Enhancing P2P File-Sharing with an Internet-Scale Query Processor

Boon Thau Loo* Joseph M. Hellerstein*† Ryan Huebsch* Scott Shenker*‡ Ion Stoica*

*UC Berkeley, †Intel Research Berkeley and ‡International Computer Science Institute
{boonloo,jmh,huebsch,shenker,istoica}@cs.berkeley.edu

Abstract

In this paper, we address the problem of designing a scalable, accurate query processor for peer-to-peer filesharing and similar distributed keyword search systems. Using a globally-distributed monitoring infrastructure, we perform an extensive study of the Gnutella filesharing network, characterizing its topology, data and query workloads. We observe that Gnutella's query processing approach performs well for popular content, but quite poorly for rare items with few replicas. We then consider an alternate approach based on Distributed Hash Tables (DHTs). We describe our implementation of PIERSearch, a DHT-based system, and propose a hybrid system where Gnutella is used to locate popular items, and PIERSearch for handling rare items. We develop an analytical model of the two approaches, and use it in concert with our Gnutella traces to study the trade-off between query recall and system overhead of the hybrid system. We evaluate a variety of localized schemes for identifying items that are rare and worth handling via the DHT. Lastly, we show in a live deployment on fifty nodes on two continents that it nicely complements Gnutella in its ability to handle rare items.

1 Introduction

Distributed query processing has been a topic of database research since the late 1970's. In recent years, the problem has been revisited in the setting of peer-to-peer (P2P) filesharing systems, which have focused on a point in the design space that is quite different from traditional database research. P2P filesharing applications demand extreme scalability and federation, involving orders of magnitude more machines than even the most ambitious goals of distributed database systems proposed in the literature. P2P filesharing networks knit together hundreds of thousands of unmanaged computers across the globe into a unified query system. The data in filesharing consists of simple files stored at the end-hosts; the names of the files are queried *in situ* without transmitting them to any centralized repository. A P2P filesharing network typically runs thousands of concurrent keyword queries over the names of the files in the network. Separately, it supports point-to-point downloads of actual file content between peers.

Popular P2P filesharing systems like Gnutella [7] and Kazaa [13] are based on very simple designs, and there is controversy over their effectiveness. These systems connect peer machines into an ad-hoc, *unstructured* network. Query processing proceeds in a very simple fashion known as "flooding": a node transmits a keyword query to its P2P network neighbors, who forward the query on recursively for a finite number of hops known as the "time-to-live" (TTL) of the query. Any node with a matching filename reports back to the query source, which displays a list of matching file locations and properties. Given the scale of these networks, flooding-based schemes are not exhaustive; a given query will visit only a small fraction of nodes in the network. As a result, these networks provide no guarantees on query recall, and often fail to return matches that actually exist in the network.

Recently, researchers have been focusing significant attention on alternative *structured* P2P networks that can support content-based routing. These networks are able to ensure that all messages labeled with a given "key" are routed to a particular machine. This allows the network to coordinate global agreement on the location of particular items[1] to be queried: keyed items are routed by the network to a particular node, and key-based lookups are routed to that same node. This functionality is provided without any need for centralized state, and works even as machines join and leave the P2P network. Content-based routing is akin to the "put()/get()" interface of hash tables, and these networks have thus been dubbed Distributed Hash Tables (DHTs). DHTs have matured rapidly in recent years via both theoretical results and system prototypes [2]. Unlike the popular unstructured P2P networks, DHTs can in principle provide

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

[1] In this paper, we will use the terms "files" and "items" interchangeably

full recall from a network of connected peers.

To date, there has been little agreement on the best design for query processing in P2P filesharing systems. In this paper we attempt to address the question on a number of fronts. First, we highlight the strengths and weaknesses of unstructured P2P networks via an extensive empirical analysis of the Gnutella network. We performed live, distributed monitoring of the Gnutella network via multiple machines spread across the two continents in the PlanetLab testbed [19]. We gathered extensive traces of the network's graph structure, its query workload, and its file contents. One of our key observations is that replication of files in the network follows a long-tailed distribution with a moderate number of "popular" files containing many replicas in the network, and a long tail of many "rare" files containing few replicas. Given that observation, we observe that the flooding-based approach in unstructured networks is an efficient, simple solution for finding copies of popular files, but has poor latency and result quality for queries that focus on rare items.

Second, we describe *PIERSearch*, our implementation of DHT-based keyword querying. PIERSearch is an application built on top of PIER [12], a DHT-based Internet-scale relational query engine we have built in our group. The DHT-based approach does provide better answers in terms of query recall, but can require more network overhead to "publish" files by keyword into the DHT, and to perform distributed joins of keyword lists at query processing time.

Based on our analysis of the workload and solutions, we propose a simple hybrid approach for high-quality P2P search, in which PIERSearch is used to build a partial index [23] over only the rare items in the Gnutella network. Queries are handled in a hybrid manner: popular items are found via the native Gnutella protocol, and rare items are found via PIERSearch.

We provide an analytical model to study the potential benefits of a universal deployment of PIERSearch bundled with Gnutella. Using this model together with our Gnutella traces, we study the trade-off between query recall and system overhead of the hybrid system. In addition, we propose and compare a variety of techniques for one of the key challenges in the hybrid solution: correctly identifying the "rare" files that should be indexed in the DHT.

Finally, we implemented this solution by modifying the open-source LimeWire Gnutella software, combining it with PIERSearch. We ran our implementation on fifty PlanetLab nodes across two continents, participating live in the Gnutella network; the addition of PIERSearch alongside Gnutella – even on a limited subset of Gnutella nodes – demonstrates notable benefits in both latency and recall for queries that focus on rare items.

2 Background: DHTs and PlanetLab

There have been many proposals for DHT designs in the last few years; we briefly describe their salient features here. An overview of DHT research appears in [2]. As its name implies, a DHT provides a hash table abstraction over multiple distributed compute nodes. Each node in a DHT can store data items, and each item is indexed via a lookup key. At the heart of the DHT is an overlay routing scheme that delivers requests for a given key to the node currently responsible for the key. This is done without global knowledge or permanent assignment of the mappings of keys to machines. Routing proceeds in a multi-hop fashion; each node keeps track of a small set of neighbors, and routes messages to the neighbor that is in some sense "nearest" to the correct destination. Most DHTs guarantee that routing completes in $O(\log N)$ P2P message hops for a network of N nodes. The DHT automatically adjusts the mapping of keys and neighbor tables when the set of nodes changes.

The DHT forms the basis for communication in PIER. With the exception of query answers, all messages are sent via the DHT routing layer. PIER also stores all temporary tuples generated during query processing in the DHT. The DHT provides PIER with a scalable, robust messaging substrate even when the set of nodes is dynamic.

Realistic assessments of peer-to-peer systems can be difficult to achieve without machines spread around the world. In our work, we made heavy use of the PlanetLab testbed [19], both to analyze Gnutella from multiple vantage points, and to test our implementation of PIERSearch in a truly distributed setting. PlanetLab is an open, globally distributed platform for developing, deploying and accessing planetary-scale network services. PlanetLab today consists of over 350 machines located at 148 sites in five continents. In our experiments, we utilized machines from different parts of North America (including Canada) and Europe.

PlanetLab enabled us to achieve serious experimental results: as we report later, we injected 63,000 queries into Gnutella, we crawled 100,000 Gnutella nodes in only 45 minutes, and we deployed the PIERSearch engine on fifty sites distributed on two continents. In fact, one of our challenges with PlanetLab was to use its power carefully: early on, our experiments raised warning flags among system administrators because they resembled malicious network behavior.

3 Overview of PIERSearch

PIERSearch is a DHT-based search engine implemented using PIER. Figure 1 shows the design of PIERSearch on a single node. PIERSearch supports a class of queries based on *keyword search*, which enables us to query for all items containing a given set of keywords (or terms). Items with filenames that contain all search terms will satisfy the query. PIERSearch consists of two main components: the *Publisher* and *Search Engine*, described in detail below.

3.1 Publisher

To support these queries, PIERSearch maintains an inverted file, which is an index structure that enables fast retrieval of all items that contain a search term. For each term, the index maintains an *inverted list* or *posting list* of all file identifiers (fileIDs) of items that contain the indexed term. In order to quickly find the inverted list for a search term, all possible query terms are organized in an index structure such as a B+ tree or hash index. In the case of PIERSearch, the indexing

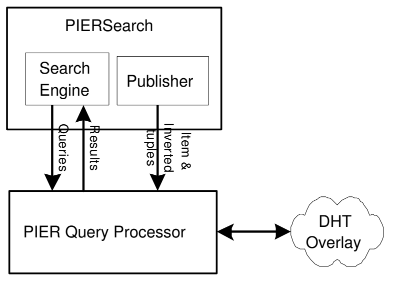

Figure 1: *PIERSearch on a single node.*

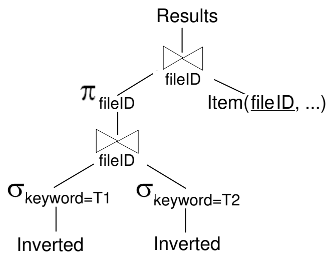

Figure 2: *Relational query plan for a two-term keyword query* "`T1 AND T2`".

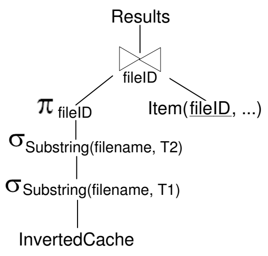

Figure 3: *Query Plan of InvertedCache Option for a two-term query* "`T1 AND T2`"*. The corresponding filename is cached on every Inverted tuple.*

mechanism is provided by the DHT itself. In summary, for each item, the *Publisher* generates tuples conforming to the following schema (primary keys are underlined):

- **`Item(fileID, filename, filesize, ipAddress, port)`**. The *Item* table contains a tuple for each item that is being shared. It stores the filename, filesize, the location (IP Address and port) of the host sharing the file and any other additional fields describing the item. The *fileID* is a unique file identifier of the item, generated by applying a hash on the other fields, and is used as the publishing (index) key for the DHT.
- **`Inverted(keyword, fileID)`**. Each item has a set of keywords that describes itself. Typically in file-sharing, this comprises the terms in the filename. Stop-words such as "MP3" and "the" are usually not considered. For each keyword, we generate an *Inverted* tuple, which contains the keyword and the fileID of the item. The fields *keyword* and *fileID* form the primary key, but only the *keyword* field is used as the publishing (index) key for the DHT. This ensures that *Inverted* tuples with the same keyword are hosted by the same node.

3.2 Search Engine

For a given search query, the *Search Engine* forms a query which its local PIER engine executes on its behalf. Figure 2 shows an example query plan for a two-term query T_1, T_2. Conceptually, the query plan retrieves two sets of *Inverted* tuples, one for $keyword = T_1$ and another for $keyword = T_2$, and executes a join of the two sets of tuples by fileIDs. *Item* tuples with the resulting fileIDs form the answer set. This query plan can be extended for queries with more than two search terms simply by adding an extra self-join with the *Inverted* relation for each additional keyword.

When this query is executed, PIER routes the query plan via the DHT to all sites that host a keyword in the query, and executes a distributed join of the posting list entries of matching *Inverted* tuples. Using our example query plan, the node that hosts the first keyword (T_1) in the query plan will send (rehash) the matching *Inverted* tuples to the node that hosts for the next keyword (T_2). The receiving node will perform a *symmetric hash join (SHJ)* between the incoming tuples and its local matching tuples, and send the results to the next node (if there are more keywords). On the node hosting the last keyword in the query plan, the matching fileIDs are streamed back to the query node, which fetches the *Item* tuples from the DHT based on the incoming fileIDs.

In addition to this distributed join algorithm, PIERSearch also provides an alternative approach that we call the *InvertedCache* option. In the InvertedCache option, the schema is modified, replacing the *Inverted* table with a new table **`InvertedCache(keyword, fileID, fulltext)`**. This table stores the *full text* (i.e. the filename) redundantly with each (keyword, fileID) pair. Figure 3 shows the query plan. The InvertedCache option essentially "caches" the file text with each inverted file entry. Consequently, the matching fileIDs for the search query can be resolved without distributed joins: the query can be sent to a single node hosting *any one* of the search terms (T_1 in the example), and the remaining search terms are filtered locally using substring selection operators. Hence, the communication cost of computing the matching fileIDs is greatly reduced since no *Inverted* tuples need to be shipped. However, this technique incurs extra publishing overheads, which are prohibitive for typical full text document search, but tolerable for indexing short filenames. We quantify this overhead experimentally for typical filenames in Section 7.

4 Gnutella Measurements

In order to motivate the need for PIERSearch to enhance existing unstructured networks, we first analyze Gnutella, a file-sharing network based on an unstructured network design. Unlike a DHT-based search scheme, data need not be published in an unstructured network. Each node that joins the network shares its local files, and queries are flooded in the network. Whenever a node receives a query, it checks its local files and returns a query response containing information on any matches in its local files.

The current Gnutella network uses several optimizations to improve the performance over the original "flat" flooding design described above. Some of the most notable optimizations include the use of *ultrapeers* [10] and *dynamic querying* techniques [8]. We describe these informally here;

a more thorough description of the Gnutella protocol today is available in the Gnutella 0.6 protocol specification [1].

When a node starts up a Gnutella client it joins as a "leaf" node of the Gnutella network: it can issue queries to the network, but will not answer or forward query requests for any other nodes. Upon joining the network, the leaf node selects a small number of nodes that have been elected "ultrapeers", and then it publishes its file list to those ultrapeers. Ultrapeers perform query processing on the behalf of their leaf nodes. A query from a leaf node is sent to an ultrapeer, which floods the query to its ultrapeer neighbors, recursing up to the query TTL. Nodes regularly determine whether they are eligible to become ultrapeers ("ultrapeer capable") by looking at their uptime, operating system and bandwidth. Once nodes decide they are ultrapeer capable, they express their capabilities to other connecting hosts via the Gnutella connection headers.

Dynamic querying is a search technique whereby queries that return few results are re-flooded deeper into the network. While this scheme is used in Gnutella today, we will have more to say about the efficacy of "deep flooding" for small results in Section 4.3.

To analyze the Gnutella network, we modified the popular LimeWire client software [15]. Our modified client can participate in the Gnutella network either as an ultrapeer or leaf node, and can log all incoming and outgoing Gnutella messages. In addition, our client has the ability to inject queries into the network and gather the incoming results. The client software was deployed on multiple PlanetLab nodes, and participated directly in the Gnutella network.

4.1 Gnutella Topology

To estimate the size of the Gnutella network and confirm Gnutella's topology, we began our study by performing a "crawl" of the Gnutella network graph; to do this we recursively invoke a Gnutella API call that returns a node's current list of neighbors. A P2P network like Gnutella is subject to noticeable "churn" of nodes joining and leaving, so an elongated crawling process does not provide an accurate "snapshot". To increase the accuracy of our estimation, we performed a distributed, parallel crawl, starting from 30 ultrapeers running on PlanetLab for about 45 minutes on 11 Oct 2003. Based on these measurements, the network size of Gnutella during the crawl was around 100,000 nodes, and there were roughly 20 million files in the system. Note that the size of the network is a lower bound since not all nodes respond to our crawler.

Our crawl also revealed that most ultrapeers today support either 30 or 75 leaf nodes. This is confirmed by the development history of the LimeWire software: newer LimeWire ultrapeers support 30 leaf nodes and maintain 32 ultrapeer neighbors, while the older ultrapeers support 75 leaf nodes and 6 ultrapeer neighbors[2].

[2]As a side note, in newer versions of the LimeWire client, leaf nodes publish Bloom filters of the keywords in their files to ultrapeers [9, 8]. There have also been proposals to cache these Bloom filters at neighboring nodes. Bloom filters reduce publishing and searching costs in Gnutella, but preclude substring and wildcard searching (which are similarly unsupported in DHT-based search schemes.).

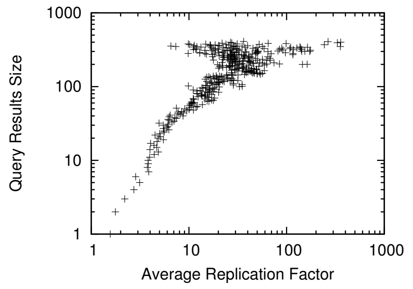

Figure 4: *Correlating Query Results Size vs. Average Replication Factor.*

4.2 Gnutella Search Quality

Next, we turn our attention to analyzing the search quality of Gnutella, both in terms of *recall* and *response time*. There are two possible definitions of recall that we will use.

- **Query Recall (QR)** is defined as the percentage of available results in the network returned. In this case, each replica of a file is counted as a distinct result. Results can be distinguished by filename, host, and filesize.
- **Query Distinct Recall (QDR)** is defined as the percentage of available *distinct* results in the network returned. In this case, there is no gain from having multiple replicas of a file within a result set. For simplicity in defining this metric, we assume that files are uniquely distinguished by their filename; files are grouped by filename in many Gnutella clients as well.

In order to measure recall accurately, we would need to know about all files in the network at the time of our experiments. Given the difficulty of taking an accurate snapshot of all files, we approximate the total number of query results available in the system by issuing the query simultaneously from all 30 PlanetLab ultrapeers, and taking the union of the results. We justify our *Union-of-30* approach to approximating the true contents of the network in two ways. First, we experimentally verified that as we increased the number of PlanetLab ultrapeers beyond 15, we found little increase in the total number of results (see Figure 6). This suggests that the number of results returned by all 30 ultrapeers a reasonable approximation of the total number of results available in the network. Second, because this approximation underestimates the number of total results in the network, the recall values that we compute for Gnutella's search strategy are at worst *overestimates* of the actual values.

We deployed our modified LimeWire ultrapeers on PlanetLab nodes on two continents to obtain real Gnutella query traces. We chose 700 distinct queries from these traces to replay at each of the PlanetLab ultrapeers. To factor out the effects of workload fluctuations, we replayed queries at three different times. In total, we injected $63,000$ queries into Gnutella ($700 \times 30 \times 3$). We make three observations based on the results returned by these queries.

First, as expected, there is a strong correlation between the number of results returned for a given query, and the number of replicas in the network for each item in the query

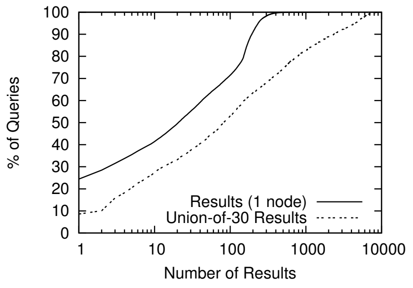

Figure 5: *Result size CDF of Queries; note the log scale on the x-axis.*

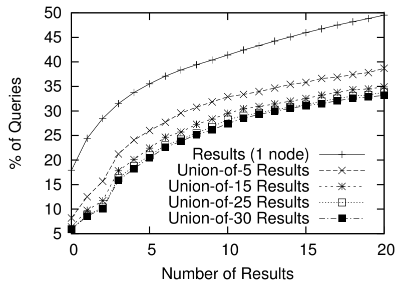

Figure 6: *Result size CDF for Queries* ≤ 20 *results.*

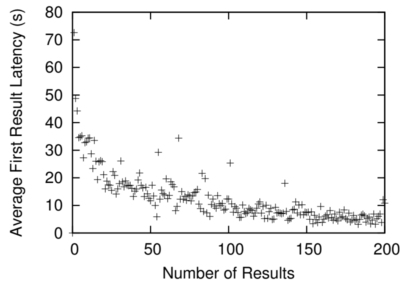

Figure 7: *Correlating Result Size vs. First Result Latency.*

result set. The *replication factor* of an item is defined as the total number of identical copies of the item in the network. Again, to approximate this number, we count the number of items with the same filename in the union of the query results obtained by the 30 ultrapeers for the same query. We then compute the average replication factor of a query by averaging the replication factors across all distinct filenames in the query result set. Figure 4 summarizes our results, where the Y-axis shows query results set size, and the X-axis shows the average replication factor averaged across all queries for each results set size. In general, queries with small result sets return mostly rare items, while queries with large result sets return both rare and popular items, with the bias towards popular items.

Second, our results demonstrate the effectiveness of Gnutella in finding highly replicated content. We present our analysis for the QR metric here; since the QDR results are similar, we omit them. Figure 5 plots the Cumulative Distribution Function (CDF) of the number of results returned by all queries (the *Results* curve), and the "Union-of-30" results, which provide a lower bound on the total number of matching items in the network. Note that there are queries returning as many as 1,500 results to a single client, which would seem more than sufficient for most file-sharing uses. In addition, Figure 7 shows that the queries with large result sets also have good response times. For queries that return more than 150 results, we obtain the first result in 6 seconds on average.

Third, our results show the *ineffectiveness* of Gnutella in locating rare items. Figure 7 shows that the average response time of queries that return few results is poor. For queries that return a single result, 73 seconds elapsed on average before receiving the first result.

An important point to note is that queries that return few items are quite prevalent. Figure 6 shows the results of the same experiment as Figure 5, limited to queries that return at most 20 results, for unions of 5, 15 and 25 ultrapeers. Note that 41% of the standard (single-node) queries receive 10 or fewer results, and 18% of standard queries receive *no* results. For a large fraction of queries that receive no results, matching results are in fact available in the network at the time of the query. By comparison, the Union-of-30 results are considerably better: only 27% of queries receive 10 or fewer results, and only 6% of queries receive no results. This means that there is an opportunity to reduce the percentage of queries that receive no results from 18% to at most 6%, or equivalently to reduce the number of queries that receive no results by at least 66%. We say "at least" because the Union-of-30 results are an underestimation of the total number of results available in the network.

When we switch to the QDR metric, while 14% of queries receive more than 100 distinct results, and 2% of queries receive more than 200 distinct results, as many as 48% of queries receive 10 or fewer distinct results. This percentage is reduced from 48% to 33% when we look at the Union-of-30 results. The improvements for empty query results remain the same for the QDR metric, since the emptyset has no duplicates. The QDR graphs are omitted for brevity.

4.3 Increase the Search Horizon?

An obvious technique to locate more rare items in Gnutella would be to increase the search horizon by using larger TTLs. While this would not help search latency, it could improve query recall. As the search horizon increases, the number of query messages sent will increase substantially, with decreasing payoffs. Figure 8 shows the the number of query messages sent on average to reach a number of ultrapeers in the network. This is based on analyzing the crawl topology obtained in Section 4.1. As the search horizon increases, even when we suppress duplicate messages received at each node, there is diminishing returns in reaching more ultrapeers as the number of messages increases. E.g., 48K messages are required to reach 9,000 ultrapeers, but to reach the next 9,000 ultrapeers, an extra 94K messages are required. Hence, even when the search horizon increases by a single hop, the number of nodes contacted does not increase at the same rate as the messaging overheads. The diminishing returns with increasing search horizon is due to nodes receiving duplicate messages from more than one neighbor node. The duplicate messages are a result of redundant paths in the network. I.e., a node that has received a query message may receive the same query message later from another neighbor as the search horizon increases. To address this problem, there has been recent proposals in the research literature [3] for flooding Gnutella nodes via a DHT overlay that would eliminate redundant paths.

Given that queries that return few results are fairly common, such aggressive flooding to locate rare items is unlikely to scale. In future work, we plan to quantify the im-

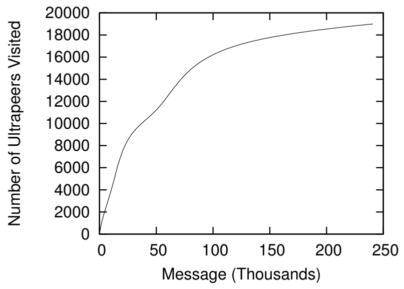

Figure 8: *Gnutella Flooding Overhead*

pact of increasing the search horizon on the overall system load.

4.4 Summary

Our Gnutella measurements reveal the following findings:

- Gnutella is highly effective for locating popular items. Not only are these items retrieved in large quantities, the queries also have good response times.
- Gnutella is less effective for locating rare items: 41% of all queries receive 10 or fewer results, and 18% of queries receive *no* results. Furthermore, the results have poor response times. For queries that return a single result, the first result arrives after 73 seconds on average. For queries that return 10 or fewer results, 50 seconds elapsed on average before receiving the first result.
- There is a significant opportunity to increase the query recall for locating rare items. For instance, the number of queries that return no results can be reduced from 18% to at least 6%.

5 Hybrid Search Infrastructure

In the view of the shortcomings of a flooding-based unstructured network, we explore the feasibility of using PIERSearch as an alternative for supporting file-sharing networks. PIERSearch utilizes DHTs; various research efforts have proposed DHT-based search engines as an alternative to unstructured networks like Gnutella, arguing that the use of DHTs can improve query performance.

While PIERSearch provides perfect recall in the absence of network failures, a full-fledged implementation where all nodes run PIERSearch has its own drawbacks. The content publishing phase can consume large amounts of bandwidth compared to queries that retrieve sufficient results via flooding in an unstructured network. Consider the query "Britney Spears" that requests all songs from this popular artist. "Britney" and "Spears" are popular keywords with large posting lists. The publishing costs of building the inverted indexes for these two keywords are high. A "Britney Spears" query also requires shipping large posting lists to perform the distributed join. Recent back-of-the-envelope calculations [14] suggest that shipping large posting lists over DHTs is bandwidth-expensive. While compression techniques and Bloom filters would reduce the bandwidth requirements of publishing, a flooding scheme that does not incur any publishing overheads is both simpler and more efficient for such queries.

On the other hand, queries over rare items are less bandwidth-intensive to compute, since fewer posting list entries are involved. To validate the latter claim, we replayed $70,000$ Gnutella queries over a sample of $700,000$ files[3] using the SHJ algorithm (optimized to compute smaller posting lists first) described in Section 3. We observed that on average, queries that return 10 or fewer results require shipping 7 times fewer posting list entries compared to the average across all queries. This motivates a *hybrid search infrastructure*, where the PIERSearch builds a *partial index* for locating rare items, and flooding techniques are used for searching highly replicated items.

The hybrid search infrastructure utilizes *selective publishing* techniques that identify and publish only rare items into the DHT. This hybrid infrastructure can easily be implemented if all the ultrapeers are organized into the DHT overlay and run the PIERSearch client. In this *full deployment* scenario, each ultrapeer is responsible for identifying and publishing rare files from its leaf nodes. Search is first performed via conventional flooding techniques of the overlay neighbors. If not enough results are returned within a predefined time, the query is reissued using PIERSearch.

A key challenge for the hybrid system is in identifying rare items for publishing into the DHT. The schemes should be as localized as possible, minimizing communication between nodes. Our proposed schemes are listed below. We will revisit the comparison of these schemes in Section 6.3.

- **Query Results Size (QRS).** Based on our initial observation in Section 4.2, rare files are those that are seen in small result sets. A parameter *Results Size Threshold* is used to determine what query results needs to be cached. Query results from queries with a results set size smaller than the threshold are published. In essence, the DHT is used to cache elements of small result sets. This scheme is simple, but suffers from the fact that many rare items may not have been previously queried and found, and hence will not be published via a caching scheme.
- **Term Frequency (TF).** Each hybrid node gathers term statistics of filenames over a period of time by monitoring filenames from the search results traffic. We use a parameter *Term Frequency Threshold* to determine whether an item is rare. Items with *at least one* term below the threshold are considered rare items. Based on our measurements of Gnutella ultrapeers, each ultrapeer sees an average of $30,000$ query results per hour. Hence, by observing for days, an ultrapeer can easily identify millions of filenames. While this is not exhaustive, it is feasible if term frequencies remain fairly constant over a short period of time.
- **Term Pair Frequency (TPF).** Individual terms may be subjected to skews in popularity. For example, a rare item may have a popular keyword. An alternative scheme considers term pair frequencies instead. Here,

[3]These queries and files were collected from 30 ultrapeers as described in Section 4.2.

Parameter	Value
N	Number of nodes in the system.
$N_{horizon}$	Number of distinct nodes contacted when a query is flooded over the Gnutella network. The horizon includes the query node itself.
R_i	Number of replicas for item i.
T_i	Lifetime of item i in the network.
Q_i	Frequency that item i is queried per time unit.

Table 1: System Parameters for Hybrid System

items with *at least one* term pair below the *Term Pair Frequency Threshold* is considered rare. Since generating all possible term pairs is memory consuming, we will only consider ordered term pairs that are adjacent to each other in the filename.

- **Sampling (SAM).** This scheme samples neighboring nodes and compute a lower bound estimate on the number of replicas for each item. A parameter *Sample Threshold* is then used by each node to select only its local items whose lower bound estimate based on the sample is below the threshold for publishing. Ideally, the sampling is done on-line every time a hybrid node joins the system. Since this may incur high overhead, a less accurate but less bandwidth consuming alternative is to gather replica counts of filenames over a period of time.

6 Modeling and Evaluating Hybrid Search

In this section, we describe a simple analytical model to quantify the benefits of the hybrid system, and to better understand the trade-off between the query recall and the system overhead. In addition, we use the model to quantify the quality of the query results obtained by using the publishing schemes described in Section 5. For this purpose we use trace driven simulations (see Section 6.3).

6.1 Model

We consider a hybrid system consisting of a Gnutella network and a PIERSearch system that share a common set of N nodes. Here we assume a Gnutella network, although the model applies to any flooding-based unstructured network. Let I be the set of items (including duplicate items) shared by all the nodes, and $i \in I$ be an arbitrary item in the network. Tables 1 and 2 summarize the notations used to describe our system. We make the following simplifying assumptions:

- All nodes are involved in query processing.
- No new items and nodes are added or removed from the network, and files are not replicated after being queried.
- Replicas are randomly distributed in the network, and no identical replicas reside on the same node. The links between nodes are random. Hence, querying an item in Gnutella is equivalent to querying a *random* sub-set of nodes in the network.
- All costs of the system are dominated by the communication overhead, which is measured in terms of transmitted messages.
- The search horizon is fixed for all queries, regardless of the number of results returned[4]
- Flooding is implemented using an efficient broadcast mechanism. Thus, it takes $n - 1$ messages to flood n nodes. Note that this overhead is within a constant factor (i.e., the average node degree) of the overhead incurred by Gnutella.

In the hybrid system, a query for item i is first issued to Gnutella. If Gnutella does not return any results, the query is re-issued to the DHT. Thus, the probability $PF_{i,hybrid}$ that an item i is found in the hybrid system is simply:

$$PF_{i,hybrid} = PF_{i,Gnutella} + PNF_{i,Gnutella} \times PF_{i,DHT} \tag{1}$$

If a query for item i in Gnutella visits $N_{horizon}$ nodes, the probability that item i is not found (and thus the query has to be re-issued in the DHT) is

$$PF_{i,Gnutella} = 1 - \prod_{j=0}^{j=N_{horizon}-1} \left(1 - \frac{R_i}{N-j}\right), \tag{2}$$

where R_i represents the number of replicas of item i in the system. Note that $(1 - R_i/N)$ represents the probability that no replica of item i is found at the first node (visited by the query), $(1 - R_i/(N-1))$ represents the probability that no replica of item i is found at the second node, and so on.

Next, we compute the overheads incurred by the query and the publishing operations. Let Q_i be the query frequency of item i, i.e., the number of queries for item i per time unit. The cost per time unit of querying item i in the hybrid system is then

$$CS_{i,hybrid} = Q_i \times ((N_{horizon} - 1) + PNF_{i,Gnutella} \times CS_{i,DHT}) \tag{3}$$

where $N_{horizon} - 1$ represents the cost of querying the item using Gnutella, and $CS_{i,DHT}$ represents the cost of querying the item in the DHT. In a typical DHT system, $CS_{i,DHT}$ is $\log N$ messages [20, 22, 24, 28] (with the *InvertedCache* option).

Further, let T_i be the life-time of node i in the system[5], and let $CP_{i,DHT}$ be the cost of publishing item i into the DHT. Then the total cost per time unit of maintaining and querying item i is

$$CO_{i,hybrid} = CS_{i,hybrid} + (PF_{i,DHT} \times \frac{CP_{i,DHT}}{T_i}) \tag{4}$$

[4]While several optimizations such as dynamic flooding have been proposed to improve the query performance, we do not consider such optimizations in our model.

[5]Typically, the life-time of an item is equal to the interval of time the node is in the system.

Variable	Value
$PF_{i,Gnutella}$	Probability that item i is found in Gnutella network.
$PNF_{i,Gnutella}$	Probability that item i is *not* found in Gnutella network. This is set to $(1 - PF_{i,Gnutella})$.
$PF_{i,DHT}$	Probability that item i is published into the DHT.
$PF_{i,hybrid}$	Probability that item i is found in the hybrid system.
$CS_{i,hybrid}$	Cost per time unit of searching for item i in the hybrid system.
$CS_{i,DHT}$	Cost of searching for item i in the DHT.
$CP_{i,DHT}$	Cost of publishing item i and its posting list entries into the DHT.
$CO_{i,hybrid}$	Overall cost per time unit of supporting item i as a result of the hybrid system.
$CP_{all,hybrid}$	Total publishing cost of the hybrid system.

Table 2: Search Capabilities and Cost Variables of the Hybrid System

Finally, the total cost of publishing rare items into the DHT is

$$CP_{all,hybrid} = \sum_{i \in I} (PF_{i,DHT} \times CP_{i,DHT}) \qquad (5)$$

The goal of the system is to maximize the probability $PF_{i,hybrid}$ that each item i is found, while minimizing the overall publishing overhead $CP_{all,hybrid}$. Maximizing the probability that an item is found in the hybrid system directly translates into improvements of the query recall.

6.2 Query Recall with Complete Knowledge

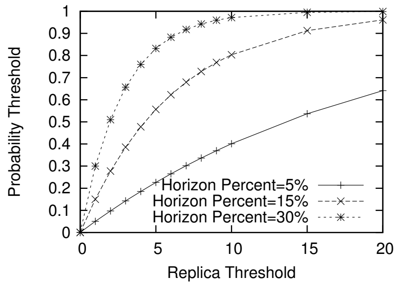

Figure 9: $PF_{threshold}$ *vs Replica Threshold.*

In this section, we quantify the search quality and the publishing overhead in the hybrid system as a function of the *replica threshold.* We assume that every node in the system has complete knowledge of the number of replicas for each item. Each node selects items whose number of replicas are smaller or equal to the replica threshold for publishing into the DHT. We derive the replica distribution of the items in the system from one of the experiments described in Section 4.2. In particular, we consider the results returned by 350 distinct queries issued from 30 ultrapeers. These results consists of $315,546$ files stored at $75,129$ nodes.

Figure 9 plots the probability threshold $PF_{threshold}$ versus the *replica threshold*, where $PF_{threshold}$ determines the lower bound on the probability $PF_{i,hybrid}$ that any item i is found in the hybrid system. The figure clearly shows a diminishing increase of $PF_{threshold}$ as more and more popular items are published into the DHT.

Figure 10 shows the publishing overhead (measured as the percentage of items being published) versus the replica threshold. Note that the percentage of items published is proportional to the total publishing cost $CP_{all,hybrid}$. When replica threshold is set to one, 23% of items are published. As the replica threshold increases, the increase of the publishing overhead diminishes.

Figure 11 plots the average *query recall* (QR) of queries in our trace versus the replica threshold for different values of the percentage of nodes in the search horizon. The search horizon represents the total number of nodes in the system that are involved in a Gnutella query. As defined in Section 4.2, QR is computed by taking a ratio of the number of results returned by the hybrid network to the total number of results in the entire network. As expected, when no items are published into the DHT (i.e., the replica threshold is zero), the average query recall is equal to the percentage of nodes in the search horizon. As the replica threshold increases, the query recall increases sharply. For a replica threshold of one, the average query recall increases to 47%, 52%, and 61%, respectively. When the replica threshold is two, the average query recall exceeds 64% in all cases.

Similarly, Figure 12 plots the *query distinct recall* (QDR) versus the replica threshold. QDR of a query is defined as the percentage of all *distinct* results in the network returned for the query. This definition naturally leads to higher recall values since replicas of the same item within the results set are ignored. Conversely, publishing multiple copies of the same item does not benefit this metric. Note that average QDR is exactly $PF_{i,hybrid}$ as computed by Equation (1).

In summary, both Figures 11 and 12 show that there is a diminishing return in the increase of the query recall as the replica threshold becomes larger. Thus, there is little benefit in publishing items that are already popular. In addition, these plots suggest that the hybrid system works well even when only the very rare items are published. For example, publishing only items with one or two replicas, raises the average QR and average QDR to 68% and 93%, respectively, for a horizon percentage of 15%.

6.3 Rare Items Schemes

In the previous section we have assumed that *all* items with a number of replicas smaller or equal to the *replica threshold* are published into the DHT. This represents the best one can do in terms of query recall subject to publishing cost constraints. For this reason, we refer to this scheme as the

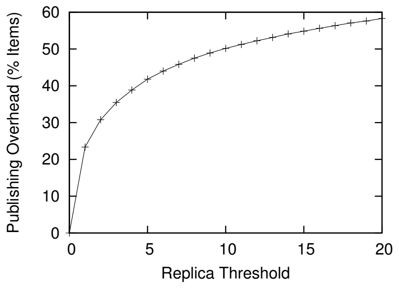

Figure 10: *Publishing Overhead (% of items published) vs Replica Threshold.*

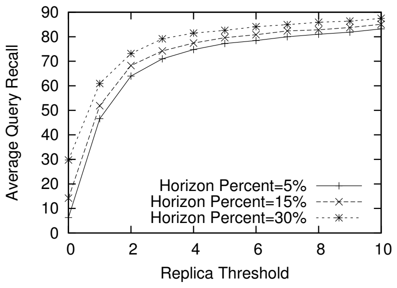

Figure 11: *Average Query Recall vs Replica Threshold*

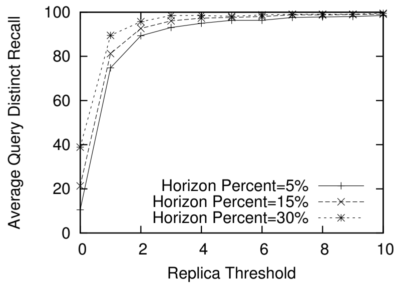

Figure 12: *Average Query Distinct Recall vs Replica Threshold*

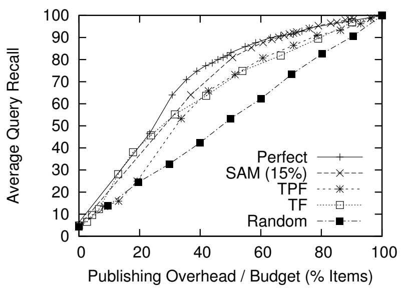

Figure 13: *Compare Schemes based on Average Query Recall for different Publishing Overhead (% of items published).*

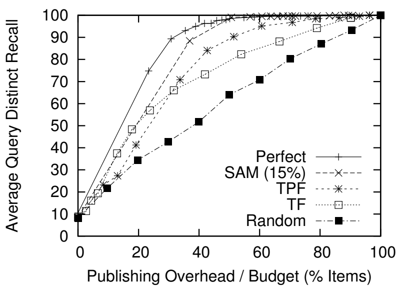

Figure 14: *Compare Schemes based on Average Query Distinct Recall for different Publishing Overhead (% of items published).*

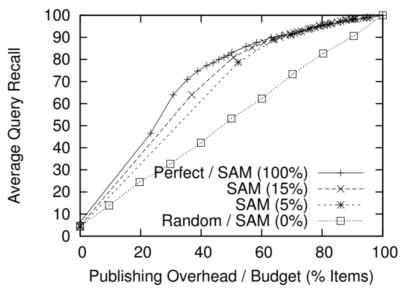

Figure 15: *Compare Sampling Sizes for SAM scheme based on Average Query Recall for different Publishing Overhead (% of items published).*

Perfect publishing scheme. Unfortunately, this scheme is not practical, as it requires knowledge of all replicas stored in the system. In this section, we evaluate the publishing schemes described in Section 5: Term Frequency (TF), Term Pair Frequency (TPF) and Sampling (SAM)[6]. In addition the *Perfect* scheme, we consider a *Random* publishing scheme, where each item is randomly published into the DHT irrespective of its number of replicas. We use the *Perfect* publishing scheme as an upper bound, and the *Random* publishing scheme as a lower bound for evaluating our publishing schemes.

Figure 13 shows the average QR achieved for each scheme, given the publishing overhead (percentage of items published) for a search horizon of 5%. The publishing overhead is the "publishing budget" available to the hybrid system. For a given budget, a good scheme that avoids false positives and false negatives will identify a set of least replicated items to be published. To vary the publishing budget in this experiment, we adjusted the *Replica Threshold*, *Term Frequency Threshold*, *Term Pair Frequency Threshold* and *Sample Threshold* for the respective schemes. We assume that SAM samples 15% random nodes, and we denoted it by SAM (15%).

As expected, the average recalls of all schemes lie between *Perfect* (best) and *Random* (worst) recalls. SAM (15%) has the highest average query recall among all schemes, achieving nearly the same average query recall as the *Perfect* recall when the publishing overhead exceeds 50%. TP and TPF perform similarly for large publishing overheads ($> 50\%$). For low publishing overheads (50%), TP performs better than TPF. Both TP and TFP provide a noticeable improvement over *Random*. For example, when the publishing overhead is 50%, the average query recall of both schemes is 70%, which represents a 40% improvement over *Random*.

Figure 14 shows the same experiment as above, but measuring average QDR instead the average query recall. The results are similar. For large publishing overheads (i.e., $> 50\%$), SAM (15%) preforms as well as *Perfect*. Similarly, TPF performs worse than TF for a low publishing overheads ($< 30\%$), and better than TF for publishing overheads larger than 30%.

We summarize our observations below:

- Term Frequency schemes work relatively well, and are attractive due to modest storage requirements to keep term statistics. In our traces, there were $38,900$ distinct terms and $193,104$ distinct adjacent term pairs. A typical PC can easily accommodate data sets that are one or even two order of magnitude higher than what we observed. To further reduce storage requirements one could use Bloom filters [17] to encode these sets. TPF is more effective than TF, except for low publishing overheads, where a large number of term pairs with low counts leads to poorer accuracy when the *Term Pair Frequency Threshold* is small.

[6]Due to the lack of sufficient queries to train the Query Results Size (QRS) scheme, we omitted evaluating the scheme.

- SAM has the best query recall, but this comes at the expense of non-trivial sampling overhead. For example, in a $100,000$ node network, SAM (15%) needs to sample about $15,000$ nodes. There are three solutions to alleviate this problem. First, we can reduce the number of nodes that are sampled. Figure 15 suggests that this is a good approach. SAM performs only marginally worse when reducing the percentage of nodes sampled from 15% to 5%. Second, the sampling can be aggregated using ultrapeers, where each ultrapeer aggregates the file information of its leaf nodes, and is responsible for sampling other nodes through their ultrapeers. This technique would reduce the sampling overheads by a factor k, where k is the average number of leaf nodes supported by an ultrapeer. Third, the sampling can be done over a long period of time by monitoring the files via the Gnutella traffic, at the expense of being less accurate.

7 Implementation and Deployment

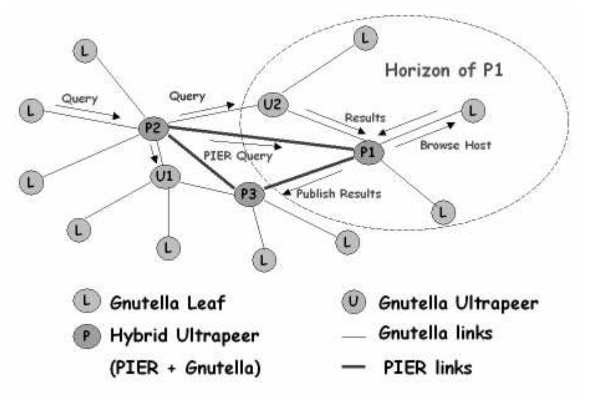

Figure 16: *Strawman Deployment on a selected set of Ultrapeers*

In this section we report on our initial deployment of the hybrid design we motivated above. Our goal in deploying a real implementation were to ensure that PIERSearch worked on live Gnutella queries at a non-trivial scale, and to get initial validation of our hypotheses about the benefits of the design. To evaluate the hybrid design, we deployed fifty hybrid LimeWire/PIERSearch clients on PlanetLab, which participate on the Gnutella network as ultrapeers. Figure 16 shows the *partial deployment* used in our experiments on PlanetLab. Unlike a *full deployment* scenario where all nodes in the system must be upgraded to run our hybrid ultrapeer, our deployment was feasible (given PlanetLab) and backward-compatible with the installed Gnutella base. Though our deployment is modest compared to Gnutella in the large, we will see that it provides significant benefits. Each hybrid ultrapeer that we deployed consists of the following main components (Figure 17):

- **Gnutella Ultrapeer.** The Gnutella Ultrapeer is based on our modified LimeWire Ultrapeer (Section 4). It participates in the Gnutella network, and from the perspective of other Gnutella nodes in the network, the hybrid ultrapeer behaves like an ordinary Gnutella ultrapeer. Our modified Limewire ultrapeer forwards both file information and queries to the *Gnutella proxy* described below. The file information is gathered via a number of mechanisms: we fetch the list of local files, fetch lists of files at neighboring nodes (accessible via Gnutella's *BrowseHost* API), and snoop file information that the LimeWire ultrapeer sees in the responses to queries it forwards on behalf of the Gnutella network[7]. The queries are also snooped from the Gnutella traffic, and can be queries issued by the leaf nodes of the local ultrapeer, or queries forwarded by the ultrapeer on behalf of neighboring ultrapeers.

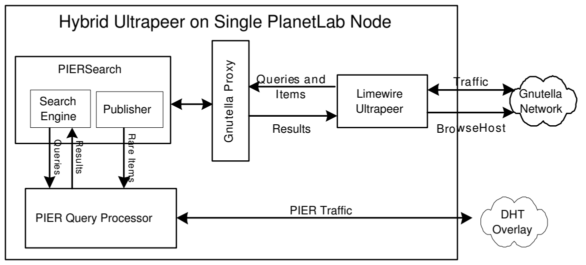

Figure 17: *Hybrid Client Implementation on a Single PlanetLab node.*

- **Gnutella Proxy.** The proxy accepts queries and file information from the Gnutella ultrapeer. The file information is filtered for rare items, which are sent to the PIERSearch client. Queries are also selectively sent by the proxy to be reissued via the PIERSearch client.
- **PIERSearch client.** The PIERSearch client receives rare items from the Gnutella proxy, and constructs the *Item* and *Inverted* tuples which are published into the DHT. Similarly, it receives queries from the proxy, formulates the query described in Section 3, which it then sends to PIER for execution.
- **PIER client.** Our DHT-based query engine, PIER utilizes the Bamboo [21] DHT. Hence, the hybrid client participates in two separate networks: the Gnutella network and the Bamboo DHT overlay.

In our deployment, each LimeWire ultrapeer monitors query results from its regular Gnutella traffic. These query results are responses to queries forwarded by the ultrapeer. Query results that belong to queries with fewer than 20 results are identified as rare items, and sent to PIERSearch for publishing. This scheme is based on the QRS rare item scheme (Section 5), which we chose because it is easy to implement in the *partial deployment* model.

We begin by describing the behavior of PIERSearch publishing in our experiments. The publishing rate we observed was approximately one file per 2-3 seconds per node. Each published file and corresponding posting list entries incurred a bandwidth overhead of 3.5 KB per file. We also

[7]In a full-deployment scenario, forwarded query results would not need to be sent to the proxy as each ultrapeer would be only responsible for indexing files for itself and its leaves.

tested the *InvertedCache* option, which increased the publishing overhead to 4 KB per file. A large part of the bandwidth consumption in PIERSearch publishing today is due to the overheads of Java serialization and self-describing tuples in PIER, both of which could in principle be eliminated.

Next, we consider the latency benefits that PIERSearch brought to Gnutella in our deployment. We tested the hybrid search technique in PlanetLab on 1739 leaf queries of the hybrid ultrapeers. In our implementation, leaf queries that return no results within 30 seconds via Gnutella are considered to have "timed-out", and are re-queried by PIERSearch. In our experiments, PIER executed the query and returned the first result within 10 seconds with the *InvertedCache* option, and 12 seconds without. While decreasing the timeout to invoke PIER would improve the aggregate latency, this would also increase the likelihood of issuing queries in PIER. As part of our future work, we plan to study the tradeoffs between the timeout and query workload.

Note that the average latency for these queries to return their first result in Gnutella is 65 seconds (see Figure 7). Hence, the hybrid approach with a 30-second timeout would reduce the latency by about 25 seconds.

We also measured the bandwidth overheads of querying with PIERSearch. Using the *InvertedCache* option, each query needs to be sent to only one node. The cost of each query is hence dominated by shipping the PIER query itself, which is approximately 850 bytes. The distributed join algorithm incurs an average of 20 KB overhead for each query. Considering these numbers as well as the publishing costs and latencies reported above, the benefits of reducing per-query bandwidth seem to outweigh the publishing overheads of storing the filename redundantly, making *InvertedCache* a more attractive option for our scenario.

Finally, we consider the benefits in answer quality that resulted from our partial deployment. Our experiments show that the hybrid solution reduced the number of queries that receive no results in Gnutella by 18%. This reduction serves as a lower bound of the potential benefits of the hybrid system. The reason why this value is significantly lower than the potential 66% reduction in the number of queries that receive no results is twofold:

- Unlike the Gnutella measurements reported in Section 4.2 where queries are proactively flooded from many ultrapeers, in our experiment, we consider only the files that are returned as results to previous queries. Thus, this scheme will not return the rare items that were not queried during our experiments. Employing other schemes for identifying rare items described in Section 5 in conjunction with peers proactively publishing their list of rare items should considerably boost the benefits of the hybrid infrastructure.
- As the number of clients that implement our scheme increase, we expect the coverage to improve as well. The coverage would be even better in a full-fledged implementation in which each ultrapeer would be responsible for a set of leaf nodes from which they would identify and publish rare items.

8 Related Work

A survey of distributed database research can be found in [18]. To our knowledge, the distributed database system that targeted the largest number of nodes was Mariposa, which envisioned scaling to "1,000 sites or more" [25]. Of course the distributed database literature typically targets much broader functionality than what is offered in peer-to-peer filesharing, including flexible schemas, general SQL queries, and transactional storage.

A goal of our work on PIER is to study the challenges in scaling to many more nodes, while relaxing some of the design requirements of traditional distributed databases [11, 12]. Our current incarnation of PIER uses the Bamboo DHT [21], not the CAN DHT described in earlier papers. The simple join algorithm we describe in Section 3 of this paper is also not discussed in our earlier papers; it arose naturally in the context of filesharing workloads.

The seeds of this paper were presented in a recent workshop [16], including some of the Gnutella measurements we present here in Section 4. The workshop paper focused largely on the case for building a hybrid search infrastructure. In this paper, we expand upon the workshop paper significantly by presenting the architecture, implementation and deployment results for our PIER-based hybrid ultrapeers. This paper proposes and analyzes solutions to identifying rare items, a problem that was left unsolved in the earlier paper. The Gnutella results we present here also flesh out some issues that were unclear in the workshop paper, including the separation of the QR and QDR metrics, and results for the QDR metric.

A recent study [4] has shown that most file downloads are for highly-replicated items. One might think that their findings contradict our analysis in Section 4.2 that shows that queries for rare items are substantial. However, the two studies both correctly reflect different aspects of the Zipfian distributions. Their study shows the *head* of the Zipfian popularity distribution, and hence they measure the download requests based on the items that match the top 50 query requests seen. In contrast, our study focuses on the long *tail* of the distribution as well. While individual rare items in the tail may not be requested frequently, they represent a substantial fraction of the query workload, and are therefore worth optimizing.

There have been other recent proposals for P2P text search over DHTs [26, 6]. A feasibility study on DHT-based P2P web search [14] focuses on the more demanding web corpus (3 billion documents) and a larger query rate (1000 queries per second). There has also been work done on optimizing search performance in unstructured networks [4, 27, 5], mostly to address the shortcomings of flooding. Our hybrid infrastructure offers a simple alternative to either optimizing searching in structured or unstructured networks, by combining the strengths from both networks.

As an alternative to our hybrid infrastructure, there is a proposal [3] to build a Gnutella-like network where nodes are organized using a structured overlay. The authors argue that building Gnutella using structured overlays lead to improved performance of floods and random walks, and also can be used to reduce maintenance overheads.

9 Conclusion

In this paper, we have presented PIERSearch, a P2P search engine that utilizes PIER, a DHT-based query processor. We proposed a hybrid search infrastructure that utilizes flooding for popular items and PIERSearch for indexing and querying rare items. To support our case, we performed live measurements of the Gnutella workload from different vantage points in the Internet. We found that Gnutella is highly effective for querying popular content, but ineffective for querying rare items. A substantial fraction of queries returned very few or no results at all, despite the fact that the results were available in the network.

A key challenge for the hybrid search infrastructure is in identify rare items for publishing. Using a model for hybrid search and Gnutella traces, we study the tradeoffs of improving query recall and publishing overheads introduced by the hybrid infrastructure. Our experiments show that building a partial index over the least replicated items can improve query recall dramatically, especially when the query does not require multiple copies of the same item within its results set. On top of that, there are diminishing returns for indexing more popular items in the DHT that can already be found via flooding.

Based on our model and Gnutella traces, we also compare different schemes for identifying rare files for selective publishing by the hybrid nodes into the DHT. Our evaluation shows that the localized schemes that we proposed for identifying rare items compare favorably to a perfect baseline that assumes global knowledge of the system. Our deployment of fifty hybrid ultrapeers on Gnutella shows that our hybrid scheme has the potential to improve the recall and response times when searching for rare items, while incurring low bandwidth overheads.

10 Acknowledgements

The authors would like to thank all members of the PIER project group and Berkeley Database research group for their insights and suggestions. We thank the anonymous reviewers for their comments. This research was funded by NSF grants ANI-0225660 and IIS-0209108.

References

[1] Gnutella Protocol v0.6 protocol specification. `http://groups.yahoo.com/group/the_gdf/files/Development/`.

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

[3] M. Castro, M. Costa, and A. Rowston. Should we build Gnutella on a structured Overlay? In *HOTNETS 2003*.

[4] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P Systems Scalable. In *Proceedings of ACM SIGCOMM 2003*.

[5] A. Crespo and H. Garcia-Monila. Routing Indices for Peer-to-Peer Systems. In *ICDCS*, 2002.

[6] O. D. Gnawali. A Keyword Set Search System for Peer-to-Peer Networks. Master's thesis, Massachusetts Institute of Technology, June 2002.

[7] Gnutella. `http://gnutella.wego.com`.

[8] Gnutella Proposals for Dynamic Querying. `http://www9.limewire.com/developer/dynamic_query.html`.

[9] Query Routing for the Gnutella Network. `http://www.limewire.com/developer/query_routing/keyword\'routing.htm/`.

[10] Gnutella Ultrapeers. `http://rfc-gnutella.sourceforge.net/Proposals/Ultrapeer/Ultrapeers.htm`.

[11] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, March 2002.

[12] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, Sep 2003.

[13] Kazaa. `http://www.kazaa.com`.

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

[15] Limewire.org. `http://www.limewire.org/`.

[16] B. T. Loo, R. Huebsch, I. Stoica, and J. Hellerstein. The Case for a Hyrid P2P Search Infrastructure. In *IPTPS 2004*.

[17] M. Mitzenmacher. Compressed Bloom Filters. In *Twentieth ACM Symposium on Principles of Distributed Computing*, August 2001.

[18] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems, Second Edition*. Prentice Hall, 1999.

[19] PlanetLab. `http://www.planet-lab.org/`.

[20] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[21] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. *UC Berkeley Technical Report UCB//CSD-03-1299*, Dec 2003. Revised version to appear in 2004 USENIX Annual Technical Conference, June-July, 2004.

[22] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[23] P. Seshadri and A. N. Swami. Generalized partial indexes. In *ICDE*, pages 420–427, 1995.

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM 2001*, pages 149–160. ACM Press, 2001.

[25] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB J.*, 5(1):48–63, 1996.

[26] C. Tang, Z. Xu, and M. Mahalingam. pSearch: Information retrieval in structured overlays. In *ACM HotNets-I*, October 2002.

[27] B. Yang and H. Garcia-Molina. Efficient Search in Peer-to-Peer Networks. In *ICDCS*, 2002.

[28] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.