Efficient Search in File-Sharing Networks

Paul Burstein
Alan J. Smith

Efficient Search in File-Sharing Networks [*]

Paul Burstein and Alan Jay Smith
Computer Science Division, EECS Department
University of California Berkeley
Berkeley, CA 94720-1776
(burst, smith)<at>eecs.berkeley.edu

Abstract

Currently, the most popular file-sharing applications have used either centralized or flood-based search algorithms. Napster has been successful in providing a centralized index with presumably perfect query recall. Succeeding, more distributed protocols like Gnutella and Kazaa have used flood-based search procedures in which a query is propagated through an unstructured network. Query result quality in such networks suffers as queries for items that are rare have high probabilities of not being found as the entire corpus is not covered during a search. In this paper, we present a new and improved implementation of a distributed file-sharing system yielding (1) query result quality better than flooding and close to a centralized index, and (2) low-maintenance network overhead. These improvements result from our optimized approaches to (a) high churn rates (clients and servers frequently entering and leaving the system) and (b) skewed workloads (high variation in access frequencies vs. key). High churn rates are addressed by keeping all data in soft state, which is periodically refreshed, such that the loss of a server or client is quickly reflected in the indexes; higher refresh rates imply fewer false positives. Skewed workloads are load balanced with the use of a layer of indirection for placing and locating data, such that data is partitioned and distributed based on the frequency of use. A trace-driven prototype evaluation based on Gnutella system traces shows that our prototype implementation achieves a low network bandwidth, attains max-average load ratios within a factor of three across all servers, and has positive recall values for over 90% of all queries, despite a high churn rate; the recall would be 100% absent churn.

1 Introduction

Keyword-based search has become an integral part of today's computing experience. One major application is that provided by Internet search engines offering full-text search over web content. The second major application consuming a substantial fraction of overall Internet traffic today is peer-to-peer file-sharing, offering keyword search over file metadata, including file name, type and author. A computer running file-sharing software provides a service which lets other peers download files stored in its public directory. A file-sharing network lets its users submit a query containing several keywords and returns pointers to peers with files which contain the keywords provided with the query in their metadata. The user can then examine the pointers and follow any of them to download a given file.

[*]Primary support for this research has been provided by AT&T Laboratories, with additional support by the University of California MICRO Program and Toshiba Corporation.

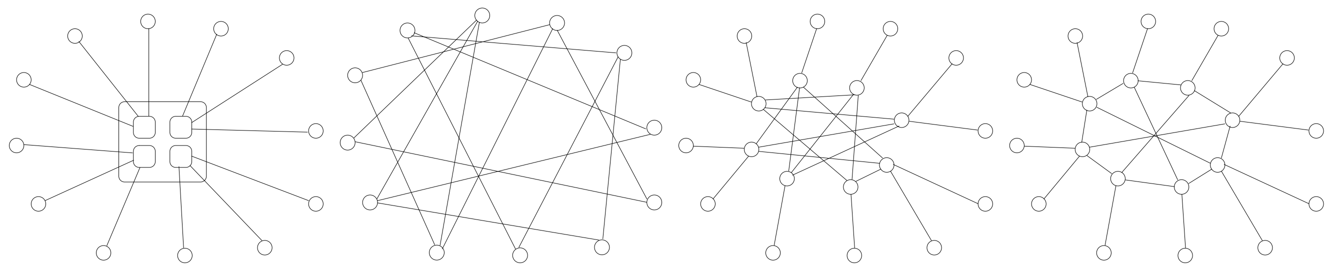
Typical file-sharing workloads play an important role in the design of file-sharing systems. The systems generally experience a large amount of churn. The composition of peers is bimodal, ranging from server-like nodes that stay online for days at a time to client-like consumers which join the networks for a few hours to run some queries and perform quick downloads while offering their data in the mean time. These high churn rates create frequent updates to the indexing structures as clients enter and leave the network. The typical distributions of keywords are highly skewed on both the querying and the metadata publishing fronts. We propose a system that explicitly deals with both churn, by keeping all the data in soft state and implementing an efficient update protocol, and the skewed workload by load balancing with the help of an additional layer of indirection which provides for location independence and load distribution. The keyword space is partitioned such that each inverted index for a particular keyword resides on one or a handful of nodes. A layer of indirection is used for locating the precise nodes. Thus, our system distributes data in such a way that finding exactly a query target can be accomplished with a small number of lookups.

In this paper, we present a new and improved implementation of a distributed file-sharing system yielding (1) query result quality better than flooding and close to a centralized index, and (2) low-maintenance network overhead. While others have proposed solutions for structured distributed indices, the problem of balancing load has not previously been addressed. Our work concentrates on the problem of load balancing within the index, which occurs due to the uneven distribution of keyword popularity in the publishing and query workloads. We achieve this through an additional layer of indirection which allows for more flexible allocation of storage and search resources. In this work we present, analyze and evaluate a structured distributed index protocol for file sharing. We evaluate a prototype implementation on PlanetLab, a global-scale test bed for deploying large distributed services. A trace-based evaluation shows that our prototype achieves good levels of load balancing while maintaining a reasonable network overhead, roughly that of streaming video. The prototype returns results for over 90% of all queries with 93% of all queries requiring less than 10 seconds to complete.

The rest of this report is organized as follows. In Section 2 we provide some background and discuss the related work. In Section 3 we present the system design overview, the details of its soft-state protocols, and a cost analysis model for evaluating the overhead associated with our protocols against varying system workloads. Section 4 addresses the issue of load balancing due to the typical non-uniformity of file sharing query and keyword distributions. In Section 5 we present experimental data evaluating our prototype and conclude in Section 6. In the Appendix, we present our analysis of the Gnutella network traces, provide the derivations behind our analytical model, and discuss and evaluate our distributed hash table implementation.

2 Related Work

The general solution space for solving the distributed search problem can be broken into four categories represented in Figure 1. In a centralized index system a cluster of nodes is responsible for maintaining an index of what files every file-sharing client stores. Each client connects to one or more machines in the



(a) Centralized Index (b) Unstructured Flooding (c) Supernode Flooding (d) Supernode Index

Figure 1: Solution space

cluster and publishes its local index once. Queries get resolved within the centralized cluster. The query result quality is ”absolute” as the whole index can be searched to resolve any incoming query. The benefit of the centralized index is that it incurs minimal publishing costs and produces full query results. The prototypical example of a centralized cluster was Napster in its original form with a cluster of about 160 machines. A typical centralized index is depicted in Figure 1(a). In a pure flooding system all nodes are equal. Each has some number of neighbors and all the nodes are organized in an unstructured network as shown in Figure 1(b). A query by a node gets forwarded to all of its neighbors, and they in turn forward it to theirs. Each query has an associated time-to-live, specified in number of hops, and only gets propagated that many hops away from the initiator of the query. The result quality grows with the number of nodes that are visited as part of the query flood but is usually only partial. Later versions of the protocol have evolved into a two tier structure in which peers are split up into regular nodes and more capable supernodes as shown in Figure 1(c). In this scheme queries get propagated only among the supernodes but the general problems of poor recall are still present.

A substantial amount or work has been done in the area of metadata search for flood-based file-sharing systems [CGM02, YGM02, CRB$^+$03, CCR04, GFJ$^+$03]. Many of the papers either have not fully considered the appropriate overheads, load balancing issues, or have not focused on search quality. Several solutions have been proposed for using an index based approach, but have usually been in the space of full-text search [RV03, TXD03, TD04, LLH$^+$03, CLL04]. These schemes utilize the structure of distributed hash tables [SMK$^+$01b, RFH$^+$01, RD, ZKJ01], and a typical two-tier setup is depicted in Figure 1(d). More recent work [LHH$^+$04, ZH05] suggests using a DHT-based index in conjunction with current flood-based approaches to achieve better overall query result quality. Caching in flood-based as well as index-based file-sharing systems has been the subject of several works [CS02, LCC$^+$02, RB02]. Caching can be considered a complementary addition to the problem of distributed search and we do not explicitly concentrate on it in our work. We do however use a moderate amount of structured replication to distribute load and achieve limited caching behavior. Measuring and understanding typical file-sharing workloads has been the interest of several studies [GDS$^+$03, GSG02, KLVW04]. In our work we leverage on the results of these studies as well as the analysis of traces from the Gnutella network, detailed in Appendix A to address the skew in distributions of keywords and queries, the primary reason behind the need for inherent load balancing in a structured file-sharing search system.

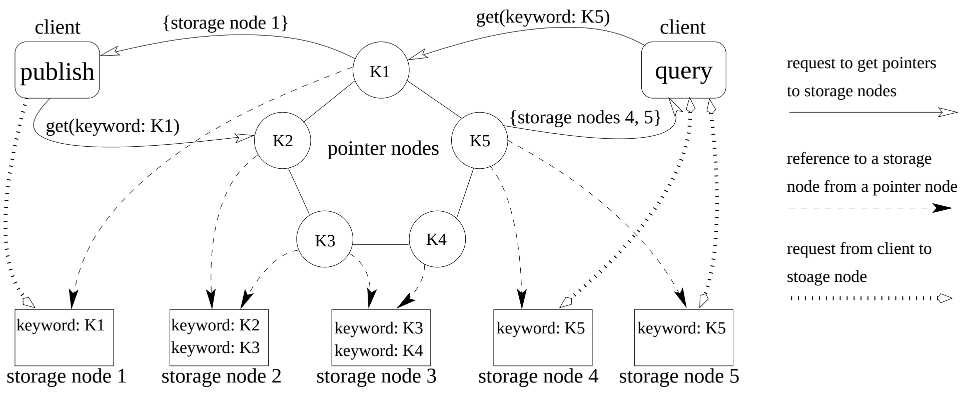


Figure 2: Design Overview

3 Design Overview

In this section we present the design of our distributed file-sharing index. We define the relevant terminology and in subsequent sections we describe the protocols used, and analyze the overheads of those protocols. In our system, users share files. They provide access to files they hold, and request files held by others. Requests for files held by others consist of queries containing keyword strings. Files are identified by their metadata. When a keyword string matches the metadata for a file, then a hit occurs and the matching file is a target being searched for.

There are two types of nodes in this system: clients and servers. Clients store files, provide files to others and make requests for files held by others. Servers perform two functions: (a) Each server node holds an index for a subset of all possible keywords. This index contains a copy of the metadata for every file in the overall system containing the indexed keyword, and a pointer to the client holding the file. (b) The servers are linked as a distributed hash table (DHT) (hashed on keyword) in which the *key* is a keyword and the *value* is the address of a server storing an index for that keyword. A server when performing function (a) is called a ”storage node” and a server when performing function (b) is called a ”pointer node.”

A client seeking a given file presents a keyword string to a storage node that indexes at least one of those keywords. The storage node, since it has the entire metadata for each file with that keyword, can do a full compare between the keyword string and the file metadata, and will return a pointer to every matching file. Thus only one storage node access is needed in order to locate target files - there is no need to search separate storage nodes on separate keywords and then do a join.

In practice, as further explained in Section 4, for load balancing purposes a given keyword index may appear on more than one storage node. In that case, each of those storage nodes will have to be accessed in order to get a full list of target files. Further, a given keyword is not bound to a given storage node - the DHT is used to find the storage node(s) for a given keyword. This latter feature allows for load balancing, and the dynamic addition and deletion of storage nodes from the system.

The pointer nodes are organized in a circular distributed hash table (dht), based on chord [SMK$^+$01b]. Each node has a unique 160-bit identifier and is responsible for storing key/value pairs for all the keywords that hash closest to its identifier. Since pointer nodes are organized in a dht, a client may contact any pointer

Entity	**Exposed Interface**	**Usage Description**
Storage Node	publish(keyword, client, entries)	*Client* publishes the inverted index entries for the given keyword
	update(keyword, client)	*Client* updates the TTL for its entries on this storage node
	query(query)	*Client* queries the storage node
Pointer Node	put(keyword, storage_node)	*Storage Node* inserts/updates its reference for the given keyword
	get(keyword)	*Client* retrieves pointers to *Storage Node*(s) for the given keyword

Table 1: Exposed interfaces

node to perform a lookup request, which gets forwarded to the appropriate pointer node within the dht. A client therefore only needs to know the location of only one server node. We describe the full operations of our implementation of the distributed hash table in Appendix C. The load balancing aspect is discussed separately in Section 4.

Clients wishing to make files available to the community ”publish” the metadata for those files. The publishing process is depicted in Figure 2. To publish its local contents a client node first needs to construct an inverted index based on the files it wants to make available. For each keyword, the inverted index contains a list of local files that contain the keyword. For every keyword, the client needs to publish the corresponding list to any storage node responsible for that keyword. In some cases several storage nodes may be responsible for a particular keyword if the entries for that keyword need to be distributed for load balancing purposes. In the example in Figure 2, the client performs a pointer lookup in the dht for keyword $K1$, receives a list consisting of storage node 1, and publishes its index for keyword $K1$ directly to storage node 1. In the rare instance when there's no storage node associated with the keyword or if all the associated storage nodes are at their full capacity, the client may pick any other storage node.

The querying process is also depicted in Figure 2. A client issuing a query for a set of keywords contacts the storage node for any one of the keywords and requests it to resolve the query. If there are several storage nodes responsible for the queried keyword, then to get the full response, the client needs to contact each of the appropriate storage nodes. To find the storage node(s) responsible for keyword $K5$, the query client in Figure 2 asks any available pointer node. It receives a response directly from the pointer node responsible for $K5$. The routing of the lookup request is done within the distributed hash table (see Appendix C). In this example, the client sends its query to storage nodes 4 and 5, which contain index entries for keyword $K5$. Even if the query contains several keywords, it is fully evaluated at the storage nodes responsible for any of the keywords in the query. The entries stored at storage nodes contain all of each file's metadata allowing for the query to be fully evaluated locally.

3.1 Soft State Protocols

In this section we describe the soft state protocols used to deal with churn. Any of the three types (client, pointer, storage) of nodes might leave the system at any point in time and without warning. Our protocol is explicitly designed to deal with churn by keeping all of the information in soft state. If a client leaves the network, the inverted index entries for that client's files need to be invalidated on all the storage nodes that contain them. If a storage node goes down, two actions must take place to reflect that all the keyword entries that it was storing are now lost. (1) The pointers to the no-longer present storage node need to be invalidated on the pointer nodes. (2) The indexed metadata that was lost needs to be recreated to reflect the clients that are still online. Finally, if a pointer node disconnects from the network, all of the pointers that it was maintaining are gone and need to be recreated on some other pointer node.

The pointer nodes provide a distributed hash table put/get interface as shown in Table 1. Each pointer to a storage node has an associated time-to-live and is removed from the pointer node if not updated by the storage node. This ensures that (a) storage node failure is reflected on the pointer nodes and that (b) there are few pointers to storage nodes that are no longer present. In order to guard against lost pointers due to pointer node failure, the storage nodes periodically update their pointers via the `put` interface. The `put` always gets routed to the *currently* responsible pointer node, which accounts for pointer node failure. An entry for each keyword is updated periodically by each storage node at a configurable period on the order of minutes. The process of storage nodes updating their pointers ensures that their pointers are refreshed. This exchange accounts for resolving pointer node as well as storage node failures.

The client and storage nodes engage in a protocol to ensure that (a) storage node failure is detected by the clients and that (b) client departure is detected by the storage nodes. The interface exposed by the storage nodes is shown in Table 1. After a client `publish`es its entries to a storage node, it needs to make sure that the entries remain there and are not lost if the storage node suddenly departs. The client does so by periodically issuing an `update` call directly to the storage node. If the storage node is found to be gone, then the client needs to republish his information. Likewise, the storage node needs to make sure that the client whose metadata it's storing is still available and that it's not storing stale data. Clients' entries have a ttl and entries that haven't been updated for a set period are discarded. A process of clients periodically pinging storage nodes mitigates both of the above mentioned problems. The client ensures that the storage node is still up and is using its data to satisfy queries. The storage node is certain that its clients are still alive and that is does not need to discard their metadata.

3.2 Protocol Overhead

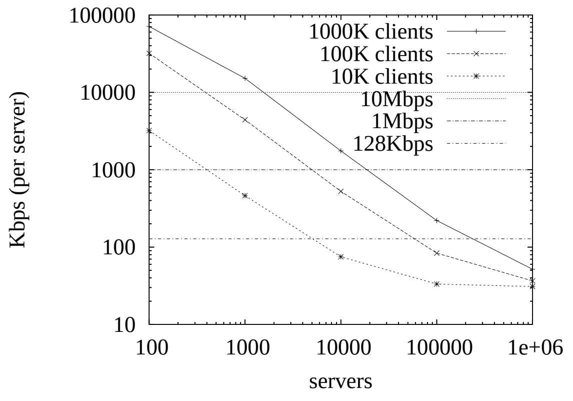
In this section we go through the steps involved in the normal operations of the pointer, storage and client protocols. We are expecting churn to be part of normal operations and it is accounted for in the protocols and their overheads. We compute the overhead of each protocol by calculating resulting per server network load in terms of bytes per second. We feel that total traffic is a good estimate because the real processing in our system comes from sending and receiving update messages. We feel that this is indicative of the amount of work that each entity needs to perform and also reflects the overall load injected into the network. Table 2 lists all the relevant system parameters, and shows how they effect the individual parts of our protocol. Two sample workloads are provided, one which we used to evaluate the system on PlanetLab (see Section 5) and one typical of real large scale setups such as the Gnutella network. Note the orders of magnitude disparity in the number of servers between the two workloads and the scaling that our system

Parameter	**Description**	**Sample A**	**Sample B**
C	Clients	130	1,000,000
S	Servers	76	100,000
P_I	Pointer Period (seconds)	2	2
P_S	Store Period (seconds)	120	120
P_C	Client Period (seconds)	240	240
P_Q	Query Period (seconds)	60	60
L	Average Message Length (Bytes)	100	100
L_Q	Query Response Length (Bytes)	500	500
r	Replication Factor	4	8
K	Total Keywords	50,000	10,000,000
K_C	Keywords / Client	1000	1000
F_C	Files / Client	600	600
K_F	Keys / File	5	5
M	Metadata Size (Bytes)	50	50
U_C	Client Uptime (seconds)	2400	3600
U_S	Server Uptime (seconds)	86400	86400
Overhead	Computation	KBps	KBps
POINTER	$(4 + 4r + 2log(s)) * L/P_I$	**1,625**	**3,461**
STORAGE	$(2 * log(s) + 4 * (r - 1)) * L/P_S * K/S$	**13,430**	**5,102**
Lookup	$(2 * log(s)) * L * K_C/U_C * C/S$	891	9,228
Transfer	$(F_C * K_F * M)/U_C * C/S$	107	417
PUBLISH	$Lookup + Transfer$	**998**	**9,644**
Churn	$PUBLISH * U_C/U_S$	28	402
CLIENT	$Churn + (2 * L * min(K_C, S))/P_C * (C/S)$	**136**	**8,735**
QUERY	$((2 * log(s) * L) + L + L_R)/P_Q * (C/S)$	**53**	**654**
TOTAL		16,241 KBps	27,596 KBps
TOTAL		129,927 Kbps	220,765 Kbps

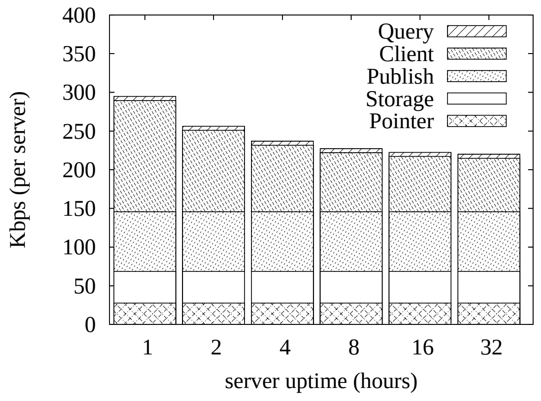
Table 2: Overhead summary

achieves with a less than two fold increase in load. For a full explanation of the system parameters and complete derivations we refer the reader to Appendix B. The five main costs are summarized in Table 3.

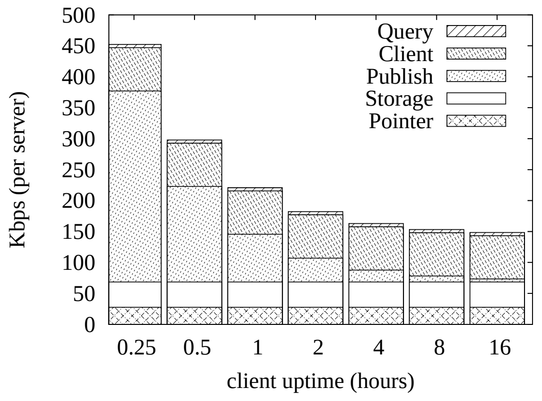
Three of our costs, the publish overhead, client overhead and query overhead, depend linearly on the client-to-server ratio (Table 2). In Figure 3(a) we observe the effective per server load as a result of varying the ratio of clients to servers in the system. We can note that as the ratio of clients to servers increases, the overhead quickly reaches unacceptable levels, far beyond those sustainable by common DSL or cable lines. The load is however reasonable for a server to maintain when the client-to-server ratio is on the order of 10. Such a ratio is typical to expect in a peer-to-peer system, where a large fraction of participants act as servers. Keeping the number of clients and servers stable at 1 million clients and 100,000 servers, (using Sample B in Table 2) we vary a number of other crucial system parameters. In Figure 3(b) we vary the average server uptime from one hour to about one day. Small server uptimes induce churn in the system and have a direct effect on the client overhead (due to continuous publishing), which in the model is the



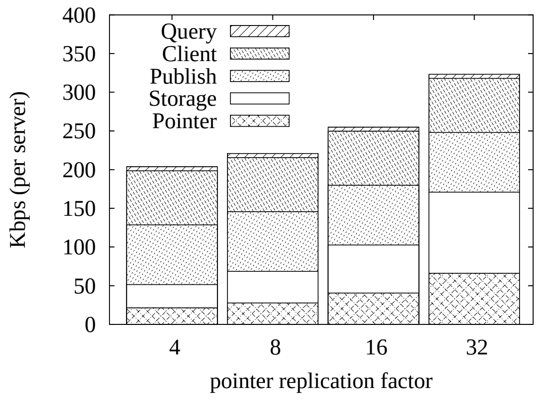
(a) Varying the client/server ratio



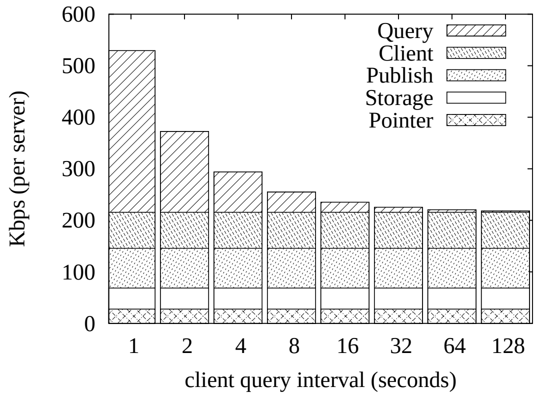
(b) Varying server uptime



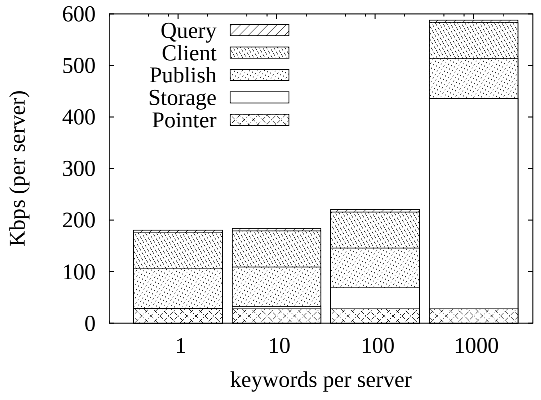
(c) Varying client uptime



(d) Varying the replication factor



(e) Varying query interval



(f) Varying number of keywords

Figure 3: Protocol Overhead. Figures (b)-(f) consider a setup with 1,000,000 clients and 100,000 servers.

only quantity affected by server churn. Typical server uptimes of at least one day are reasonable to expect [GSG02]. Varying the client uptime has a direct effect on the publishing cost as shown in Figure 3(c). In our model we assume that the average number of clients up at any time remains stable. As the client uptimes become shorter and more clients enter the system on average, more data needs to be published.

Cost	**Description**
QUERY	cost due to resolving the queries generated by clients
CLIENT	the cost of clients refreshing their index entries on the storage nodes, and recreating lost entries due to storage node failure (affected by server churn)
PUBLISH	the cost of clients publishing their shared metadata when they join the network (affected by client churn)
STORAGE	cost of the storage node interaction with the pointer nodes during the pointer update protocol
POINTER	cost of the dht maintenance protocol among the pointer nodes

Table 3: Cost summary

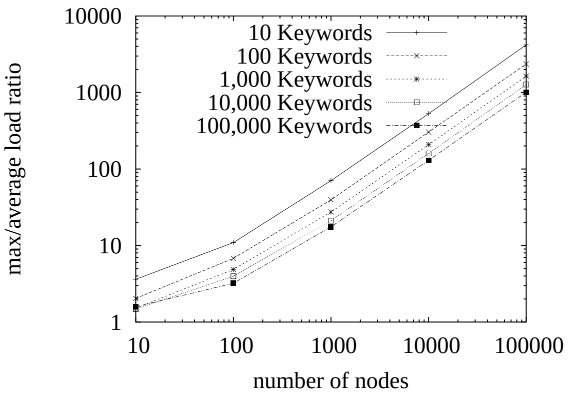
The expected average client uptime is about one hour [GSG02].

Within the dht, each pointer to a storage node is replicated on a fixed number of adjacent pointer nodes. The replication factor improves load balancing and increases response time and is fully discussed in Section 4 and Appendix C. Changing the pointer replication parameter in Figure 3(d) affects only the pointer overhead and the storage overhead. The rate at which clients issue queries has a linear effect only on the query cost. Figure 3(e) shows the effect of changing the average number of queries issued by clients from one per second, to one in two minutes. Finally, in Figure 3(f) we look at the effect which the total number of distinct keywords has on the system. The number of distinct keywords effects the number of keywords that each storage node is responsible for and that has a linear effect on the storage node overhead.

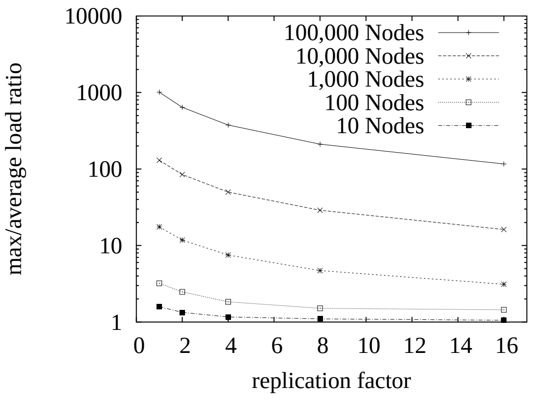
4 Load Balancing

In this section we address the problem of balancing load in our system. Specifically, we look at two problems of (1) balancing the query load among pointer nodes and (2) balancing the storage load among storage nodes. By query load we mean the number of lookup messages received by each pointer node. By storage load we mean the number of index entries stored per storage node.

The first step of each client query involves obtaining pointers to storage nodes from the pointer dht. Each pointer node is responsible for storing pointers and resolving queries for roughly the same number of keywords. However, if the distribution of queries per keyword is not uniform, pointer nodes responsible for the more popular keywords will be likely to receive more lookup requests. To deal with this problem, each pointer is replicated on a fixed number of adjacent pointer nodes. We show that this provides a moderate improvement in the pointer node load distribution. On the other hand, the number of entries that a storage node stores for a given keyword depends on the number of files in which that keyword appears. If the distribution of files per keyword is not uniform, the number of index entries for different keywords may be drastically different. The amount of storage space that each storage node would need to contribute would depend directly on which keywords it's storing entries for. To deal with this problem we split the larger keyword entries and distribute them among several storage nodes, while using indirection through the pointer layer to identify where the split entries reside.



(a) Query load for various number of nodes and number of keywords



(b) Effects on max/average ratio as result of varying replication factor for different number of nodes and 100,000 keywords

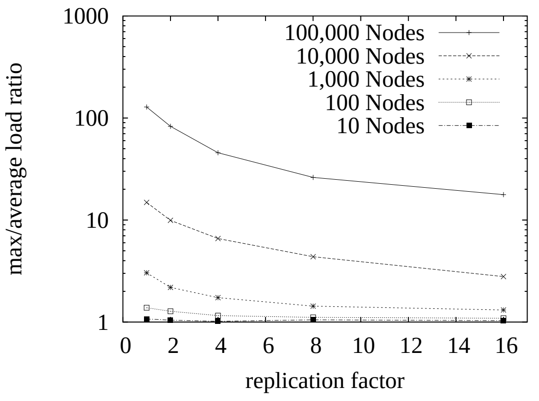
Figure 4: Evaluating query load

4.1 Query Load

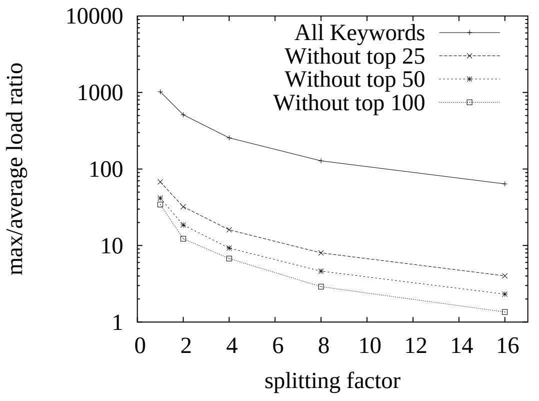
First, we observe the effect that the ratio of the number of servers to the number of keywords has on the load. If there are more keywords than nodes, each pointer node becomes responsible for several keywords at a time and that in itself has a favorable impact. In Figure 4(a) we look at the max/average load ratio as we vary the number of keywords for which queries are issued and the number of servers in the system. The Zipf distribution is sampled to drive the keyword frequency in queries (as observed in Appendix A). When the ratio of keywords to nodes is large, each node is responsible for several keywords. Therefore, there are fewer nodes that are under-loaded and the max/average ratio is rather small. As the number of nodes increases, and each node becomes responsible for a smaller number of keywords we can clearly see the effects of the uneven query distribution.

To ameliorate the imbalance, we rely on the fact that in our dht implementation each pointer is replicated on a fixed number of nodes, and contacting any one of them is sufficient to retrieve the pointer to the storage node. In Figure 4(b) we observe the improvement in load balancing max/average ratio as we vary the value of the replication factor. In this experiment we fix the number of keywords at 100,000, which is roughly the number of keywords we observed in Gnutella traces (see Appendix A) and analyze the max/average ratio. While not completely perfect, increasing the replication factor shows improvements in the load ratio.

We can further reduce the load imbalance by letting each multi-keyword query perform a lookup for the least popular of the keywords. Each client can estimate global keyword frequency by computing it over the metadata that it holds. In Appendix A we analyzed the Gnutella query traces, and for every query with more than one keyword only recorded the frequency for the least popular of the keywords. The query keyword frequency for the most selective keywords is still somewhat Zipfian but the highest appearing frequencies are reduced by an order of magnitude (Figure 12(b) in Appendix A). Figure 5(a) shows the max/average ratios derived from the traces in which a query is issued for the least popular keyword and we can observe that the max/average ratio is also reduced by an order of magnitude.



(a) Effects on max/average ratio as result of varying replication factor with 100,000 keywords



(b) Storage load balancing with replication and stop words, with 100,000 clients and 10,000 servers

Figure 5: Query and storage load distribution

4.2 Storage Load

The distribution of keyword frequencies in filenames, like the distribution of keywords in queries, resembles the Zipfian (Figure 11(c) in Appendix A). A small fraction of keywords such as file extensions and popular English words appear in a disproportionate amount of file names. To ensure that the load on the storage nodes is balanced we employ two solutions. First, the index entries for keywords with a large number of files are split, and stored on any set of storage nodes. This is achieved with the help of indirection through the dht, which can store several pointers for a given keyword. Second, we make sure that clients do not publish index entries for a fixed number of top ranking keywords, unless there are no other available keywords. While the system cannot restrict which keywords get queried, we can control which entries get published. Clients can choose which entries get published based on local index estimates or on global values, for instance the number of current storage nodes for a keyword. Storage nodes also have complete freedom in determine which index entries and how many they wish to store. If a storage node is overloaded with keyword entries, it can choose not to accept new `publish` requests from clients, and clients will find a different storage node to publish to. This can be done as we enforce no binding between a storage node and the keywords it stores.

In Figure 5(b) we look at the performance of the two schemes for distributing the storage load. This shows a simulation of a network with 100,000 clients and 10,000 storage nodes with clients having an average of 100 files with 5 keywords per file (as observed in Appendix A). We can observe the effects of increasing the splitting factor as well as increasing the number of stop words for which entries do not get published. We can see that simply increasing the splitting factor by itself or only removing stop words from being published, while effective, is not as successful as the two measures taken together. Including the two simultaneously reduces the max/average load ratio by two orders of magnitude and brings it down to reasonable limits.

Duration	5 hours
Number of servers	76
Average number of simultaneous clients	123
Total number of clients	1,108
Average client uptime	35 minutes
DHT Ping Period	2 seconds
Storage Pointer TTL	5 minutes
Storage Node Update Period	2 minutes / keyword
Client Entry TTL	10 minutes
Client Entry Update Period	4 minutes / keyword
Average Query Rate	1 query per minute / client
Queries Issued	31,284 queries

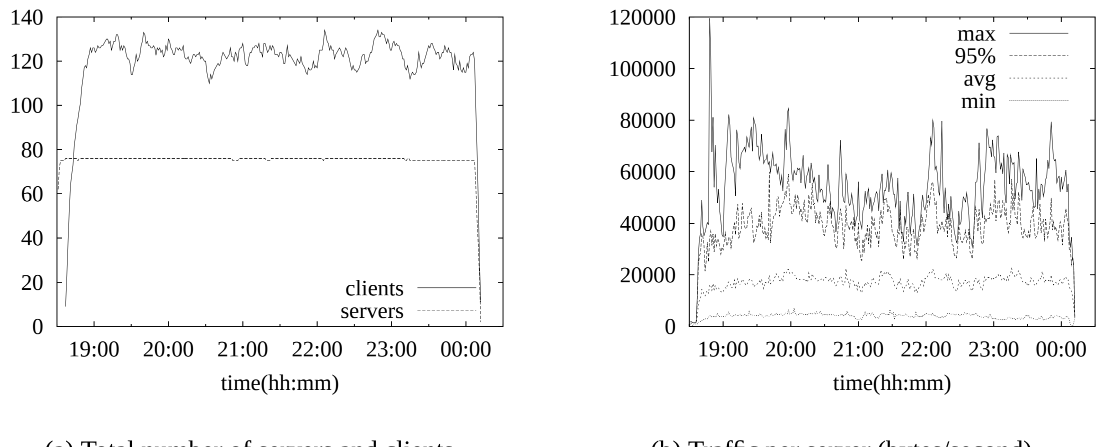
Table 4: Experimental setup and summary

5 Evaluation

A prototype of the system has been implemented and incorporates all of the elements described above, including load balancing and soft state maintenance. The prototype implementation is about 12,000 lines of Java and includes modules for the routing and pointer layers, the storage layer as well as the client. The goals of our evaluation are threefold. First, we want to observe the overhead of the protocols on a running system. Second, we want to observe the distribution of load under non-uniform query and publishing workloads. Finally, our goal is to evaluate the effective query result quality, compared against a centralized index solution.

Our evaluation is trace driven. We use existing client publishing workload utilizing the data from 603 real Gnutella clients (discussed in Appendix A). An existing query workload of 70,000 queries from [LHH$^+$04] is also used. To facilitate our experiments, we run a central distribution host which has two main responsibilities. First, it assigns publishing and query workloads to clients based on the traces. Second, it acts as a centralized server resolving queries locally to evaluate the effective recall. We utilize a single local machine to act as the distribution host. About 76 PlanetLab nodes are used to stimulate the server and client workloads. Each PlanetLab node runs a server (a pointer node and a storage node coupled together) and two client instances. A client instance continuously loops in two modes: up and down to simulate churn. In this evaluation each client instance goes up for average of 35 minutes, shuts down for average of 10 minutes and repeats. Each time a client comes up it gets a new publishing workload from the distribution host and sends it heartbeats while running in up mode. The client also receives a query workload from the distribution host, and issues the queries at a rate of one query per minute. Each time a client issues a query to our distributed index, it also resolves it with the distribution host for comparison. The experimental setup is summarized in Table 4.

The system is configured with the following parameters. The system-wide pointer replication factor is four. The pointer node update period is set to 2 seconds. Each storage node updates its entries in the dht every 2 minutes and each entry is set to expire after 5 minutes, if not updated. Storage node statistics used for load balancing, precisely the current number of entries and the number of keys, are piggybacked



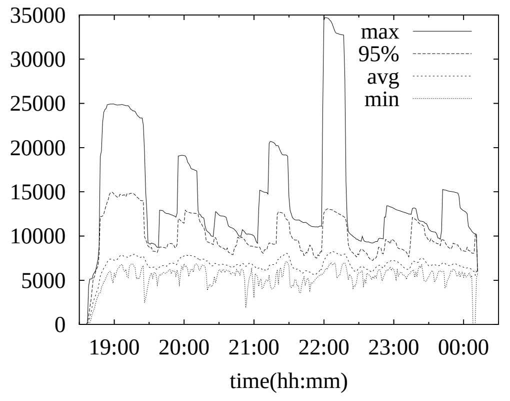
(a) Total number of servers and clients (b) Traffic per server (bytes/second)

Figure 6: Experimental results

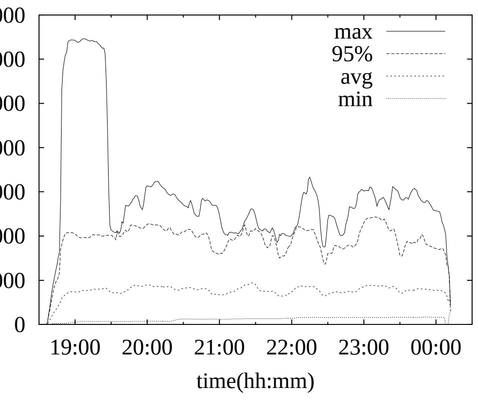
with dht ping messages as well as pointer update messages. These statistics are later used by clients for load balancing. The client update period is set to 4 minutes and client entries expire at storage nodes after ten minutes. That means that there is a lag of at most ten minutes between the time when a client goes down and the time when queries are no longer resolved against its entries. This lag time is the source of any false positives returned with query results.

Two specific policies are used by the clients, one for publishing and one for querying. When publishing its entries, clients can choose which storage node the entries for a particular keyword should go to. The clients publish entries for all keywords, no matter how many entries there are or what ranking the keyword has. When the client looks up the current pointers for a keyword, it gets a list of storage nodes currently storing entries for that keyword along with their last reported statistics. Also included is a short list of other storage nodes that the responding pointer node knows about along with their current statistics. If the keyword is not in the top 50 *locally* and the number of entries is less than 500, the client publishes to the storage node which already stores entries for that keyword. If there are several, it chooses the one with the least number of reported entries. If there are none, then it chooses a storage node from the list with the least number of entries, and the pointer nodes are updated. On the other hand, if the keyword is either in the top 50 or has more than 500 entries, the client chooses the storage node from the whole list that has the least number of entries, whether it's already storing that keyword's entries or not. For multiple keyword queries a simple local policy is implemented for choosing the least selective keyword. If all of the keywords appear locally, the client chooses the keyword with the lowest local ranking. Otherwise, it just chooses the longest length keyword.

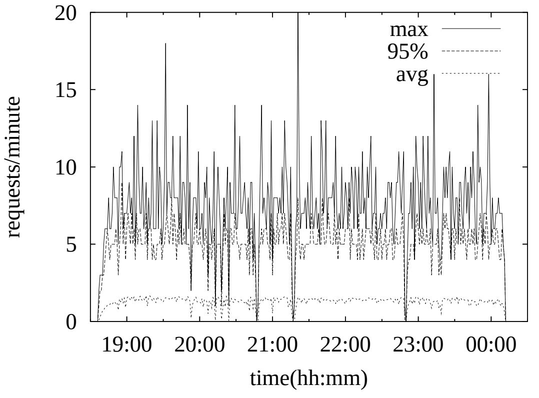
The experiment ran for a total of about five hours with an average of 76 servers and 123 clients up at any instance. Over 1,100 distinct client instances were observed with an average uptime of 35 minutes. The number of operational clients and servers aggregated in one minute intervals is shown in Figure 6(a).



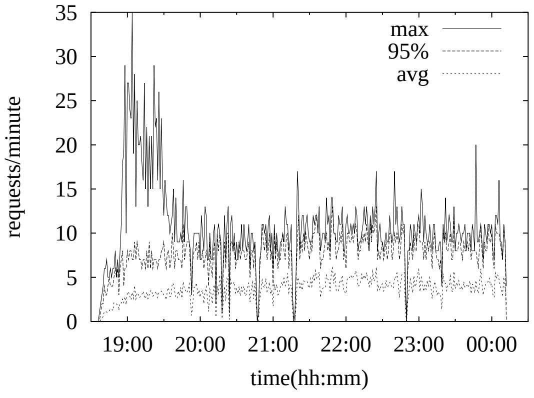
(a) Number of index entries per server



(b) Number of keys per server



(c) Query requests per **pointer** node



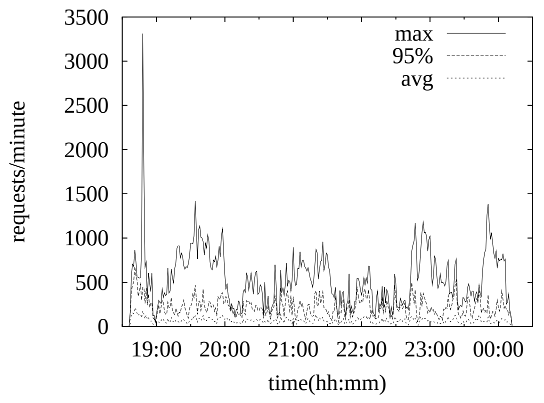
(d) Query requests per **storage** node

Figure 7: Experimental results

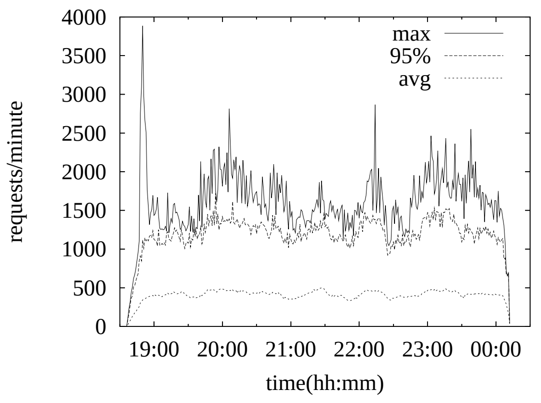
5.1 Traffic and Storage Load

The network traffic generated by the experiment is presented in Figure 6(b). For each server we capture the total number of bytes, sent and received, in a bytes/second rate, averaged over one minute intervals. This data includes the overheads of the pointer protocol and storage protocols as well as the file metadata published by clients. For each one minute interval Figure 6(b) shows the maximum, average, minimum and 95 percentile data transfer rates over all operational servers. The average amount of traffic per server was 15,888 bytes per second, which closely corresponds to the result obtained with our model of 16,241 bytes per second (derived in Appendix B and summarized in Table 2).

Figures 7(a) shows the distribution for the number of keyword entries stored at each server. We do observe some periodic spikes in the maximum number of entries per storage node. The 95th percentile is almost always within a factor of two of the average, and the minimum number of entries is extremely close to the average, and the max/average ratio hovers around two for the majority of the run. The distribution of keywords per storage node shown in Figures 7(b) has a steady 95th percentile twice above the average. The peak maximum at the beginning of the run gets resolved within the first hour and the max/average ratio is steadily below 4 for the rest of the run.



(a) Publish **get** requests per pointer node



(b) Publish **put** requests per pointer node

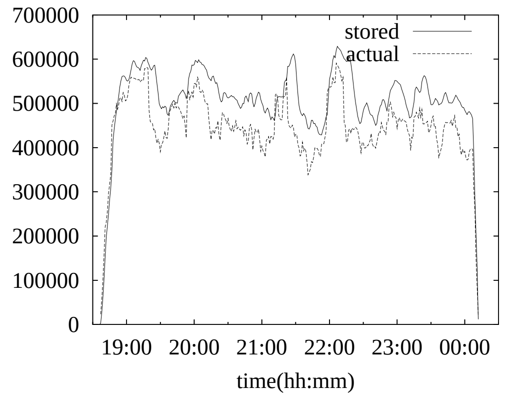
Figure 8: Experimental results

5.2 Query and Publish Load

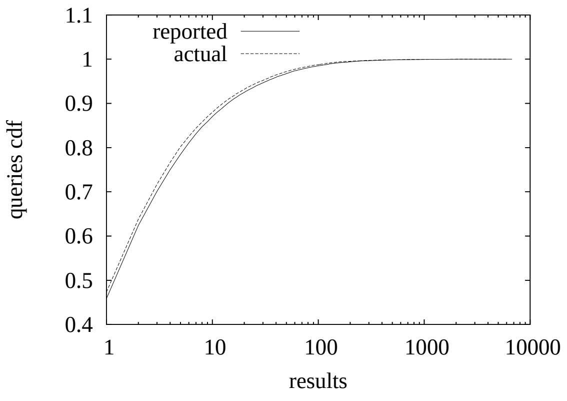
In this section we examine the amount of query and publishing load exerted on the system. Figure 7(c) shows the distribution of the average number of `get` requests received per minute by each pointer node. A `get` is issued by a client as the first step of each query to discover the storage nodes that maintain client entries for the queried keyword. The average number of queries per minute observed by each server is about two as we have about twice as many clients as there are servers issuing queries at a steady rate of one per minute each. The max/average ratio is steady at about 5.

Figure 7(d) shows the distribution of query requests per storage node. The average number of queries per storage node is a little higher than the average number of `get` requests per pointer node. This is due to the fact that some keyword entries are split between several nodes. A querying client in this experiment queries all the applicable storage nodes. For every query, there is exactly one request issued in the dht, but might be more than one issued to storage nodes. One trend to observe is there seems to be a hotspot in the beginning of the experiment but we see that it gets relieved as the experiment continues. The hotspot is due to the temporarily uneven distribution of keywords per storage node as was observed in Figure 7(b), but it gets resolved as a side effect of dynamic load balancing. Once the hot spot has been eliminated the distribution shows a low max/average ratio.

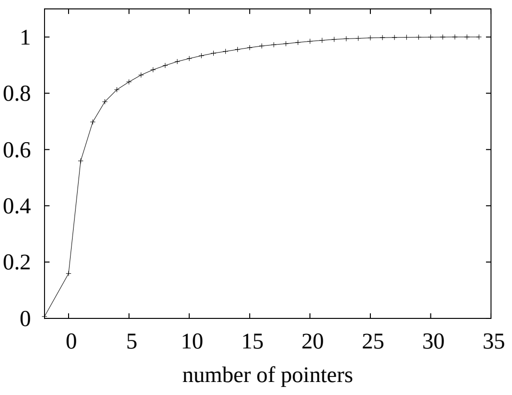
The poorest resulting load distribution is the one due to the client publishing process. Figure 8(a) shows the distribution of lookup requests per pointer node, which get issued when a client is deciding to which storage node it should publish entries for a particular keyword. As some keywords appear in most clients' filenames, the pointer nodes responsible for those keywords get hit the most. This can only be controlled by the clients as they choose which keyword entries to publish, and which lookups to perform. It can however, be improved by using a different client-side policy requiring clients not to publish any entries for the top ranking keywords as was shown in Section 4. The max/average ratio for this distribution is the highest of all, peaking at around 35. On the other hand, the load generated by the storage nodes' pointer insert protocol is very well balanced as shown in Figure 8(b).



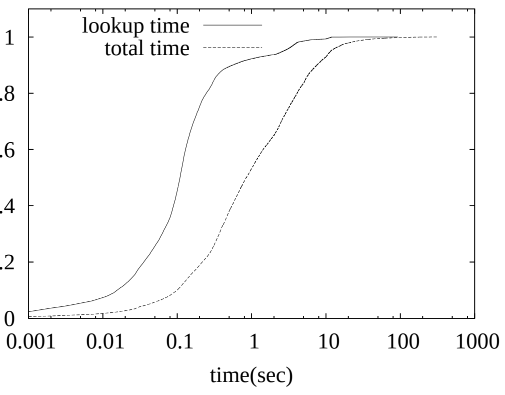
(a) Number of client entries: actual vs. stored



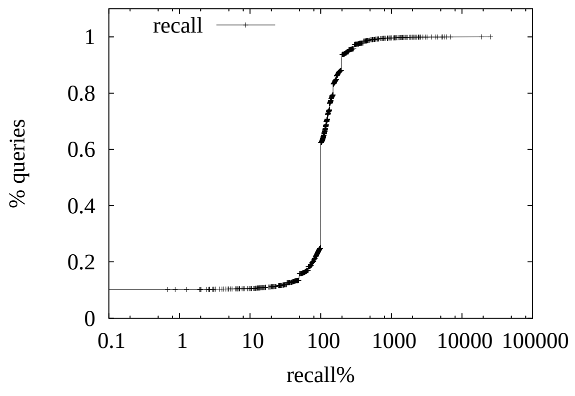
(b) CDF of results returned per query



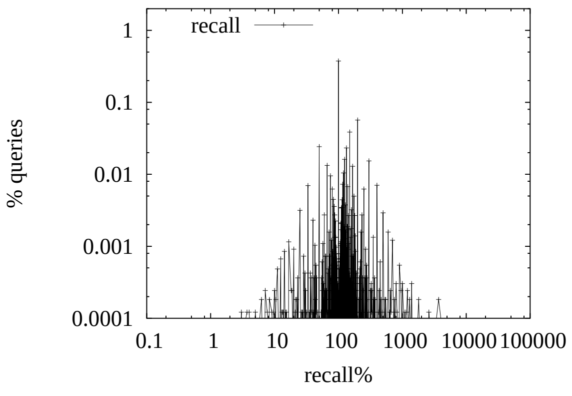
(c) Number of storage node pointers per lookup



(d) Query response time



(e) Query recall CDF



(f) Query recall PDF

Figure 9: Experimental results

5.3 Query Quality and Recall

Figure 9(a) shows the true size of the index for currently running clients against the size of the index stored in our system. We can see that the number of entries stored follows the trends in the ”actual” curve but lags behind by about ten minutes. The reason behind this disparity is that as clients go down, their entries don't expire in the storage nodes until the entry ttl expires. The client entry ttl in this experiment is set to 10 minutes. This lag causes a bit of a problem for query quality as queries are resolved against some stale data and the results produced contain false positives. Figure 9(b) is the cdf of the number of results per query, both the results reported by our system and the centralized distribution server. About 45% of all queries reported no results. The queries used were the ones that had an answer when evaluated against our whole dataset of 603 clients. In the experiment, only a fraction of those clients are up at any one point, and the issued queries are only evaluated against this subset. We can see that the two curves are very similar with the "actual" curve a bit higher, testifying to the fact that the reported results contain some false positives.

Figures 9(c) and 9(d) mostly testify to the efficiency of our dht implementation. Figure 9(c) shows the distribution of the number of pointers to storage nodes returned to clients when issuing queries. With the `get` operation set to timeout after 5 seconds, only 0.7% percent of the requests actually timed out. Less than 16% percent of lookups returned zero pointers signifying the fact that there were no entries stored for the queried keyword. Almost 40% of all queried keywords did not have their inverted indices split and were located on a single storage node. Only 7 percent of queried keywords had their entries split on more that 10 storage hosts, with the largest split factor of 34. The effective query completion times are shown in Figure 9(d). Resolving each query is a two step process composed of issuing a `get` lookup in the dht to get the pointers and then following the pointers to the storage nodes. In this experiment, if more than 1 pointer is returned, we follow all of them, primarily to get the absolute number of query results. 92% of all pointer lookups get resolved in less than 1 second and 93% of all queries get resolved in less than 10 seconds. The longest running query took a total of five minutes to complete, contacting 22 storage nodes in the process.

Finally, we look at the effective query recall in our system. To compute query recall we looked at all the queries that had one or more results and used the ratio of results reported by our system to "actual" results (as provided by the centralized distribution host). Figure 9(e) shows the query recall cdf and Figure 9(f) the corresponding pdf. A total of 16,478 queries out of 31,284 queries issued actually had one or more matches, and we only consider these queries when computing the recall. Exactly 10% of these queries had reported no results in our system. We attribute this to the temporal lag in the client publishing process. The publishing process is not instantaneous and some queries are issued for keywords whose entries have not been published yet. A total of 15 percent of the queries have recall less than 50% and 25 percent have recall less than 100%. Exactly 40% of all queries get resolved absolutely, with 100% recall quality. Less than 7% of all queries got a false positive rate of more than 2. The rest of the queries contain more false positives due to the 10 minute lag present when clients leave the system.

6 Conclusion

In this paper, we presented a new implementation of a file-sharing system based on partitioned inverted indices. The system yields (1) query result quality better than flooding and close to a centralized index, and (2) a low-maintenance network overhead. These improvements result from our optimized approaches to handling (a) high churn rates and (b) skewed publishing and querying workloads. We addressed high churn rates by keeping all data in soft state, which is periodically refreshed, such that the loss of a server or client is quickly reflected in the indexes. Skewed workloads are load balanced with the use of a layer of indirection for placing and locating data based on the frequency of use. Through a synergy of a dht-based layer of indirection and an original soft-state maintenance protocol we have created a platform that goes well beyond the current unstructured flood-based file sharing systems.

References

[CCR04] Miguel Castro, Manuel Costa, and Antony I. T. Rowstron. Should we build gnutella on a structured overlay? *Computer Communication Review*, 34(1):131–136, 2004.

[CGM02] Arturo Crespo and Hector Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS*, pages 23–, 2002.

[CLL04] Jacky Chu, Kevin Labonte, and Brian Neil Levine. An evaluation of chord using traces of peer-to-peer file sharing. In *SIGMETRICS*, pages 432–433, 2004.

[CRB$^+$03] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *SIGCOMM*, pages 407–418, 2003.

[CS02] Edith Cohen and Scott Shenker. Replication strategies in unstructured peer-to-peer networks. In *SIGCOMM*, pages 177–190, 2002.

[DKK$^+$01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[GDS$^+$03] P. Krishna Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *SOSP*, pages 314–329, 2003.

[GFJ$^+$03] Zihui Ge, Daniel R. Figueiredo, Sharad Jaiswal, James F. Kurose, and Donald F. Towsley. Modeling peer-peer file sharing systems. In *INFOCOM*, 2003.

[GSG02] P. Krishna Gummadi, Stefan Saroiu, and Steven D. Gribble. A measurement study of napster and gnutella as examples of peer-to-peer file sharing systems. *Computer Communication Review*, 32(1):82, 2002.

[KLVW04] Alexander Klemm, Christoph Lindemann, Mary K. Vernon, and Oliver P. Waldhorst. Characterizing the query behavior in peer-to-peer file sharing systems. In *Internet Measurement Conference*, pages 55–67, 2004.

[LCC$^+$02] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS*, pages 84–95, 2002.

[LHH$^+$04] Boon Thau Loo, Joseph M. Hellerstein, Ryan Huebsch, Scott Shenker, and Ion Stoica. Enhancing p2p file-sharing with an internet-scale query processor. In *VLDB*, pages 432–443, 2004.

[LLH$^+$03] Jinyang Li, Boon Thau Loo, Joseph M. Hellerstein, M. Frans Kaashoek, David R. Karger, and Robert Morris. On the feasibility of peer-to-peer web indexing and search. In *IPTPS*, pages 207–215, 2003.

[RB02] Mema Roussopoulos and Mary Baker. Cup: Controlled update propagation in peer-to-peer networks. *CoRR*, cs.NI/0202008, 2002.

[RD] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218.

[RFH$^+$01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[RGK$^+$05] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. Opendht: A public dht service and its uses. In *Proceedings of the ACM SIGCOMM '05 Conference*, August 2005.

[RGRK04] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the 2004 USENIX Technical Conference, Boston, MA, USA*, June 2004.

[RV03] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *Middleware*, pages 21–40, 2003.

[SMK$^+$01a] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[SMK$^+$01b] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[Str] Jeremy Stribling. Planetlab all-pairs ping.

[TD04] Chunqiang Tang and Sandhya Dwarkadas. Hybrid global-local indexing for efficient peer-to-peer information retrieval. In *NSDI*, pages 211–224, 2004.

[TXD03] Chunqiang Tang, Zhichen Xu, and Sandhya Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *SIGCOMM*, pages 175–186, 2003.

[YGM02] Beverly Yang and Hector Garcia-Molina. Improving search in peer-to-peer networks. In *ICDCS*, pages 5–14, 2002.

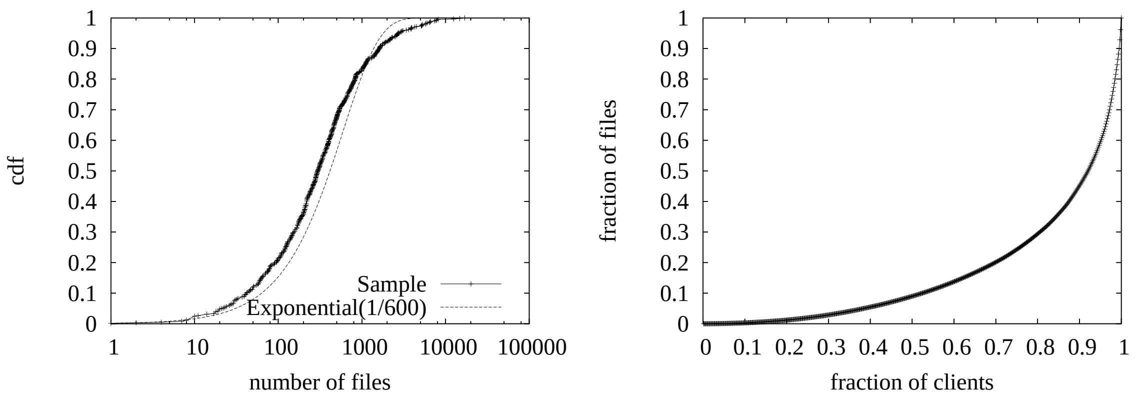
[ZH05] Rongmei Zhang and Y. Charlie Hu. Assisted peer-to-peer search with partial indexing. In *IEEE INFOCOM*, March 2005.

[ZKJ01] 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, April 2001.

Appendix

A Gnutella Trace Analysis

Gnutella clients expose a `browse host` interface through which they report the names of all the files that are residing in their shared directory and that they are offering to other Gnutella clients. We have instrumented our local Gnutella client to connect to the Gnutella network, issue queries for a hand-picked set of keywords with the intention of getting many query replies with links to numerous clients. For each unique client that we observe in the query replies, we issue the `browse host` call. In this trace, from May 2, 2005, we have issued 1090 `browse host` requests to unique Gnutella clients and have received replies containing one or more files from 603 clients. A reply was received for each file on a client and contains the unique client identifier and the full file name. We have received listings for 443,213 files containing a total of 180,683 unique keywords obtained from the file names using white space and '.' as the delimiters. We use this trace to obtain the distributions of files per host, keywords per file name and the keyword popularity in file names.

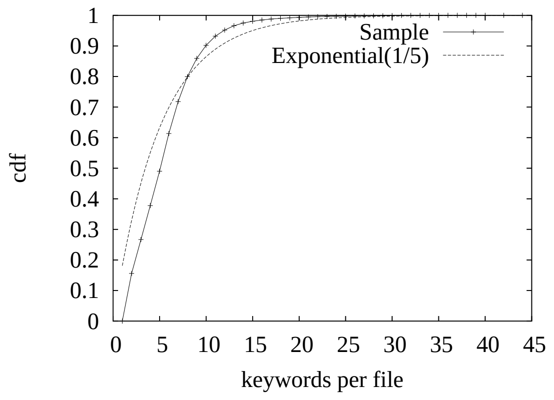


(a) files per client distribution (b) fraction of files shared by clients

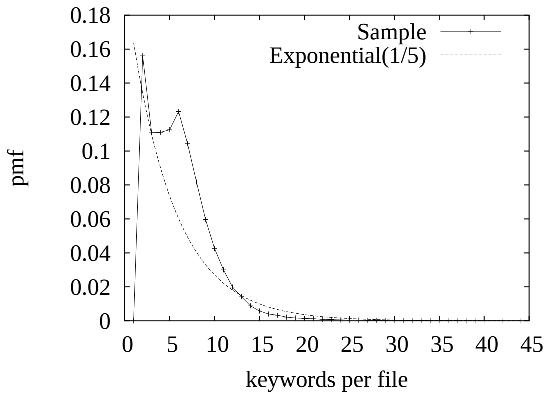
Figure 10: Distribution of files per client

In Figure 10(a) we plot the distribution of number of files per host. The sample cdf is plotted against the exponential cdf with mean 600. The median and average number of files per host is 297 and 735 respectively. The maximum number of files per host observed in this trace was 16,921. The middle 50% of all clients share between 125 and 670 files. About 13% of all clients have more than 1,000 files. To closely observe the disparity between the number of files shared by clients we ranked the clients based on the number of files they share and plot the cumulative percentage of files shared by clients in Figure 10(b). This plot shows that 50% of all available files are offered by only 8% of all clients.

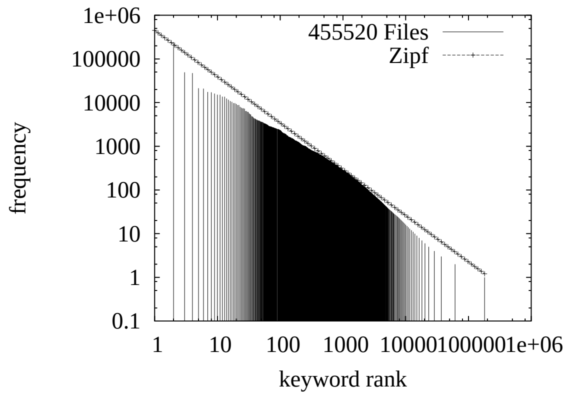
Figures 11(a) and 11(b) show the cdf and the pmf for the number of keywords observed in a file name. The median number of keywords is five and the average number of keywords is six. The maximum number of keywords per file observed in our traces was 44. The distribution peaks at two, has a somewhat flat head between 3 and 7 keywords and then drops with a long tail.



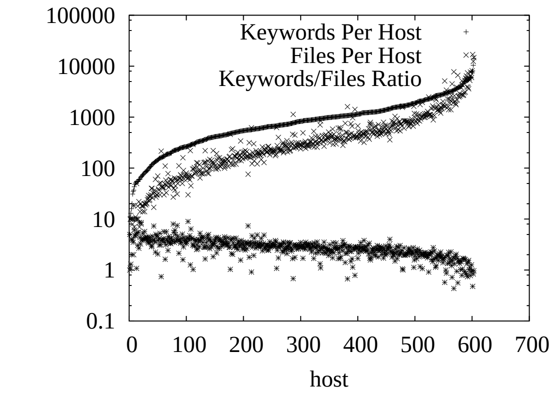
(a) Distribution of keywords per file name (cdf)



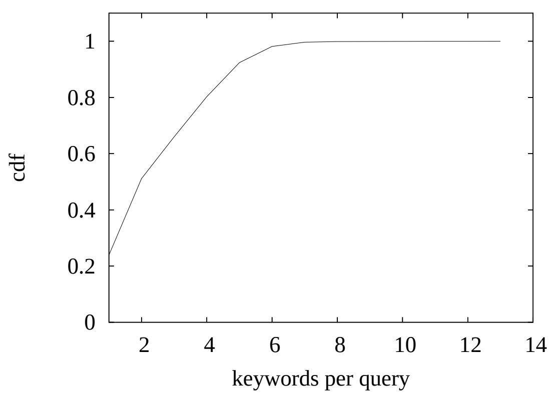
(b) Distribution of keywords per file name (pmf)



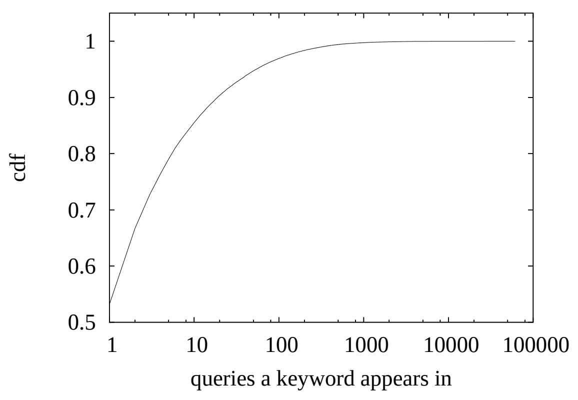
(c) Keyword frequency distribution in file names



(d) Number of files and distinct keywords per host



(e) Keywords per query

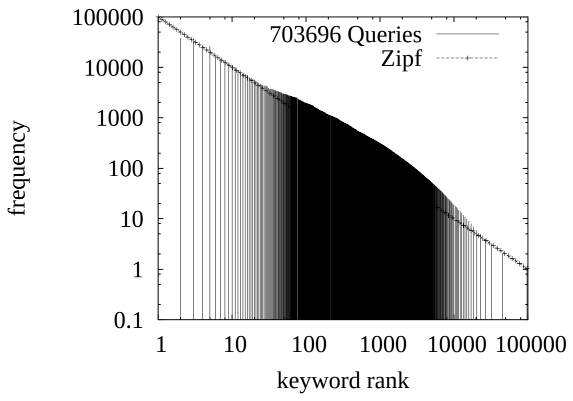


(f) Queries per keyword

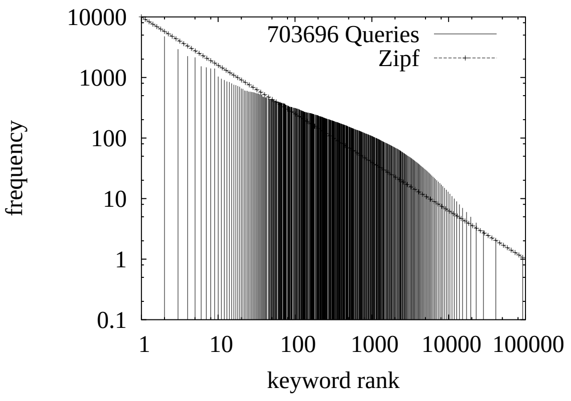
Figure 11: Trace analysis

Rank	1	2	3	4	5	6	7	8	9	10
Keyword	-	mp3	jpg	the	gif	exe	wmv	you	of	&
Frequency	257451	241169	49415	47364	21408	20899	17514	17235	16084	15157

Table 5: Top 10 keywords and frequencies



(a) keyword frequency in queries



(b) keyword frequency in queries using the most selective keyword

Figure 12: Keyword frequency in queries

Figure 11(c) plots the distribution of keyword frequency among all file names obtained against the Zipf distribution. Once again, 180,683 unique keywords we observed and their popularity in file names drops roughly according to Zipf. Table 5 lists the top 10 most popular keywords. Due to the choice of our keyword delimiter, '-' appears to be most popular. The rest of the keywords in the top ten are file extensions and common English words.

In Figure 11(d) we ranked the hosts based on the number of distinct keywords their local files contained. The median and average number of keywords per client were 831 and 1201 and the maximum is 14,312 keywords. For each point we also plot the number of files residing on that client. The ratio of number of keywords to number of files per host is also plotted and the average ratio is about 2.8.

From the traces gathered in [LHH$^+$04] we are able to obtain the following data about the distribution of keywords in queries. More than 700,000 queries were observed containing more than 93,000 unique keywords. Figure 11(e) displays the distribution of the number of keywords per query. The average number of keywords is 2.9 and the median is 2. Less than 2% of all observed queries had more than 6 keywords in them. Figure 11(f) shows the distribution of the frequency of keywords in queries. More than 50% of all observed keywords were seen in exactly one query, and more than 85% of the keywords were seen less than 10 queries. Figure 12(a) plots the keyword frequency in queries against the Zipf distribution, and Figure 12(b) plots the same distribution using only the most selective keyword from each query.

B Protocol Overhead

In this section we go through the steps involved in the normal operations of the pointer, storage and client protocols. We are expecting churn to be part of normal operations and it is accounted for in the protocols and their overhead. We calculate the overhead of each protocol by calculating resulting per server network overhead in terms of bytes per second. Total traffic is a good estimate because the real processing in our system comes from sending and receiving messages. The servers (pointer/storage nodes) don't really do

Parameter	**Description**	**Sample A**	**Sample B**
C	Clients	130	1,000,000
S	Servers	76	100,000
P_I	Pointer Period (seconds)	2	2
P_S	Store Period (seconds)	120	120
P_C	Client Period (seconds)	240	240
P_Q	Query Period (seconds)	60	60
L	Average Message Length (Bytes)	100	100
L_Q	Query Response Length (Bytes)	500	500
r	Replication Factor	4	8
K	Total Keywords	50,000	10,000,000
K_C	Keywords / Client	1000	1000
F_C	Files / Client	600	600
K_F	Keys / File	5	5
M	Metadata Size (Bytes)	50	50
U_C	Client Uptime (seconds)	2400	3600
U_S	Server Uptime (seconds)	86400	86400

Table 6: Overhead parameter summary

anything else. We feel that this is indicative of the amount of work that each entity needs to perform and reflects on the overall load injected into the network. Table 6 lists all the relevant system parameters, and provides two sample workloads, one which we used to evaluate the system on PlanetLab (see Section 5) and one typical of real large scale setups such as the Gnutella network.

Parameters C and S represent the number of clients and servers in the system. Servers are the nodes running the pointer and storage protocols. Everything in the system is a timed event with a period expressed in seconds. Pointer nodes engage in a stabilization protocol to maintain the integrity of their routing tables every P_I seconds. Each storage node issues a `put` request to update the pointer to itself at a rate of P_S seconds for every keyword it stores. Clients engage in update protocol with the storage nodes at a period of P_C. We model the event of queries being issued by clients as a periodic, per-client process with one query being issued every P_Q seconds. These periods, except for query rate, are configurable system parameters. The pointer replication factor r is also a system wide parameter.

To keep the model cleaner, we use L as the average length of all the control and lookup messages. The actual variance in the length of different control messages is small, and each message is sent over UDP. The typical length of the query results is represented by L_R. K represents the number of distinct keywords that appear in all of the clients' shared files. This is a non-decreasing function of C, the number of clients but we do not have a closed form solution for it. In this analysis we will use reasonable estimates or observed values. We use the ratio K/S as an estimate for number of keyword per server. This assumes that we can achieve perfect load balancing and that each storage node will be responsible for an equal portion of the key space. In Section 4 we discuss techniques to achieve balanced storage load distributions and confirm with experiments in Section 5.

The following four parameters, K_C, F_C, K_F, and M are properties of the client shared workload and are derived from our analysis of Gnutella client traces in Appendix A). We observe the average values for the number of files per client, distinct keywords per client, keywords per file and average size of the file metadata. The final parameters that affect the model are the average client and server uptimes. We assume that the number of clients and servers up at any time is approximately constant, an assumption backed by the observation of typical Gnutella usage. Not surprisingly, churn is the direct byproduct of client and server uptimes.

B.1 Pointer Node Overhead

Each pointer node performs three actions every P_I seconds to keep its routing tables up-to-date. First, it successively pings one node from it's routing table to make sure that it's still alive. Second, it sends a ping to each of it's r immediate neighbors to make sure that its neighbor set is always up to date. Finally, each node updates its i^{th} routing table entry by finding the current node 2^i away from it in the ring structure. To perform a ping, the node sends a ping message, recipient receives it, sends a reply, and the node receives reply. A total of $4*L$ bytes are sent and received by the servers. As S servers send pings every P_I seconds, the resulting traffic is $4*L*S$ bytes total, $4*L*S/P_I$ bytes per second and finally $4*L/P_I*S/S$ bytes per second per server. To keep its neighbor set up-to-date, each node sends a message to all of its r neighbors. Each neighbor replies with its current neighbor set. Two messages are sent and two are received for a total of $4*r*L$ bytes. The total traffic overhead for the neighbor protocol is $(4*r*L)/P_I$ bytes per second per server. To fix a routing table entry, one lookup is made for the current successor. The average number of messages sent is $log(S)$ for a cost of $(2*log(S)*L)/P_I$ bytes per second per server. The total cost of the pointer protocol is

$$((4+4*r+2*log(S))*L)/P_I$$

bytes per second per server.

B.2 Storage Node Overhead

Storage nodes need to periodically update their pointers. For every keyword at a storage node, a pointer is inserted into the dht every P_S seconds. We assume that a storage node is responsible for an average of K/S keywords, and thus needs to perform $K/S/P_S$ *put* operations per second. Each *put* is routed through $log(S)$ nodes and at the destination gets forwarded to $r-1$ neighbor nodes (for replication) with an expected acknowledgement. Thus, for each *put* $(2*log(S)+4*(r-1))*L$ bytes are sent and received. $(2*log(S)+4*(r-1))*L*(K/S/P_S)*S$ bytes per second are sent by all S servers, which amounts to the overhead of

$$(2*log(S)+4*(r-1))*L/P_S)*(K/S)$$

bytes per second per server.

B.3 Publishing Overhead

During its uptime of U_C seconds, each client needs to publish its entries into the system. The publishing process can be viewed as composed of two stages, the lookup stage and the transfer stage. For each keyword a client first performs a lookup in the dht to find an appropriate storage node and then sends the portion of the entries to the chosen storage node. If a client has F_C files and each file name contains an average of K_F keywords the number of entries in the client's inverted index will be $F_C * K_F$, as there will be K_F entries for each file. The size of the inverted index will be $F_C * K_F * M$ bytes, where M is the average length of a file's metadata (the length of the file name). Each client will publish it's index once in it's lifetime, of U_T seconds. We account for storage node churn, and the need to re-publish lost entries in the client overhead, in the next section. Thus, each client will make the storage nodes receive an average of $(F_C*K_F*M)/U_C$ bytes per second as part of the transfer process. With an average of C clients publishing their entries every U_C seconds, with the publishing load split among S storage nodes, the transfer cost is $(F_C * K_F * M)/U_C * (C/S)$ bytes per second per server. With a total of K_C distinct keywords per client, each client needs to perform K_C *get* lookups in the dht during it's lookup stage. Each lookup exerts a load of $(2 * log(S) * L)$ on the pointer nodes, for a total load of $(2 * log(S) * L) * K_C/U_C * (C/S)$ bytes per second per server. The total resulting publishing overhead is

$$((2 * log(S) * L * K_C) + (F_C * K_F * M))/U_C * (C/S)$$

bytes per second per server.

B.4 Client Update Overhead

The client's index entries are set to expire in the storage node unless they are updated by the client. The client periodically sends a small refresh message directly to the storage nodes letting them know that its entries should not be discarded. Once the client is past the publishing stage and knows the address of each storage node maintaining its entries, it can communicate with the storage node directly to refresh its entries. Through this mechanism the client also makes sure that the storage node is still alive and is storing the client's entries. If not, the client needs to find a new storage node and republish the missing entries.

If the number of storage nodes S is significantly smaller than the number of keywords on a client K_C, then it makes sense for the client to send updates on per storage node basis. If however, the number of storage nodes is much larger than the number of keywords then the client updates its entries per keyword. The number of updates that each client needs to send every P_C seconds is then $min(K_C, S)$. The refresh cost is that of sending and receiving a refresh message of size L to and from $min(K_C, S)$ storage nodes every P_C seconds. There are $2 * L * min(K_C, S)/P_C$ bytes per second due to every client. The resulting refresh load is $(2 * L * min(K_C, S))/P_C * (C/S)$ bytes per second per server.

The cost of churn can be estimated using the average storage node uptime U_S and the client uptime U_C. For each of the K_C keywords on a client, the probability that the corresponding storage node wend down in the last P_C seconds is $Pr[down] = P_C/U_S$. Republishing entries for one keyword requires a lookup and a corresponding transfer, for a total of $(P_C/U_S) * (2 * log(S) * L + (F_C * K_F * M)/K_C)$ bytes. Considering all the K_C keywords, each client sends $K_C * (P_C/U_S) * (2 * log(S) * L + (F_C * K_F * M)/K_C)/P_C$ bytes per second to account for storage node churn. For a total of

$$K_C * (P_C/U_S) * (2 * log(S) * L + (F_C * K_F * M)/K_C)/P_C * (C/S)$$

$$= (P_C/U_S) * (2 * log(S) * L * K_C + (F_C * K_F * M))/P_C * (C/S)$$

$$= (U_C/U_S) * (2 * log(S) * L * K_C + (F_C * K_F * M))/U_C * (C/S)$$

$$= (U_C/U_S) * PUBLISH$$

Thus, the total client overhead, including churn and refresh is

$$(U_C/U_S) * PUBLISH + 2 * L * min(K_C, S)/P_C * (C/S)$$

bytes per second for each server.

B.5 Query Overhead

Querying involves performing a lookup to find the correct storage node, sending the query to the storage node and receiving the results. The query and result length are specified by parameters L and L_R bytes. A single client query issued once every P_Q seconds requires work of $(2 * log(S) * L + L + L_R)/P_Q$ bytes per second. With C clients issuing queries at this rate, the resulting load is

$$(2 * log(S) * L + L + L_R)/P_Q * (C/S)$$

bytes per second per server.

C Index Layer

We build our index layer on top of a distributed hash table. In general, a DHT exposes a thin interface for the typical hash table *put* and *get* operations. For every object identifier id_o, a unique node exists in the DHT that is responsible for id_o. In some DHTs it's the node whose id immediately follows the object's identifier in a geometric structure like the ring [SMK$^+$01a], while in others it's the node whose id is numerically closest to the object's identifier [RGRK04]. Nodes in a DHT experience varying network latencies and application response times due to physical network proximity and machine load. When a source node initiates a lookup for a particular identifier, the response time it experiences greatly depends on the response times of the individual nodes that are involved in resolving the lookup. A lookup in a DHT with n nodes is expected to contact $O(log(n))$ nodes and if any one of them is slow to respond, the entire lookup will be prolonged. Intermediate node selection in a DHT lookup is crucial to the lookup's overall performance. Here we explore DHT routing algorithms and their performance with respect to the overall

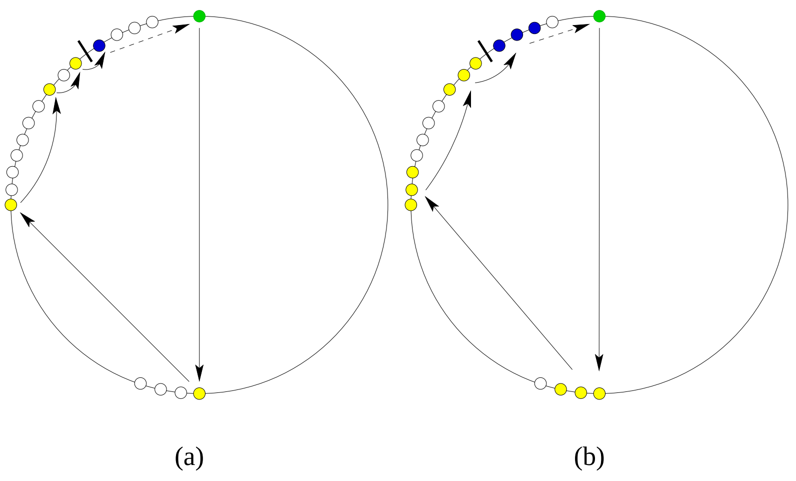


Figure 13: Index Routing. (a) Deterministic routing for a lookup where the top node is the source of the lookup, the tick mark represents the id being looked up and dark node is the successor of the id. (b) Resolving a lookup with proximity successor selection with replication factor of 3.

lookup latency. An evaluation of our implementation of the index layer is presented in Appendix C.4.

C.1 Deterministic Routing

We will use Chord as the base case for deterministic routing [SMK+01a]. In Chord all nodes are organized in a ring and have 160-bit identifiers. A **successor** of an identifier is defined to be the node whose id *immediately* follows the identifier in the ring. A **predecessor** of an identifier is *any* node whose id precedes the given identifier in the ring space. An immediate predecessor of an identifier, or a node, is defined to be the closest predecessor. A node n with identifier id_n makes sure that it always knows its r immediate predecessors and r immediate successors through a neighbor upate protocol. It also maintains a routing table where for every $i \in \{0 \dots 159\}$ it stores the address of the successor of $(id_n + 2^i) mod 2^{160}$. For node n to complete a *get* request for id_o, n routes the request to the closest predecessor p_1 of id_o that n knows about. If p_1 cannot determine the successor of id_o locally, then p_1 recursively proceeds with the request by finding the closest predecessor of id_o in its routing table and forwards the request there. When the request reaches its destination, the actual successor of id_o, and it's guaranteed to do so in $O(logn)$ steps, the successor sends a reply back to n, the source of the request.

Figure 13(a) pictorially depicts this process. The top node issues a get request for an object whose identifier is represented by the tick mark on the ring. The node responsible for storing the object for that identifier is the node whose id directly follows the object identifier in the ring, represented by the dark node. The source node forwards the message to the routing entry that is closest to the object id, which is half way around the ring. The farthest that the next hop node can forward the message without overshooting the destination is a quarter way around the ring. The message proceeds taking smaller and smaller hops around the ring until it reaches the destination node's predecessor, and the predecessor fully aware of its neighborhood can safely forward the message to its destination. The destination node then resolves the get request and sends the reply directly to the source of the message. The problem with this routing scheme is that given a concrete network, a source node n and lookup identifier id_o, the choice of nodes that will be contacted as the lookup proceeds is predetermined based on the identifiers. The request can be delayed at any one of the nodes and not much can be done to expedite it.

C.2 Proximity Neighbor Selection

Proximity neighbor selection, also called proximity route selection, addresses the problem of limited next-hop choices by considering more nodes for the routing table. As described above, routing table entry i of node n keeps the address of the node 2^i away from n in the ring. With PNS, instead of storing a unique node for every routing table entry, entry i considers up to r nodes in the range 2^i to $2^{i+1} - 1$ and uses the entry with the smallest observed delay. Delays are observed with through periodic pings as described earlier. For large i, many alternate nodes can be considered for the routing table, but as i decreases fewer choices become available. As a message gets routed closer to its destination in the network, the number of choices for the next hop depends on the routing table entries that can be used.

We now consider the choices that are available in the routing scenario of Figure 13(a). The destination node needs to send a direct reply to the source of the lookup message so there is no choice present there. The last hop in the lookup can only go to the destination of the message and there is only one choice present there as well. Working backwards, the nodes on the previous hops start having more freedom in choosing who to forward the message to. The second to last hop has a choice of two, the one before it a choice of four and so forth. We can approximate average lookup latency in a DHT implementing PNS assuming an exponential per-hop latency distribution with median delay δ. The average lookup latency is approximated as $\delta + (\delta + \delta/2 + \delta/4 + \ldots) = 3\delta = 1.5$RTT where the first δ is for the reply to the source of the request, the second δ is for the message to the successor, and the rest of the series is for the best hop among a choice of routing entries.

C.3 Proximity Successor Selection

Even though proximity neighbor selection presents more choices for most of the hops taken when routing a message, there is still only one choice for the last hop to the destination and for the hop back to the source. Although we cannot change the source of the request, the last hop to the single destination presents us with a bottleneck to be dealt with. To deal with it, we make sure that objects are stored not only on a single node but on r nodes, which we make a system-wide parameter. Now, the original destination node of an object's identifier plus its $r - 1$ immediate successors in the ring are responsible for maintaining the object. A get request can be forwarded to any of the r nodes and be resolved locally. A put request can be forwarded to any of the r nodes and then propagated to the other $r - 1$ nodes. This replication assumption is a common one and has been used in existing systems [RGK$^+$05, DKK$^+$01], and has been taken advantage of by Pastry's [RD] routing algorithms. Figure 13(b) shows the possible hops taken while resolving a lookup using proximity successor selection. To mimic the calculations of the previous section, the average lookup should take $\delta + (\delta/r + \delta/r + \delta/r + \ldots)$ where r is the system-wide parameter for the number of immediate successors. In the case where r is configured to be $logn$, the average lookup delay will be equal to $2\delta = 1$RTT.

C.4 Index Performance

C.4.1 Analysis

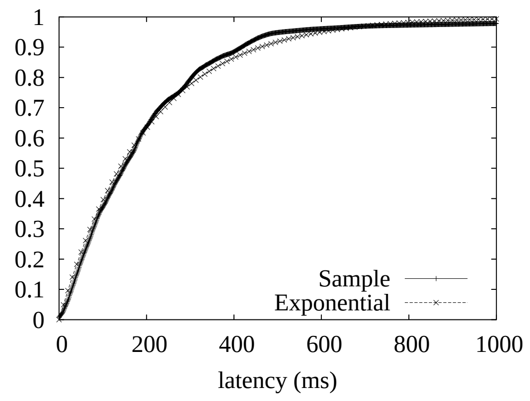
Here, we analyze how the additional next hop choices offered on selected hops affect the expected delay of a lookup message. Consider the one-way delay between two nodes on the network as a random variable, and let the expected value of the one-way delay be δ. When a node considers k possible next hops and chooses the next hop based on the smallest delay, the expected delay now becomes the expected value of the minimum of the k random variables. If the k random variables are independent and identically distributed, exponentially with mean λ, the expected value of the minimum of these variables is λ/k and is also exponentially distributed. If we let δ be the mean delay of a one-way hop, then $\lambda = 1/\delta$. Now, let λ_k be mean of the random variable $X = min(X_1, \ldots, X_k)$, which is the minimum of k choices, where X_i is the delay to the i^{th} choice, and let δ_k be the mean delay when k choices are present. Then $\lambda_k = k\lambda = 1/\delta_k$ and $\delta_k = \delta/k$. Thus, when a node can consider k nodes for its next hop, the expected delay of the next hop is $1/k^{th}$ of the average one-way delay δ.

For deterministic routing if δ is the expected value of the one way delay between two nodes and the lookup message makes $logn$ node-to-node hops, the expected value for the delay of the lookup is $\delta * logn$. To calculate the approximate average lookup latency in a DHT implementing PNS we need to once again consider the number of next hop choices present at every node. The average lookup latency is approximated as $\delta + (\delta + \delta/2 + \delta/4 + \ldots) = 3\delta = 1.5$RTT where the first δ is for the reply to the source of the request, the second δ is for the message to the destination, and the rest of the series is for the best hop among a choice of neighbors. To mimic the previous calculations, the average lookup with PSS should take $\delta + (\delta/r + \delta/r + \delta/r + \ldots)$ where r is the system-wide replication parameter. There is still only one choice for the message reply back to the source of the lookup but all the other hops can safely choose between r nodes. If r is configured to be $logn$, as recommended [SMK+01a], then the average lookup delay will be equal to $2\delta = 1$RTT. Even if r is set to a smaller value, as long as that value is greater than one, significant improvement in the observed lookup latency can be expected.

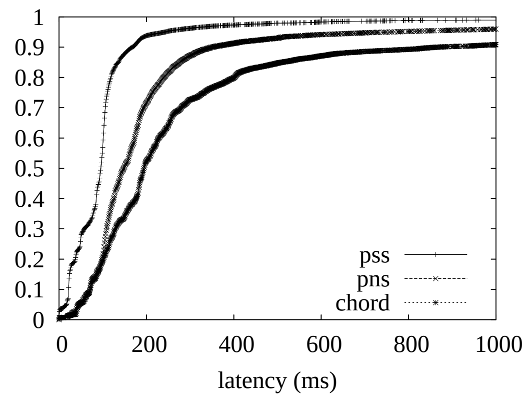
Figure 14(a) shows the distribution of one way delays between nodes on PlanetLab measured with ping [Str], plotted against the exponential distribution with mean λ =200ms. They are more skewed than the exponential distribution and therefore the expected value of the minimum of k delays is at least as good as the expected value when the exponential is used.

C.4.2 Evaluation

The index layer implementation contains support for all of the three routing algorithms which allows for an easy evaluation. In this evaluation, the implementation has been deployed on about 200 PlanetLab nodes. Each index node runs a proxy which exposes the put and get interfaces to machines which are not part of the index. In the overall system this proxy is used by both client and storage nodes. (In the following discussion when we refer to proxy clients, we are including both client and storage nodes.) In the current



(a) CDF of one-way latency between nodes on PlanetLab.



(b) Experiments on PlanetLab with 200 nodes, $r = 4$ and 10,000 requests for random identifiers through one proxy

Figure 14: Lookup performance

implementation, when a proxy receives a request from a client it routes the request on behalf of the client and maintains its connection to the client until it receives a response or it times out. The proxy issues the request every two seconds until a response is received and propagated to the client or the timeout expires and an error is returned. The timeout is specified by the client.

	Chord	PNS	PSS
50%	0.194s	0.147s	0.095s
90%	0.865s	0.350s	0.167s
95%	2.652s	0.756s	0.242s
99%	$>$ 10s	3.846s	1.115s
99.9%	$>$ 10s	$>$ 10s	3.963s
99.99%	$>$ 10s	$>$ 10s	5.054s
100%	$>$ 10s	$>$ 10s	5.858s

Table 7: PlanetLab latency summary

In this experiment we first find a proxy node which is not heavily loaded and is close to our client. We then randomly choose 10,000 identifiers and for each one issue three get requests, one for each routing algorithm. To give a fair evaluation, we have to make sure that we evaluate the three algorithms with the same source destination pairs. We can explicitly specify to the proxy which algorithm we want to be used for routing the messages. The network runs with the system-wide parameter r set to four and each request to the proxy is set to timeout after 10 seconds without a response. Figure 14(b) shows the cumulative distribution function for get request latency and Table 7 summarizes the results of our experiment. Not only is the average behavior of PSS much better than the other two routing algorithms, but the long tail behavior is also eliminated. While 1% of all deterministic routing requests and 0.1% of all requests using PNS require more than 10 seconds to complete, all requests routed using PSS complete in less than 6 seconds.