$). As a result, for every possible radius $r$, we evaluate a single policy $P_r$, defined as $P_r = \{..., D - 3r - 1, D - r\}$. For clarity, Figure 13 lists these policies explicitly.

**Cost.** In the following cost evaluations, we vary `QueryJoinRatio` because cost is highly dependent on this parameter. The default value for `QueryJoinRatio` observed in Gnutella is roughly 10 [17]. Recall that `QueryJoin-Ratio` gives us the ratio of queries to joins/leaves in the network (i.e., roughly how many queries a user submits during a session). However, here we do not vary the value of `QueryUpdateRatio`, the ratio of queries to updates in the network, because we found that varying this parameter had the same types of effects as varying `QueryJoinRatio`. The default value of this parameter was obtained indirectly, because it was impossible to observe how frequently users

| 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 13: Policies for Local Indices

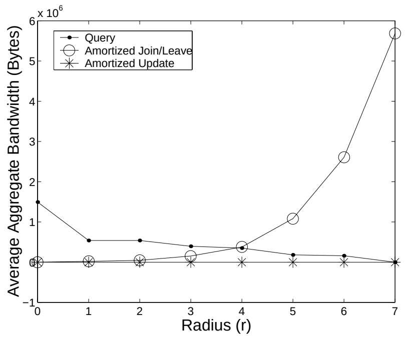

Figure 14: Comparison of Bandwidth Consumed by Queries and Join/Leaves

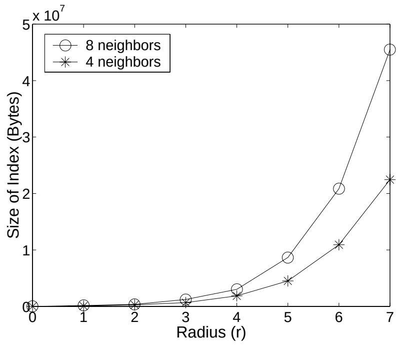

Figure 15: Size of the Index for different Radii(r)

updated their collections. We first assumed that most online updates come as a result of users downloading files from other users. We then use the ratio of queries to downloads observed in [18] for the OpenNap system as an estimate for Gnutella. Our estimate for `QueryUpdateRatio` is therefore 0.5, meaning that on average, a user updates 2 files for every query he submits.

Figures 11 and 12 give us the cost of the Local Indices policies for various values of `QueryJoinRatio` (QJR), in terms of average aggregate bandwidth and average aggregate processing cost, respectively. Along the x-axis, we vary the radius $r$. Recall that $r = 0$ is the degenerate case where a node keeps an index over its own collection only; $P_0$ is equivalent to a BFS of depth $D$.

Again, we can see huge cost savings from the Local Indices technique, especially as `QueryJoinRatio` increases. At `QueryJoinRatio` = 10, policy $P_1$ uses about $5.8 \cdot 10^5$ bytes of aggregate bandwidth, which is about 39% of what the default BFS scheme uses (i.e., $r = 0$), and it uses $2.1 \cdot 10^4$ units of processing power, which is about 51% of what the default scheme uses. When `QueryJoinRatio` = 100, only 28% of the bandwidth and 21% of the processing cost of the default scheme is required. We see from both figures that cost decreases as `QueryJoin-Ratio` increases. Because there are more queries for every join and leave in the network as `QueryJoinRatio` increases, the amortized cost of joins and leaves decreases. However, the cost for $r = 0$ is the same for all values of `QueryJoinRatio` because it is the degenerate case where a node keeps an index over its own collection only. At $r = 0$, the cost of joins, leaves and updates is zero.

In Figures 11 and 12, as $r$ increases, the cost of the policies decrease, and then increase again. To understand the reason for this behavior, we looked at the cost of queries, the amortized cost of joins and leaves, and the amortized cost of updates separately in Figure 14, for `QueryJoinRatio`=20. When $r$ is small, the cost of queries is quite large, and dominates over the cost of joins, leaves and updates. As $r$ increases, the cost of queries decreases, because fewer nodes need to process the query, and because messages need to travel fewer hops. However, the amortized cost of joins and leaves increases exponentially, such that when $r$ is large, this cost dominates over the cost of queries. The point at which total cost is minimized is somewhere in between the two extremes – in the case of Figure 14, it is at

$r = 1$. The amortized cost of updates is always a relatively small fraction of total cost.

While in some cases, relatively large $r$ have the best performance, we must keep in mind the storage requirements for the index at each node. Figure 15 shows us the size of the index that our node would have as a function of $r$, had we implemented Local Indices. The size requirement for a policy is simply 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. Since our node was relatively well-connected (average degree is roughly 2 in Gnutella), the numbers shown in Figure 15 are significantly larger than they would be for an average node.

Though the size of the index grows exponentially, it is still practical for all $r$ in our range. 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 the sake of the application. For $r = 1$, the size of our index would have been roughly 71 KB – an index so small that almost certainly no users would object to keeping.

For today's system with `QueryJoinRatio` = 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 for $r = 1$ is so small, it should not cause any barrier to implementation. In the future, when `QueryJoinRatio` increases, the best value for $r$ will also increase.

**Time to Satisfaction.** In the absence of data from a system that actually uses indices, calculating time to satisfaction for Local Indices is very hard. Thus, we only qualitatively discuss this metric for Local Indices when compared to BFS.

For the typical query, most results under a BFS with depth limit $D$ will come from $D$ hops away. In a Local Indices policy, most results will come from $D - r$ hops away. Because most Query and Response messages under Local Indices have $r$ fewer hops to travel, the time to forward messages to the outermost depth and back to the source will be shorter (for $r > 0$) than for BFS. However, because nodes have larger indices, processing the query should take more time. It is difficult to tell how these effects balance each other out as $r$ changes, but we believe that time to satisfaction changes *slowly* with $r$. Hence, time to satisfaction for Local Indices with small $r$ is comparable to time to satisfaction for BFS (as BFS is equivalent to Local Indices with $r = 0$).

## 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 of processing queries over the entire system, while maintaining equally high quality of results. Table 10 summarizes the performance tradeoffs among our proposed techniques, as compared to the BFS technique currently in use. Because

| Technique | Time to Satisfy | Probability of Satisfaction | Number of Results | Aggregate Bandwidth Cost | Aggregate Processing Cost |
|---|---|---|---|---|---|
| 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 10: 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 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] Eytan Adar and Bernardo 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. Submitted to ICDCS 2002, October 2001.

[4] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

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

[6] Gnutella website. http://gnutella.wego.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.musiccity.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. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (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. of the 18th ACM Symposium on Operating Systems Principles*, 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, Department of Computer Science and Engineering, July 2001. Available at http://www.cs.washington.edu/homes/tzoompy/publications/tr/uw/2001/uw-cse-01-06-02.

[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. Originally an online document available through DSS Clip2, now unavailable because the company has gone out of business. Please contact the primary author of this paper for a copy of the document.

[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. 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. Available at http://www.cs.berkeley.edu/ ravenben/publications/CSD-01-1141.pdf.

## A Calculations for $N(Q, n)$ and $C(Q, n)$

Under BFS, if a Ping and Query message are sent out at the same time from the same node with the same TTL, in theory, every node that processes the Query message should also process the Ping. Hence, the number of nodes that process the Query should be equal to the number of nodes that return a Pong message. In practice, however, not all nodes that process the query will also return a Pong message. For example, some nodes do not respond to Ping messages, to save bandwidth. To estimate $N(Q, n)$, therefore, we look at the following sets of IP addresses: (1) Set $A$: The IP addresses of nodes $n$ hops away that responded to the latest Ping sent out before query $Q$ was submitted, (2) Set $B$: The IP addresses of nodes $n$ hops away that responded to the earliest Ping sent out after query $Q$ was submitted, and (3) Set $C$: The IP addresses of nodes $n$ hops away that responded to query $Q$. All three sets of nodes generally overlap, and we know that the "correct" set of nodes contains some subset of $A$ and $B$, and completely contains $C$. Since nodes join and leave the network frequently, using the approximation $N(Q, n) = |A \cup B \cup C|$ will almost certainly be an over-estimation. However, using $|A \cup C|$, which is theoretically the correct choice, will almost certainly be an under-estimation since many nodes do not respond to Pings. Hence, we estimate $N(Q, n)$ to be $N(Q, n) = MAX(|A \cup C|, |B \cup C|)$.

An edge is said to be *redundant* at depth $n$ if a message traveling through this edge on its $n$th hop arrives at a node that has already seen the message – that is, if it closes a cycle on the $n$th hop. To estimate $C(Q, n)$, the number of redundant edges at depth $n$, we observed in our previous experiment that our client dropped roughly 30% of the Query messages it received (Table 1). A Query message is dropped if it has been seen by the client before – that is, if it has just traveled through a redundant edge. Given that our client dropped 30% of all Query messages it received, and assuming our client to be a model for typical behavior, we estimate 1 out of 3 of all messages in a breadth-first traversal are dropped due to redundant edges. In other words, if there are 60 nodes at depth 3, then 90 messages are sent from nodes at depth 2 to nodes at depth 3, and 30 of these messages are dropped. Since $N(Q, n)$ represents the number of messages that are not dropped, $C(Q, n) = N(Q, n)/2$.

## B Time To Satisfaction Calculations

**Iterative Deepening** Let $L_{(Q,n)}$ denote the ordered list of times at which result records are received for query $Q$ from nodes $n$ hops away. For example, $L_{(Q,n)}[i]$ is the time the $i$th result record was received from a node $n$ hops away. The time at which a result record is received is simply the time elapsed from when the query is first submitted, to when the response message containing that record was received. From our logs, we can directly extract $L_{(Q,n)}$ for each query $Q$, for each value of $n$ from 1 to $D$.

Now let us define $L^t_{(Q,n)}$ to be a list of the same length as $L_{(Q,n)}$, and where $L^t_{(Q,n)}[i] = L_{(Q,n)}[i] + t$. That is, we increment every element in $L_{(Q,n)}$ by $t$. Let us define function `merge(L,K)` over two ordered lists $L$ and $K$ to return a single ordered list formed by merging the elements of list $L$ and $K$. Finally, let us define function $I(n, P)$ to return the iteration on which depth $n$ will be reached, under policy $P$. For example, if $P = \{2, 5, 7\}$, then $I(1, P) = 1$, $I(2, P) = 1$, $I(3, P) = 2$, etc.

We can now calculate the time to satisfaction for a query $Q$ by the following procedure:

```
$RT(Q, P)$
(1)    $T = L_(Q, 1)$
(2)    for $n = 2$ to $D$
(3)        $T =$ merge$(T, L_{(Q,n)}^{(I(n,P)-1)\cdot W})$
(4)    return $T[Z]$
```

Let us assume query $Q$ does not end before reaching depth $n$ under policy $P$. In this case, then nodes at depth $n$ will receive the query on the $I(n, P)$th iteration, meaning the nodes would not receive the query until $(I(n, P)-1)\cdot W$ time units after th query was first submitted. We therefore estimate the times at which these results arrive at the source node to be $L_{(Q,n)}^{(I(n,P)-1)\cdot W}$.

Let us assume that query $Q$ is not satisfied until the last iteration. Then, the response times at which each result, for all depths, are received is given by $T$, calculated in lines $1 - 3$ in the above procedure. The times at which the query is satisfied is the time at which the $Z$th result is received, which is $T[Z]$.

Now, assume query $Q$ is satisfied at some iteration $i$ that is not the last iteration. Let $x$ be the depth of the iteration $i + 1$. All depths $d$ such that $d \geq x$ will not be reached by query $Q$. Will merging $L_{(Q,d)}^{(I(d,P)-1)\cdot W}$ into $T$ for $d \geq x$ therefore give us an incorrect result? Since $Q$ is satisfied during iteration $i$, we know that it satisfied at some time prior to $(I(n, P) - 1) \cdot W$, by definition (otherwise, iteration $i + 1$ would have been initiated). Hence, there are at least $Z$ elements in $T$ which are less than $(I(n, P) - 1) \cdot W$. Since merging $L_{(Q,d)}^{(I(d,P)-1)\cdot W}$ into $T$ for $d \geq x$ will only add elements into $T$ that are greater or equal to $(I(n, P) - 1) \cdot W$, the value of $T[Z]$ is unaffected by the merging of these lists.

**Directed BFS** Response time for a query $Q$ is simply $T(Q, Z, y)$, a value that we can directly observe.

**Local Indices** In the absence of data from a system that actually uses indices, calculating time to satisfaction for Local Indices requires much guess work. We felt that rough guesses on this metric would not be valuable, so we omit this metric when evaluating Local Indices. For a qualitative discussion of time to satisfaction under this technique, please see Section 6.3.

## C Neighbor Statistics for Directed BFS Experiments

- Number of results for previous queries from this neighbor, with a maximum query history of 10
- Average TTL of results for previous queries from this neighbor, with a maximum query history of 10
- Average response time of results for previous queries from this neighbor, with a maximum query history of 10
- Number of messages (all types) received from this neighbor, since our client first connected to the neighbor

- Number of messages (all types) sent to this neighbor, since our client first connected to the neighbor
- Number of messages (all types) dropped from this neighbor, since our client first connected to the neighbor
- Size of the message queue to this neighbor, at the time the query is submitted
- Latency of pong responses from this neighbor, at the time the query is submitted
- $N(Q, 1, y)$: an estimate of the degree of the neighbor immediately before query $Q$ is submitted

## D Aggregate Bandwidth Calculations

**Iterative Deepening** To calculate the aggregate bandwidth consumed under the iterative deepening technique, we must take into account the cost of sending Resend messages, as well as the possibility that a query may terminate before reaching the maximum depth $D$.

First, let $P$ be the iterative deepening policy we are evaluating. Recall that policies are defined as an ordered list of values, where the $i$th value tells us the depth at which the $i$th iteration ends, or freezes. We say a particular depth $d$ is a "policy level" if $d \in P$.

Let us define following indicator functions:

$$I_1(Q,n,W,P) = \begin{cases} 1 & \text{if } n \text{ is a policy level,} \\ & \text{and } Q \text{ does } \textit{not} \text{ end} \\ & \text{at depth } n \\ 0 & \text{otherwise} \end{cases} \quad I_2(Q,n,W,P) = \begin{cases} 1 & \text{if } Q \text{ does} \\ & \textit{not} \text{ end} \\ & \textit{before} \text{ level } n \\ 0 & \text{otherwise} \end{cases} \quad I_3(n) = \begin{cases} 1 & \text{if } n < D \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

For $I_1$, a query $Q$ is said to "end" at depth $n$ is $n$ is a policy level, and the query source determines that $Q$ has been satisfied after the query has reached a depth of $n$ but, before the time limit has expired. Recall that the time allowed between iterations is $W$ time units. Hence, if $n$ is the $i$th value in policy $P$, query $Q$ "ends" at depth $n$ if $T(Q, Z, W, P) < i \cdot W$. In terms of $I_1$, $I_2(Q, n, W, P) = 0$ if and only if $\exists k$ such that $h(Q, k, W, P) = 1$, for $k < n$ – that is, iff query $Q$ has ended at some previous policy level. Depth $n$ does not need to be a policy level. Function $I_3$ is self-explanatory.

Now, the formula for calculating aggregate bandwidth of query $Q$ with Iterative Deepening is as follows:

$$\begin{aligned} &BW_{id}(Q,W,P) = \\ &\quad \sum_{n=1}^{D} I_2(Q,n,W,P) \cdot \Big(a(Q) \cdot \big(N(Q,n) + C(Q,n)\big) + n \cdot \big(c \cdot R(Q,n) + d \cdot M(Q,n)\big)\Big) \\ &\quad + \sum_{x \in P} \Big(I_1(Q,x,W,P) \cdot I_2(Q,x,W,P) \cdot I_3(x) \cdot \sum_{n=1}^{x} b \cdot \big(N(Q,n) + C(Q,n)\big)\Big) \end{aligned} \tag{5}$$

The first summation gives us the bandwidth consumed by sending the query to level $n$, and receiving response messages from level $n$; it is almost identical to the summation in Equation 1. If the query ends before reaching level $n$, however, then it will not be sent to level $n$ nor receive results from level $n$. So in the above equation, we multiply the terms in the first summation by the indicator function $I_2(Q, n)$.

The second summation gives us the bandwidth consumed by sending out Resend messages. Resend messages are sent to depth $n$ given three conditions: (1) $x$ is a policy level, (2) query $Q$ is not satisfied at or before depth $x$ within the time limit, and (3) $x < D$, since the Query message cannot reach a depth greater than $D$. The first condition is satisfied because we only sum over those levels $x$ such that $x \in P$. The second condition is satisfied by multiplying the terms in the summation by $I_1(Q, x, W, P) \cdot I_2(Q, x, W, P)$, which equals 0 iff the query is satisfied at or before depth $x$. Finally, the third condition is satisfied by multiplying the terms by $I_3(x)$, which equals 0 if $n = D$.

**Directed BFS** Calculating bandwidth for Directed BFS is almost identical to calculating bandwidth for regular BFS, except that now before each query, we must first determine which neighbor would have been selected by the heuristic we are evaluating. Let us define the function $Y(Q, h)$ to return the neighbor that a query $Q$ is submitted to, under heuristic $h$. For each query $Q$ and heuristic $h$ that we consider, we determine $Y(Q, h)$ from our logs by looking at the statistics on each neighbor that were gathered before the query was submitted. For example, say right before query $Q$ is submitted, we find that neighbor A has a queue length of 10, neighbor B has a queue length of 5, and neighbor C has a queue length of 11. If heuristic $h$ is to choose the neighbor with the shortest message queue, we define $Y(Q, h)$ to return neighbor B.

We can now calculate the aggregate bandwidth of this query, under a heuristic $h$, simply by using Equation 1, but replacing the functions $M(Q, n)$, $R(Q, n)$, and $N(Q, n)$, with the analogous functions $M(Q, n, y)$, $R(Q, n, y)$, and $N(Q, n, y)$:

$$BW_{dbfs}(Q, h) = \sum_{n=1}^{D} \left( a(Q) \cdot \Big( N(Q, n, y) + C(Q, n, y) \Big) + n \cdot \Big( c \cdot R(Q, n, y) + d \cdot M(Q, n, y) \Big) \right) \tag{6}$$

where $y = Y(Q, s)$.

**Local Indices** Several new message types are introduced for the Local Indices technique. First, a Join message contains a Gnutella header, and the metadata over one node's collection. In Gnutella, metadata consists of the names of the files in the user's collection. Since the average Gnutella user has 340 files and the average file metadata is roughly 72 bytes long (Table 1), the average size of a Join message is 24560 bytes. An Update message contains a single file metadata, along with headers, totaling 152 bytes.

The size of a Response message is a function of the query $Q$, and the radius $r$. Since each node contains an index over all nodes within $r$ hops, a Response message from a node $n$ will contain results that come from the collections of all nodes within $r$ hops from $n$. Naturally, the number of results depends on $Q$, and as $r$ increases, the number of results generally increase as well. To estimate how many results will be in an average Response message for a given $Q$ and $r$, we use the number of results our own node received from nodes within $r$ hops, and scaled this value appropriately to reflect the fact that our node had $x$ neighbors at the time $Q$ was submitted, but the average degree of a node is roughly 2.

In addition to result records, a Response message contains a special header for every node whose results are included in the response message. For example, if node $A$ indexes the libraries of nodes $B$, $C$ and $D$, and node $A$ returns a response message that contains results from nodes $B$ and $D$, then the response message must contain a header (which in turn contains node information) for nodes $B$ and $D$. Again, we estimate the number of nodes represented in a response message by the number of nodes within $r$ hops of our node that replied to query $Q$, scaled appropriately.

Now, to calculate aggregate bandwidth consumed, let us define $o_j =$`QueryJoinRatio`,$o_u =$`QueryUpdateRatio`, and $m = MAX(p)$ for all $p \in P$. Bandwidth consumed is now calculated by the following formula, for a given radius $r$ and policy $P$:

$$\begin{aligned} BW_{li}(Q,P,r) = & \\ & \sum_{n=1}^{m} a(Q) \cdot \Big(N(Q,n) + C(Q,n)\Big) + \sum_{x \in P} x \cdot e(Q,r) \cdot N(Q,x) \\ & + \frac{1}{o_j}\left(f \cdot \sum_{n=1}^{r} \Big(N(Q,r) + C(Q,r)\Big) + f \cdot \sum_{n=1}^{r} N(Q,r)\right) \\ & + \frac{1}{o_u}\left(g \cdot \sum_{n=1}^{r} \Big(N(Q,r) + C(Q,r)\Big)\right) \end{aligned} \tag{7}$$

The first summation is the cost of sending the query to all nodes within a radius of $m$. The second summation accounts for the cost of transferring response messages back to the query source. Only nodes at policy levels execute the query, hence the summation is over policy levels only. At a policy level $x$, every node returns one response message of average size $e(Q,r)$, and this response message must be forwarded through $x$ hops.

The third term accounts for the cost of joining and leaving the network. First, Join messages of size $f$ are broadcast to all nodes within $r$ hops. Next, when a node joins the network and sends out its Join message, all nodes that receive this message will in turn send their Join messages directly back to the joining node. The cost of joins and leaves are then amortized over the cost of a query by dividing the join/leave cost by $o_j$, the value of `QueryJoinRatio`.

The fourth term accounts for the cost of transferring update messages. The cost of an update is amortized over the cost of a query by dividing the update cost by $o_u$, the value of `QueryUpdateRatio`.

## E Aggregate Processing Power Calculations

**Iterative Deepening** The formula for aggregate processing power is very similar to Equation 5, the formula for calculating aggregate bandwidth in iterative deepening, since for every message transferred, there is usually a corresponding cost for transferring and processing the message:

$PP_{id}(Q,P) =$

$$\sum_{n=1}^{L} I_2(Q,n) \cdot \Big(t(Q) \cdot \Big(N(Q,n) + C(Q,n)\Big) + n \cdot \Big(u \cdot R(Q,n) + v \cdot M(Q,n)\Big) + \Big(w \cdot R(Q,n) + x \cdot N(Q,n)\Big)\Big)$$
$$+ \sum_{x \in P} \Big(I_1(Q,x) \cdot I_2(Q,x) \cdot I_3(x) \cdot \sum_{n=1}^{x} s \cdot \Big(N(Q,n) + C(Q,n)\Big)\Big) \tag{8}$$

The first summation gives us the cost of broadcasting the query and receiving results, corresponding directly to the first summation in Equation 5. The first term of this summation gives us the cost of transferring the query message to level $n$. The second term gives us the cost of transferring the response messages back to the query source. For each Response message (there are $M(Q,n)$ Response messages at depth $n$), there is the base cost of transferring the message. In addition, for each result record (of which there are $R(Q,n)$), there is the additional cost of transferring those bytes. Finally, the third term in the summation gives us the cost of processing the query and generating the results.

The second summation gives us the cost of broadcasting Resend messages.

**Directed BFS** Again, the formula for calculating processing power for Directed BFS is almost identical to the formula for calculating bandwidth for Directed BFS deepening given in Equation 1:

$$PP_{dbfs}(Q,h) = \sum_{n=1}^{D} \Big(t(Q) \cdot \Big(N(Q,n,y) + C(Q,n,y)\Big) + n \cdot \Big(u \cdot R(Q,n,y) + v \cdot M(Q,n,y)\Big) + \Big(w \cdot R(Q,n,y) + x \cdot M(Q,n,y)\Big)\Big) \tag{9}$$

where $y = Y(Q,h)$.

The first term in the summation gives us the cost of sending the query to level $n$, and the second term gives us the cost of transferring the responses from level $n$ back to the query source. The third term gives us the cost of generating the responses at level $n$.

**Local Indices** Several new actions types must be considered for Local Indices.

First, when a Join message is processed, the node processing the Join will add the metadata contained in the message to its own index. On average, each Join message will contain metadata over a collection of about 340 files (Table 1). This cost is estimated by the cost of processing a Login message containing 340 filenames in the OpenNap system, given in [18], since the two actions involve the same types of operations over the same kinds of data structures. The formula given in [18] is in cycles; we normalize this cost to our generic units by dividing by the rough conversion factor of 7300 cycles per unit. Likewise, the cost of processing an Update message is estimated by the cost of processing a Download message in the OpenNap system, given in [18], because the two actions involve the same types of operations over the same kinds of data structures.

When a node leaves the network or dies, other nodes will detect the timeout and remove that node's metadata from their index. We estimate that processing a timeout is equal in cost to processing a Join message, since the same data structures that are written during a join, must also be written during a leave.

The cost of processing a query is similar to processing a query under iterative deepening or directed BFS. There is a fixed overhead for initiating the query, and an additional cost for every result discovered. Likewise, the cost of transferring a Response message also has a fixed overhead for sending the message, and an additional cost for every result record in the message. Again, the challenge is estimating how many results an average Response message will contain. Here, we use the same method described in Appendix D to estimate this value.

The procedure used to calculate aggregate processing power for a query is similar to the procedure used to calculate aggregate bandwidth. First, the cost of a query is calculated, then the cost of a join and leave (i.e., timeout). The cost of the join and leave is then amortized over the cost of the query, using the parameter `QueryJoinRatio`. Again, let us define $m$ to be the largest level specified in policy $P$. The formula for aggregate processing power for a query $Q$ is:

$$\begin{aligned}
&PP_{li}(Q,P,r) = \\
&\quad \sum_{n=1}^{m} t(Q) \cdot \Big(N(Q,n) + C(Q,n)\Big) + \sum_{x \in P} \Big(x \cdot y(Q,r) \cdot N(Q,x) + z(Q,r) \cdot N(Q,x)\Big) \\
&\quad + \frac{1}{o_j}\left((h+k) \cdot \sum_{n=1}^{r} N(Q,r) + j \cdot \sum_{n=1}^{r} \Big(N(Q,r) + C(Q,r)\Big) + j \cdot \sum_{n=1}^{r} N(Q,r)\right) \\
&\quad + \frac{1}{o_u}\left(\Delta \cdot \sum_{n=1}^{r} N(Q,r) + l \cdot \sum_{n=1}^{r} \Big(N(Q,r) + C(Q,r)\Big)\right) \qquad (10)
\end{aligned}$$

The first two summations are exactly analogous to the first two summations in the bandwidth calculation of Equation 7. The first summation gives us the cost of transferring the query to all nodes within a radius of $m$, and the second summation gives us the cost of generating and transferring responses from the nodes at the policy levels.

The third term in the formula accounts for the cost of joining and leaving the network. First, when a node joins, all nodes within a radius of $r$ will process the Join, which costs $x$ units. Similarly, when the node leaves, all nodes within a radius of $r$ will process the timeout, which costs $z$ units. These costs account for the term $(h+k) \cdot \sum_{n=1}^{r} N(Q,r)$.

Next, these Join messages just mentioned are broadcast with a TTL of $r$, meaning they will be sent to every node within a radius of $r$, as well as every cycle-completing edge within a radius of $r$. Total cost of transferring these messages is therefore $j \cdot \sum_{n=1}^{r} \Big(N(Q,r) + C(Q,r)\Big)$.

Finally, when a node receives a Join message, it will generate a Join message over its own collection, and send this message directly to the joining node. The cost of transferring these messages is $j \cdot \sum_{n=1}^{r} N(Q,r)$. Note that we do not include the cost of processing these Join messages at the joining node. This is because we do not consider a node as part of the network until after it has processed all of these join messages. Hence, the cost of processing these joins is incurred on a node that is not yet part of the network, which we do not want to count.

The total cost of processing and transferring Join messages and timeouts is finally divided by $o_j$, the *QueryJoin-Ratio*, in order to amortize the cost of join/leave over the cost of a query.

The fourth term in the formula accounts for the cost of transferring Update messages and then processing these messages. This cost is then divided by $o_u$ to amortize it over the cost of a query.