***Arpeggio*: Metadata Indexing in a Structured Peer-to-Peer Network**

by

Dan R. K. Ports

S. B., Computer Science and Engineering
Massachusetts Institute of Technology, 2005

S. B., Mathematics
Massachusetts Institute of Technology, 2005

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2007

Author . . . .
Department of Electrical Engineering and Computer Science
February 2, 2007

Certified by . . . .
David R. Karger
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

***Arpeggio*: Metadata Indexing in a Structured Peer-to-Peer Network**

by

Dan R. K. Ports

Submitted to the Department of Electrical Engineering and Computer Science on February 2, 2007, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Peer-to-peer networks require an efficient means for performing searches for files by metadata keywords. Unfortunately, current methods usually sacrifice either scalability or recall. *Arpeggio* is a peer-to-peer file-sharing network that uses the Chord lookup primitive as a basis for constructing a *distributed keyword-set index*, augmented with index-side filtering, to address this problem. We introduce *index gateways*, a technique for minimizing index maintenance overhead. Arpeggio also includes a content distribution system for finding source peers for a file; we present a novel system that uses Chord subrings to track live source peers without the cost of inserting the data itself into the network, and supports *postfetching*: using information in the index to improve the availability of rare files. The result is a system that provides efficient query operations with the scalability and reliability advantages of full decentralization. We use analysis and simulation results to show that our indexing system has reasonable storage and bandwidth costs, and improves load distribution.

Thesis Supervisor: David R. Karger
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I gratefully acknowledge the contributions of the colleagues who have proved invaluable in conducting this research. I am especially indebted to my advisor, Prof. David Karger, who provided essential guidance for this project since I began working with him as an undergraduate. I also owe thanks to Prof. Frans Kaashoek and the PDOS group for their help and support; I would not have made much progress without aid from the Chord team. My colleague Austin Clements helped make many of the key design decisions behind Arpeggio, and was always willing to lend an ear as I proceeded through implementation and evaluation. Finally, Irene Zhang kept me sane throughout the whole process, a difficult task even at the best of times.

This research was conducted as part of the IRIS project (`http://project-iris.net/`), supported by the National Science Foundation under Cooperative Agreement No. ANI0225660, and was additionally supported by the Office of Naval Research under a National Defense Science and Engineering Graduate (NDSEG) Fellowship.

Portions of this thesis are adapted from

> Austin T. Clements, Dan R. K. Ports, and David R. Karger, *Arpeggio: Metadata searching and content sharing with Chord*, Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS '05) (Ithaca, NY, USA), Lecture Notes in Computer Science, vol. 3640, Springer, February 2005, pp. 58–68.

Contents

List of Figures

List of Tables

Chapter 1

Overview

1.1 Introduction

Peer-to-peer file sharing systems, in which users locate and obtain files shared by other users rather than downloading from a central server, have become immensely popular. Indeed, in recent years, peer-to-peer file sharing traffic has amounted to over 60% of aggregate internet traffic [8]. Such systems are useful, as well as interesting to researchers, because they operate at extremely large scale, without a central infrastructure, and can tolerate many kinds of failures.

Users of peer-to-peer file sharing systems use a client that upon startup connects to other peers in a network, and becomes a part of their distributed system and begins sharing files that the user has made available. Users perform *searches* for files, usually specifying part of a file name or other criteria, and the peers in the system work to cooperatively identify matching files that other users are sharing. The user can then request to download the file, and the client will retrieve it, often obtaining pieces from multiple users who are sharing the same file. We refer to these two tasks as the problems of *searching* and *content distribution*.

Performing searching in a peer-to-peer network is a challenging problem. Many current file sharing networks are implemented using techniques such as centralized indexes, which lack the scalability and resilience of a distributed system, or query flooding, which is notoriously inefficient. Often, they trade-off scalability for correctness, resulting in either systems that scale well but sacrifice completeness of search results or vice-versa. In this thesis, we present *Arpeggio*, a novel file sharing system based on the Chord distributed hash table [64]. Arpeggio includes a system for building a distributed index of file metadata that is fully decentralized but able to efficiently answer search queries with near-perfect recall. In addition, Arpeggio also includes a separate content distribution subsystem for identifying peers that have a particular file available.

1.2 Goals

The principal problem that Arpeggio addresses is that of searching for files by metadata. For each file, a set of *metadata* is associated with it. The metadata is represented as a set of key-value pairs containing tagged descriptive information about the file, such the file name, its size and format, etc. Ideally, the file would also have more descriptive metadata categorizing its content: an academic paper could be tagged with its title, authors, venue,

(debian, disk1, iso)	
name:	Debian Disk1.iso
file hash:	cdb79ca3db1f39b1940ed5...
size:	586MB
type:	application/x-iso9660-image
⋮	

Figure 1-1: *Example file metadata*

publisher, etc; a song could be tagged with its artist, album name, song title, genre, length, etc. For some types of data, such as text documents, metadata can be extracted manually or algorithmically. Some types of files have metadata built-in; for example, MP3 music files have ID3 tags listing the song title, artist, etc. We refer to the file's metadata as a *metadata block*. An example is shown in Figure 1-1.

Upon joining the system, peers register the files that they have available, sending "Insert" requests with their files' metadata blocks. They also perform queries, which specify a list of keywords; the system should be able to respond to a Query request with a list of file metadata blocks that contain all of the requested keywords.

Several of the key design goals of Arpeggio are listed below:

Full decentralization. The system must be fully decentralized; it must be able to withstand the failure of any individual node.

Perfect recall. If a file is being shared in the system, it should always be returned in response to a search request for its metadata. Many systems, such as Gnutella, introduce scalability optimizations that sacrifice this property. They focus primarily on finding results for popular files, for which many copies exist, and may fail to locate rare files even if a small number of peers has them available.

Scalability and load balancing. Because peer-to-peer systems are designed at a large scale — the largest networks have millions of users — it is necessary for the algorithms used to handle these numbers of users. In particular, because the data involved (the lists of files available in the network) are large, they must be stored efficiently. However, some of our techniques trade off scalability for improved load balancing properties. For example, we are willing to suffer an increase in overall index size in order to ensure that the load imbalance is decreased, so that there does not exist a small number of peers doing most of the work and likely becoming overloaded as a result.

1.2.1 Non-goals

There are several related problems that we explicitly do not address because they are outside the scope of Arpeggio; they are separate problem that we are not attempting to solve, or have been solved elsewhere and can be integrated with Arpeggio. Here, we list a few of these and provide the rationale.

Full-text document search. Arpeggio's search techniques are intended for indexing the metadata of files, and rely on the assumption that the searchable metadata content of each item is small. As a result, they do not scale to support full-text indexing of documents; Arpeggio would not be practical for indexing the web, for example.

File transfer. File sharing clients often include elaborate mechanisms for downloading files as quickly as possible, selecting multiple sources for files, and deciding which peers to download from and upload to. We only address file transfer to the extent of identifying peers that share a particular file; a system such as BitTorrent [14] can be used to actually download the file.

Heterogeneity. Nodes in peer-to-peer networks vary greatly in their bandwidth, storage capacity, and stability. Many peer-to-peer networks exploit this, allowing the more capable nodes (such as those on high-speed connections) to bear more load proportional to their resources, and allowing nodes with low resources (such as those on modem links) to act only as clients, not participating in the operation of the network. We do not discuss this in our design, but do assume that some mechanism is in place. Within a distributed hash table, this can be achieved by using the standard technique of having nodes operate multiple virtual nodes, with the number proportional to their capacity [9].

Bootstrapping. In order to join a peer-to-peer network, a new node must be able to contact one of the existing nodes in the network. Discovering such a node presents a bootstrapping problem if it is to be done without centralized infrastructure. In Gnutella, for example, this is achieved with a system of many web caches that track which nodes are available [30]. We assume that a new user is somehow provided with the address of a well-known node from which they can bootstrap their entry into the network.

Security. There are many security issues inherent in a peer-to-peer system [63]. In general, it is difficult to make any guarantees about security in a network that depends on cooperation from many untrusted nodes, particularly when one attacker may operate many nodes in the network [19]. We do not address security concerns in this thesis.

1.3 Outline

The remainder of this thesis proceeds as follows. Chapter 2 provides necessary background on indexing techniques and distributed hash tables. Chapter 3 discusses techniques for building indexes for efficient searching in distributed systems. We discuss various approaches to indexing, incrementally building up Arpeggio's algorithm and comparing it to related work. Chapter 4 turns to the problem of content distribution; we present two content distribution subsystems that can be used in conjunction with our indexing system, one based on BitTorrent and one designed from scratch. Chapter 5 presents our implementation of the system, with commentary on its high-level architecture, and discusses its user interface. Chapter 6 evaluates the feasibility and effectiveness of the system in the contexts of various sample applications, and Chapter 7 concludes.

Chapter 2

Background and Related Work

Many peer-to-peer file sharing systems currently exist. Like Arpeggio, these systems typically comprise both an *indexing* subsystem, which allows users to search for files matching some query, and a *content distribution* subsystem, which allows them to download the file contents from peers. We begin with an overview of the indexing techniques used by common peer-to-peer systems.

2.1 Indexing Techniques

2.1.1 Centralized Indexes

The earliest peer-to-peer systems, such as Napster [52] performed their indexing via centralized servers that contained indexes of the files being shared by each user in the network. This type of indexing is relatively straightforward to implement, but is not truly "peer-to-peer" in the sense that the index server comprises a required, centralized infrastructure. It achieves many of the benefits of peer-to-peer systems: the data is still transferred peer-to-peer, with clients downloading file content from each other, and the index server needs to store only pointers to the file data. Nevertheless, the dependence on the centralized server limits the scalability of the system, as the server must hold information about all the users in the system, as well as limiting its reliability because the server is a central point of failure.

In spite of their disadvantages, certain types of centralized index systems have remained popular. The popular BitTorrent [4] protocol defines only a mechanism for transferring files, rather than searching for files, so it is often used in conjunction with an index website that tracks available torrents and provides a search engine. Moreover, the file transfer protocol itself traditionally relies upon a centralized tracker for each file that keeps track of which peers are sharing or downloading the file and which pieces of the file they have available. An extension to the protocol adds support for distributed trackers based on distributed hash tables, but searching for files can only be done via the aforementioned centralized index websites. Recently, legal action has forced several of these sites to be shut down [72, 15], and perhaps as a result the popularity of BitTorrent has decreased in favor of decentralized networks such as eDonkey [8].

2.1.2 Unstructured Networks

At the opposite end of the design spectrum exist purely unstructured networks, such as Gnutella [27]. Responding to the problems inherent to centralized index systems, these systems are completely decentralized, and hence eliminate central points of failure. In Gnutella, the nodes form a *unstructured overlay network*, in which each node connects to several randomly-chosen nodes in the network. The result is a randomly connected graph. Each node stores only local information about which files it has available, which keeps maintenance cost to a minimum. Queries are performed by flooding: a node wishing to perform a query sends a description of the query to each of its neighboring nodes, and each neighbor forwards it on to its neighbors, etc. Every node receiving the query sends a response back to the original requester if it has a matching file. This query-flooding search mechanism is quite expensive, so these systems also usually scale poorly: not only do heavy query workloads rapidly overwhelm the system, increasing the number of nodes in the system serves to exacerbate the problem rather than ameliorate it because every query must be forwarded to so many nodes.

A number of optimizations have been proposed to improve the scalability of Gnutella-like systems. Both Gia [11] and newer versions of the Gnutella protocol perform queries using random walks instead of flooding: each node forwards a query request to a randomly-chosen neighbor instead of all neighbors. Hence, queries no longer reach every system in the network. As a result, the system sacrifices *perfect recall*: queries for rare data may return no results even if the data is present in the network. Proponents of such designs argue that most queries are for common files for which many replicas exist in the network, and so a random walk is likely to return results, particularly if one-hop replication, in which each node also indexes the contents of its immediate neighbors, is used. In contrast, we have designed our system, Arpeggio, to provide perfect recall.

A common refinement that proves quite effective involves exploiting the heterogeneity of client resources: the connectivity of nodes in peer-to-peer systems ranges from low-bandwidth, high-latency dialup links to high-bandwidth backbone connections. Likewise, the node lifetime ranges from connections lasting under a minute (presumably users performing a single query then disconnecting) to connections lasting for many hours [28]. Hence, allocating greater responsibility to the more powerful nodes is effective for increasing performance. A simple technique for accomplishing this goal is to designate the fastest and most reliable nodes as *supernodes* or *ultrapeers*, which are the only nodes that participate in the unstructured overlay network. The remainder of the nodes act as clients of one or more supernodes.In the FastTrack [31] network (used by the KaZaA and Morpheus applications, among others), the supernodes maintain a list of files that their files are sharing and answer queries directly; in newer versions of the Gnutella protocol [60], clients provide their ultrapeers with a Bloom filter [7] of their files, which allows the ultrapeer to forward only some queries to the client, filtering out queries for which the client is known not to have a match. Gia uses a more sophisticated technique that can exploit finer-grained differences in client resources: the degree of each node in the graph is proportional to its resources, and search requests are biased towards nodes with higher degree.

2.1.3 Structured Overlay Networks

Structured peer-to-peer overlay networks based on *distributed hash tables* (*DHTs*) [64, 61, 57, 47, 36, 42] strike a balance between the centralized index and unstructured overlay designs. While the system remains fully decentralized, a distributed algorithm assigns responsibility for certain subsets of the data to particular nodes, and provides a means for efficiently routing messages between nodes while maintaining minimal state. As a result, they provide efficiency closer to that of a centralized index while retaining the fault-tolerance properties of a decentralized system.

Though the precise details differ between DHT designs, the lowest layer is fundamentally a *distributed lookup service* which provides a LOOKUP operation that efficiently identifies the node responsible for a certain piece of data. In the Chord algorithm [64], which is described in detail in Section 2.2, this LOOKUP operation requires $O(\log n)$ communications, and each node must be connected to $O(\log n)$ other nodes, where n is the total number of nodes in the network. Other DHTs make different tradeoffs between LOOKUP cost and node degree.

Building on this primitive, DHash [17] and other *distributed hash tables* (DHTs) implement a storage layer based on a standard GET-BLOCK/PUT-BLOCK hash table abstraction. The storage layer makes it possible to store data in the network by handling the details of replication and synchronization.

Distributed hash tables are a natural fit for developing peer-to-peer systems because they are efficient and decentralized. However, their limited interface does not immediately meet the needs of many applications, such as file sharing networks. Though DHTs provide the ability to find the data associated with a certain key or file name, users often wish to perform keyword searches, where they know only part of the file name or metadata keywords. In other words, the *lookup by name* DHT semantics are not immediately sufficient to perform complex *search by content* queries of the data stored in the network.

2.2 The Chord Protocol

Arpeggio uses the Chord lookup protocol as a basis for its indexing system. Though Arpeggio mainly uses Chord as a “black box”, we provide a brief overview of how the lookup protocol works.

Chord uses *consistent hashing* [37] to map keys (such as file names, or in our system, keywords) to the nodes responsible for storing the associated data. Consistent hashing uses a single, standardized hash function such as SHA-1 [53] to map both nodes and keys into the same circular identifier space. Each node is assigned a 160-bit identifier by taking the SHA-1 hash of its IP address. Likewise, each key is assigned the 160-bit identifier that results from taking its SHA-1 hash. We then assign the responsibility for storing the data associated with a particular key to the first node that follows it in the identifier space, as shown in Figure 2-1.

Because the hash function behaves like a random function, load is distributed approximately uniformly among the nodes in the system. If there are n nodes in the system, each node is responsible for $1/n$ of the keys in expectation, and no more than $O\left(\frac{\log n}{n}\right)$ with high probability; the latter figure is reduced to $O\left(\frac{1}{n}\right)$ if each node operates $O(\log n)$ virtual nodes [37]. It also deals well with dynamic membership changes in the set of nodes: it is easy to see that if a node joins or leaves the system, the only keys that will need to be transferred are among those stored on that node and its neighbors.

As a result, if each node knows the identifiers of all other nodes of the system, it can determine which node is responsible for a certain key simply by calculating a hash function, without communicating with a central index or any other nodes. However, in large peer-to-peer systems, it is impractical for each node to keep a list of all other nodes, both because the list would be very large, and because it changes frequently and keeping it up to date would require large amounts of maintenance traffic.

Hence, it is necessary to have a system to allow nodes to look up the responsible node for a certain key, while maintaining minimal state on each node. The Chord distributed

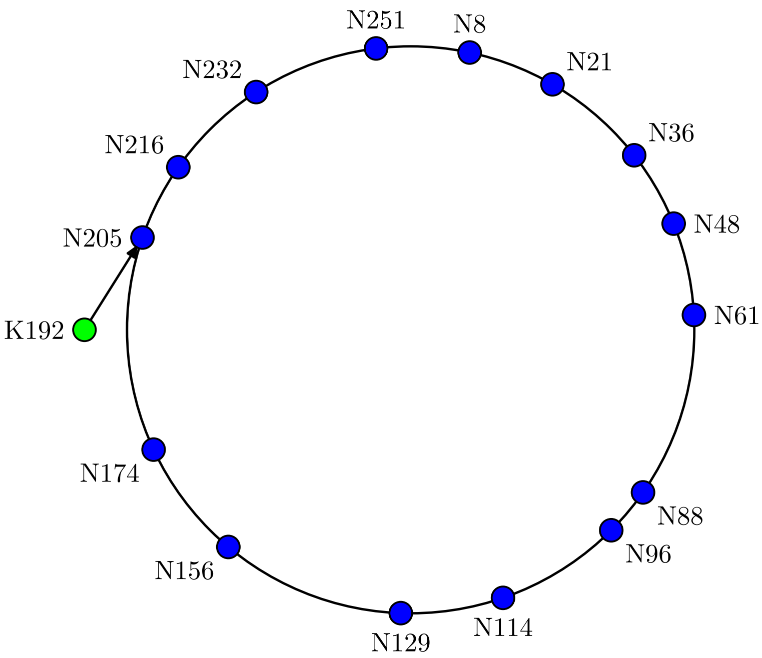


Figure 2-1: *Mapping of keys to nodes using consistent hashing. Key #192 is mapped to node #205 because it is the lowest node number greater than #192.*

lookup protocol fills this need.

As an initial step, suppose that each node in the network contains a pointer to its *successor*: the node with the next larger ID, as shown in Figure 2-2. This ensures that it is always possible to locate the node responsible for a given key by following the chain of successor pointers. It is reasonably straightforward to keep the successor pointers up to date: each node can maintain the addresses of several successors and predecessors, and periodically poll them to determine if they have failed or if a new successor or predecessor has come online.

Though maintaining correct successor pointers guarantees correctness, performing a lookup still requires $O(n)$ hops in the average case, giving rather unsatisfying performance. To improve performance, each node also maintains $O(\log n)$ finger pointers: a node with id n knows the successor of keys $\left(n + 2^i \pmod{2^{160}}\right)$ for all values of i, as shown for key #8 in Figure 2-3. This corresponds to knowing the location of nodes halfway around the ring, 1/4th across the ring, 1/8th across, etc. As a result, Lookup operations can be performed using only $O(\log n)$ messages; intuitively, this is because each hop reduces the distance to the destination (in terms of keyspace) by at least half. Though proving this in a dynamic network is far more involved, a maintenance protocol can ensure that the system remains sufficiently stable to allow logarithmic lookups even as nodes constantly join and leave the system [43].

Finally, the DHash storage layer lies atop the Chord lookup service. This layer stores the actual data, using a hash table Get/Put interface, where keys are associated with data. Since nodes can fail or leave the system without notice, in order to ensure durability the data must be replicated on multiple nodes. This is achieved by storing the data not only on the immediate successor of the key, but on the next k successors; nodes periodically poll their neighbors to ensure that they are still functioning and enough copies of the data exist. The value of k can be chosen to provide the desired balance between minimizing the

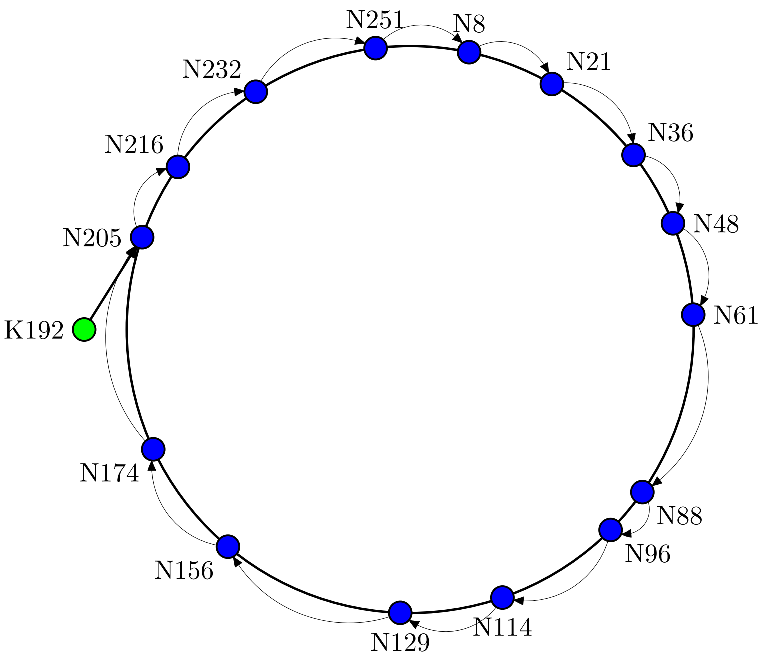


Figure 2-2: *Successor pointers in a Chord ring*

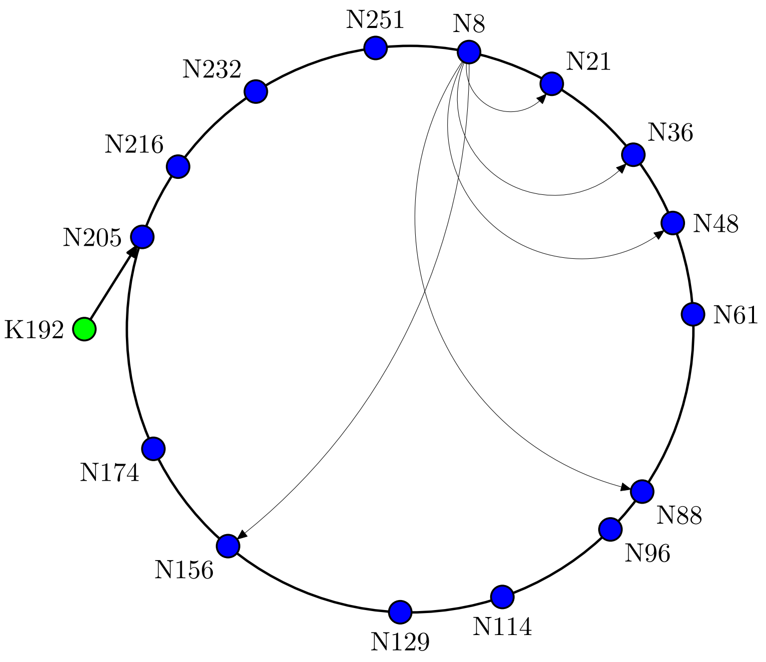


Figure 2-3: *Finger pointers for one node in a Chord ring*

probability of failure and keeping the required maintenance traffic low [12].

Many optimizations are possible in the storage layer, such as using erasure codes such as IDA [56] instead of replication to improve data durability and reduce maintenance costs [74], or using a protocol based on Merkle trees [49] to more efficiently synchronize replicas [10].

Generality of Arpeggio. Though we present our Arpeggio indexing algorithms in terms of Chord in Chapter 3 and our implementation described in Chapter 5 uses Chord as its basis, we emphasize that this is not a requirement of the system. Chord can be replaced as the underlying substrate of the system with any other protocol that provides the ability to route to the nodes responsible for a particular key. In particular, it could be used in conjunction with the Kademlia DHT [47] already deployed in the trackerless BitTorrent system [6]. Note, however, that because Arpeggio requires a more complex interface than the GET/PUT interface that distributed hash tables provide, the nodes in the system must be modified to support the extended query processing interface. This is discussed in Section 5.5.

Chapter 3

Indexing Techniques

The central problem for a content-sharing system is searching: it must be able to translate a search query from a user into a list of files that fit the description and a method for obtaining them. In Arpeggio, we concern ourselves with file *metadata* only. As discussed previously, each file shared on the network has an associated set of metadata. We only allow searches to be performed on the small amount of metadata associated with the file, rather than the contents, which are potentially much larger.

Limiting searches to file metadata is an important restriction that enables us to use a number of algorithms that would not be suitable for full-text search of documents. Indeed, Li et al. [41] investigated the communications costs that would be required to perform full-text search of the web using a distributed index, and found that the size of the data set made the problem infeasible using current techniques. Our work does not contradict these findings; rather, it addresses a different problem. Peer-to-peer indexing for metadata remains feasible, because the datasets we consider have a small number of metadata keywords per file (on the order of 5–10), which is much smaller than the full text of an average Web page.

3.1 Design Evolution

Our indexing techniques are designed to ensure that the system scales with the addition of new nodes, and to ensure that the indexing cost is shared equally among the nodes rather than causing one node to bear a disproportionate amount of the load. To justify our design decisions, we present the evolution of our indexing system as a series of possible designs, exploring the problems with each and how to address them.

3.1.1 Inverted Indexes

The standard building block for performing searches is, of course, the *inverted index*: a map from each search term to a list of the documents that contain it. Such indexes can be searched by scanning over all entries in an index to identify documents that match the full query, or finding the intersection of the indexes corresponding to each search term.

There are two main approaches to distributing an inverted index. In *partition-by-document* indexing, each node in the system stores all of the inverted index entries corresponding to some subset of the indexed documents. To perform a query, the system must contact a set of nodes that collectively hold index entries for every document; if index

```
Query(criteria)
indexes ← ∅
for keyword, value in criteria
        do indexes.Append(Get(keyword))
return Intersection(indexes)
```

Figure 3-1: *Pseudocode for distributed inverted index*

entries are not replicated, this means sending the query to every node in the system. In *partition-by-keyword* indexing, each node stores the inverted index entries corresponding to some range of *keywords*.

Both approaches have been employed in a number of systems. Partition-by-document indexing is used for web searches by Google [3], and for indexing scholarly publications in OverCite [65]. Li et al. [41] observed that naive implementations of partition-by-document indexing require three orders of magnitude less bandwidth to perform a search than naive implementations of partition-by-keyword indexing. Though this appears to be a compelling argument for a partition-by-document index, we instead base our system on partition-by-keyword indexing. We note that there are a number of optimizations that can be employed to reduce the bandwidth cost, which we explore in Sections 3.1.2 and 3.1.3. Moreover, partition-by-document indexing requires contacting nodes from each index partition. In the two partition-by-document systems described above, this is reasonable: Google's index servers are likely to be in the same cluster, and thus enjoy low latency; OverCite has a relatively small number of nodes, each of which holds as much as $1/2$ of the whole index. In a large-scale peer-to-peer system, it is not practical to contact so many nodes or store so much index data on each node. Partition-by-keyword indexing, on the other hand, maps nicely to the interface provided by consistent hashing and distributed hash tables, allowing us to perform queries while contacting a minimal number of nodes.

3.1.2 Partition-by-Keyword Distributed Indexing

A partition-by-keyword inverted index can be implemented in a relatively straightforward way on top of a distributed hash table: each keyword maps to the server responsible for storing its inverted index. To perform a query, the client can fetch each index list from the DHT, and calculate their intersection, as shown in Figure 3-1. No modifications to the standard DHT Get/Put interface are required.

This naive approach scales poorly to large numbers of documents. Each keyword index list must be transferred in its entirety; these keyword index lists can become prohibitively long, particularly for very popular keywords, so retrieving the entire list may generate tremendous network traffic.

There is clearly room for optimization, as the desired intersection of the indexes is typically much smaller than any one of the index lists, yet in the scheme above the full index lists are transferred. Bloom filters [7] use hashes to encode a summary of the contents of a set in a small data structure; set inclusion queries are probabilistic, with no false negatives but some chance of a false positive. Reynolds and Vahdat [58] proposed a protocol in which one index server transmits a Bloom filter of its results to another, allowing the second to determine a conservative superset of the intersection to send to the client. Li

QUERY(*criteria*)
1 *index* ← RANDOM-KEYWORD(*criteria*)
2 **return** FILTERED-GET(*index*, *criteria*)

Figure 3-2: *Pseudocode for distributed inverted index*

et al. [41] proposed several further optimizations, including compression of the index lists and adaptive intersection algorithms [18]. These optimizations can substantially reduce the amount of bandwidth required to find the index intersection; however, we do not employ them in Arpeggio because our restrictions on the type of data being indexed enables the following more effective optimization.

3.1.3 Index-Side Filtering

Performance of a keyword-based distributed inverted index can be improved by performing *index-side filtering* instead of performing an index intersection at the querying node or a distributed index join. Because our application postulates that metadata is small, the entire contents of each item's metadata can be kept in the index as a *metadata block*, along with information on how to obtain the file contents. To perform a query involving a keyword, we send the full query to the corresponding index node, and it performs the filtering and returns only relevant results. This dramatically reduces network traffic at query time, since only one index needs to be contacted and only results relevant to the full query are transmitted. This appears to be similar to the search algorithm used by the Overnet network [54], which uses the Kademlia DHT, though details of its operation are difficult to find. The same technique is also used in eSearch [67].

Though this seems like a trivial change, it represents an important change in the role of the DHT from passive storage element to active processing component. The procedure of transmitting the query to the index server for execution is reminiscent of the use of searchlets in Diamond [32], though far simpler. It requires extending the DHT interface beyond the standard GET-BLOCK/PUT-BLOCK hash table abstraction; our implementation accesses the underlying Chord LOOKUP function to build more powerful functions involving network-side processing. As a result, our algorithm cannot be deployed atop an existing DHT service, such as OpenDHT [59] without modification. This presents a deployment problem as several BitTorrent clients [6, 2] already operate their own (mutually incompatible) DHTs; we cannot use them to store index data without modifying all existing clients. We discuss this in Section 5.5.

3.1.4 Keyword-Set Indexing

While index-side filtering reduces network usage, query load may still be unfairly distributed, overloading the nodes that are responsible for indexing popular keywords. To overcome this problem, we propose to build inverted indexes not only on individual keywords but also on keyword *sets*: pairs, triples, etc. up to some maximum size. As before, each unique file has a corresponding metadata block that holds all of its metadata. Now, however, an identical copy of this metadata block is stored in an index corresponding to each subset of at most K metadata terms. The maximum set size K is a parameter of the network. This is the Keyword-Set Search system (KSS) introduced by Gnawali [26].

Essentially, this scheme allows us to precompute the index intersection for all queries of up to K keywords. It is not a problem if a particular query has more than K keywords: the query can be sent to the index server for a randomly chosen K-keyword subset of the query, and the index server can filter the results to only send back responses that match the full query. This approach has the effect of querying smaller and more distributed indexes whenever possible, thus alleviating unfair query load caused by queries of more than one keyword.

The majority of searches contain multiple keywords, as shown in Reynolds and Vahdat's analysis of web queries [58] and our analysis of peer-to-peer query traffic in Section 6.2. As a result, most queries can be satisfied by the smaller, more specific multiple-keyword indexes, so the larger single-keyword indexes are no longer critical to result quality. To reduce storage requirements, maximum index size can be limited, preferentially retaining entries that exist in fewest other indexes, i.e. those with fewest total keywords.

In Arpeggio, we combine KSS indexing with index-side filtering, as described above: indexes are built for keyword sets and results are filtered on the index nodes. We make a distinction between *keyword metadata*, which is easily enumerable and excludes stopwords, and therefore can be used to partition indexes with KSS, and *filterable metadata*, which is not used to partition indexes but can further constrain a search. Index-side filtering allows for more complex searches than KSS alone. A user wish to restrict his or her keyword query to files of size greater than 1 MB, files in `tar.gz` format, or MP3 files with a bitrate greater than 128 Kbps, for example. It is not practical to encode this information in keyword indexes, but the index obtained via a KSS query can easily be filtered by these criteria. The combination of KSS indexing and index-side filtering increases both query efficiency and precision.

Nevertheless, the index structure imposes limitations on the type of queries supported. Because keywords or keyword sets must be used to identify the index to be searched, the system cannot support general substring queries (though it is possible to use index-side filtering to restrict the output of a keyword search to those also matching a substring or regular expression query). We argue that this is not a major limitation, as it is shared by many widely-used systems that exist today. In addition to being a limitation inherent to systems based on partition-by-keyword distributed indexes, the use of Bloom filters in Gnutella's query routing protocol [60] also limits the user to keyword queries.

3.2 Indexing Cost

Techniques such as KSS improve the distribution of indexing load, reducing the number of very large indexes — but they do so by creating more index entries. In order to show that this solution is feasible, we argue that the increase in total indexing cost is reasonable.

Using keyword set indexes rather than keyword indexes increases the number of index entries for a file with m metadata keywords from m to $I(m)$, where

$$I(m) = \sum_{i=1}^{K} \binom{m}{i} = \begin{cases} 2^m - 1 & \text{if } m \leq K \\ O(m^K) & \text{if } m > K \end{cases}$$

For files with many metadata keywords, $I(m)$ is polynomial in m. If m is small compared to K (as for files with few keywords), then $I(m)$ is exponential in m — but this is no worse, since m is so small anyway. The graph in Figure 3-3 shows that $I(m)$ grows polynomially

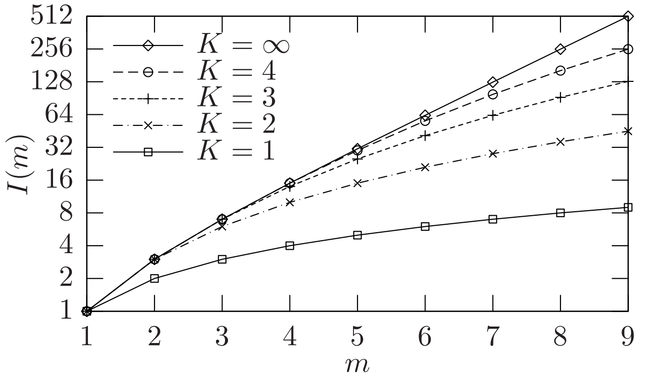


Figure 3-3: *Growth of $I(m)$ for various values of K*

with respect to m, and its degree is determined by K. As discussed in Section 6.2.1, for many applications the desired value of K will be small (around 3 or 4), and so $I(m)$ will be a polynomial of low degree in m.

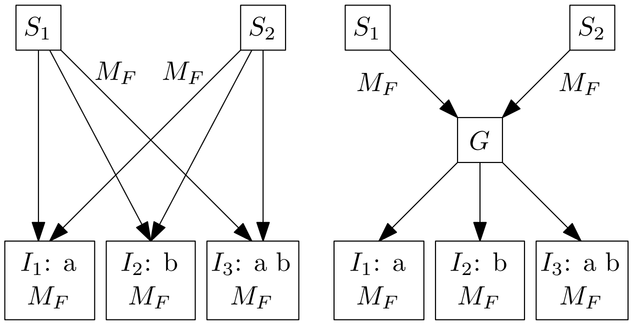
3.3 Index Maintenance

Peers are constantly joining and leaving the network. In addition to causing index servers to become unavailable, this churn also causes new files to become available and old files to disappear as the nodes containing them join and leave the network. Thus, the search index must respond dynamically to the shifting availability of the data it is indexing and the nodes on which the index resides. Furthermore, certain changes in the network, such as nodes leaving without notification, may go unnoticed, and polling for these changing conditions is too costly, so the index must be maintained by passive means.

3.3.1 Metadata Expiration

Instead of actively polling for the departure of source nodes, or expecting source nodes to send a notification before leaving, the index servers expire file metadata entries on a regular basis so that long-absent files will not be returned by a search. Nevertheless, during the expiration delay indexes may contain out-of-date references to files that are no longer accessible. Thus, a requesting peer must be able to gracefully handle failure to contact source peers. To counteract expiration, source peers periodically *refresh* metadata that is still valid by re-inserting the metadata blocks for their files, thereby resetting its expiration counter.

The choice of expiration time balances index freshness against maintenance cost: a short expiration time means that the index will not contain many invalid entries, but that peers must constantly send refresh messages for all their files. Though choosing a short expiration time for index freshness seems desirable at first glance, we argue in Section 4.3.3 that there can be value in long expiration times for metadata, as it not only allows for low refresh rates, but for tracking of attempts to access missing files in order to artificially replicate them to improve availability.



(a) *Without gateway* **(b)** *With gateway*

Figure 3-4: *Two source nodes $S_{1,2}$, inserting file metadata block M_F with keywords $\{a, b\}$ to three index nodes $I_{1,2,3}$, with and without a gateway node G*

3.3.2 Index Gateways

If each node directly maintains its own files' metadata in the distributed index, and multiple nodes are sharing the same file, the metadata block for each file will be inserted repeatedly. Consider a file F that has m metadata keywords and is shared by s nodes. Then each of the s nodes will attempt to insert the file's metadata block into the $I(m)$ indexes in which it belongs, as in Figure 3-4a. The total cost for inserting the file is therefore $\Theta(s \cdot I(m))$ messages. Since metadata blocks simply contain the keywords of a file, not information about which peers are sharing the file, each node will be inserting the *same* metadata block repeatedly. This is both expensive and redundant. Moreover, the cost is further increased by each node repeatedly renewing its insertions to prevent their expiration.

To minimize this redundancy, we introduce an *index gateway* node that aggregates index insertion requests. Index gateways are not required for correct index operation, but they increase the efficiency of index insertion using the standard technique of introducing an intermediary. With gateways, rather than directly inserting a file's metadata blocks into the index, each peer sends a single copy of the block to the gateway responsible for the block (found via a Chord LOOKUP of the block's hash), as in Figure 3-4b. The gateway then inserts the metadata block into all of the appropriate indexes, but only if necessary. If the block already exists in the index and is not scheduled to expire soon, then there is no need to re-insert it into the index. A gateway only needs to refresh metadata blocks when the blocks in the network are due to expire soon, but the copy of the block held by the gateway has been more recently refreshed.

Gateways dramatically decrease the total cost for multiple nodes to insert the same file into the index. Using gateways, each source node sends only one metadata block to the gateway, which is no more costly than inserting into a centralized index. The index gateway only contacts the $I(m)$ index nodes once, thereby reducing the total cost from $\Theta(s \cdot I(m))$ to $\Theta(s + I(m))$. For files that are shared by only a single source node, using the gateway only negligibly increases the insertion cost, from $I(m)$ to $I(m)+1$. Another benefit is that it spreads the cost of insertion among the nodes in the network. A common scenario is that a node joining the network will want to register all of the files it has available with the index. Without index gateways, it would have to contact every index server; with index gateways, it only needs to contact the gateways, and the cost of contacting the index servers is spread amongst the gateways — which, thanks to consistent hashing, are likely to be distributed equally around the network.

3.3.3 Index Replication

In order to maintain the index despite node failure, index replication is also necessary. As described in Section 2.2, DHTs typically accomplish this by storing multiple copies of data on replicas determined via the lookup algorithm, or storing erasure-coded fragments of the data in the same way. The choice between erasure coding and replication affects not only the probability of data loss during node failures, but also performance: replication can provide lower read latency because only a single replica needs to be contacted [17]. Because metadata blocks are small and our application requires reading from indexes to have low latency, we opt for replication over erasure coding. Moreover, it is not straightforward to implement server-side processing elements such as index-side filtering in an erasure-coded system.

Furthermore, because replicated indexes are independent, any node in the index group can handle any request pertaining to the index (such as a query or insertion) without interacting with any other nodes. Arpeggio requires only *weak consistency* of indexes, so index insertions can be propagated periodically and in large batches as part of index replication. Expiration can be performed independently by each index server.

Chapter 4

Content Distribution

The indexing system described in the previous chapter addresses one of the two central challenges of a file sharing system: it provides the means to perform searches for files. The other major problem remains to be addressed: finding sources for a particular file and beginning to download it.

Our indexing system is independent of the file transfer mechanism. Thus, it is possible to use an existing content distribution network in conjunction with Arpeggio's indexing system. For example, a simple implementation might simply store a HTTP URL for the file as an entry in the metadata blocks. If using a content-distribution network such as Coral [22] or CoDeeN [73] , the appropriate information can be stored in the metadata block. This information is, of course, opaque to the indexing system.

4.1 Direct vs. Indirect Storage

A DHT can be used for direct storage of file contents, as in distributed storage systems like CFS, the Cooperative File System [16]. In these systems, file content is divided into chunks, which are stored using the DHT storage layer. Because the storage layer ensures that the chunks are stored in a consistent location and replicated, the system achieves the desirable availability and durability properties of the DHT. The cost of this approach is maintenance traffic: as nodes join and leave the network, data must be transferred and new replicas created to ensure that the data is properly stored and replicated. For a file sharing network, direct storage is not a viable option because the amount of churn and the size and number of files would create such high maintenance costs.

Instead, Arpeggio uses content-distribution systems based on *indirect storage*: it allows the file content to remain on the peers that already contain it, and uses the DHT to maintain pointers to each peer that contains a certain file. Using these pointers, a peer can identify other peers that are sharing content it wishes to obtain. Because these pointers are small, they can easily be maintained by the network, even under high churn, while the large file content remains on its originating nodes. This indirection retains the distributed lookup abilities of direct storage, while still accommodating a highly dynamic network topology, but may sacrifice content availability, since file availability is limited to those files that are shared by peers currently existing in the network.

In this chapter, we discuss several possible designs of content-distribution systems that can be used in conjunction with Arpeggio's indexing functionality.

4.2 A BitTorrent-based System

In earlier versions of our design [13], we proposed a comprehensive system including its own content distribution mechanism, which we describe below (Section 4.3). However, after our system was designed, BitTorrent increased in popularity and gained a decentralized tracker method that made it more suitable for use in a system like ours. As a result, we implemented the following system, which combines Arpeggio's indexing mechanism with BitTorrent's transfer capabilities.

BitTorrent [4] is an extremely popular protocol for peer-to-peer file downloads, because it achieves high download speeds by downloading from multiple sources. It uses a distributed tit-for-tat protocol to ensure that each peer has incentives to upload to others [14]. Besides this use of incentives, BitTorrent differs from other peer-to-peer file sharing systems in that it does not provide any built-in searching capabilities, relying instead on index websites. The information needed to download a set of files is stored in a `.torrent` file, including the file names, sizes, and content hashes, as well as the location of a tracker. The tracker maintains a list of peers currently downloading or sharing (*seeding*) the file, and which pieces of the file they have available.

Since a centralized tracker has many disadvantages, implementers of BitTorrent clients introduced decentralized mechanisms for finding peers. In the Peer Exchange protocol [55], peers gossip with each other to discover other peers. Though this idea is employed by many of the most popular clients (albeit with mutually incompatible implementations), it only functions in conjunction with a tracker. Trackerless systems [6] have been implemented that use a Kademlia DHT to maintain lists of peers for each torrent. Both the official BitTorrent client [5] and the Azureus client [2] support this approach, though again with mutually incompatible implementations.

With trackerless capabilities, BitTorrent serves as an effective content distribution system for Arpeggio. We must simply add a mechanism for mapping the files found by a metadata search to a `torrent` file for download. It is not desirable to store the contents of the `torrent` file in the file's metadata block, because many copies of the metadata block are created for KSS indexing. Instead, we employ a layer of indirection: the metadata block contains an identifier which can be used to look up the `torrent` file via a DHT lookup. Indeed, support for this level of indirection is already built-in to some BitTorrent clients, such as Azureus: these clients support Magnet-URIs [46], which are a content-hash-based name that can be resolved via a DHT lookup. For clients that do not support Magnet-URI resolution, a similar step can be implemented using Arpeggio's DHT. Note also that for torrents that contain more than one file, a separate metadata block with the same torrent identifier can be created for each file, making it possible to search for any of the individual files.

4.3 A Novel Content-Distribution System

Though the BitTorrent-based system above is practical and uses existing technologies in order to ease deployment, starting from scratch allows us to develop a more comprehensive system tuned to some of the unique requirements of a peer-to-peer file sharing network. Below, we describe some salient features of a novel content distribution system we have designed but have yet to implement; it includes mechanisms for sharing content between similar files, efficiently locating sources for a file, and for improving the availability of

Table 4.1: *Layers of lookup indirection*

Translation	Method
keywords → file IDs	keyword-set index search
file ID → chunk IDs	standard DHT lookup
chunk ID → sources	content-sharing subring

popular but rare files.

4.3.1 Segmentation

For purposes of content distribution, we segment all files into a sequence of *chunks*. Rather than tracking which peers are sharing a certain file, our system tracks which chunks comprise each file, and which peers are currently sharing each chunk. This is implemented by storing in the DHT a *file block* for each file, which contains a list of *chunk IDs*, which can be used to locate the sources of that chunk, as in Table 4.1. File and chunk IDs are derived from the hash of their contents to ensure that file integrity can be verified.

The rationale for this design is twofold. First, peers that do not have an entire file are able to share the chunks they do have: a peer that is downloading part of a file can at the same time upload other parts to different peers. This makes efficient use of otherwise unused upload bandwidth. For example, Gnutella does not use chunking, requiring peers to complete downloads before sharing them. Second, multiple files may contain the same chunk. A peer can obtain part of a file from peers that do not have an exactly identical file, but merely a *similar* file.

Though it seems unlikely that multiple files would share the same chunks, file sharing networks frequently contain multiple versions of the same file with largely similar content. For example, multiple versions of the same document may coexist on the network with most content shared between them. Similarly, users often have MP3 files with the same audio content but different ID3 metadata tags. Dividing the file into chunks allows the bulk of the data to be downloaded from any peer that shares it, rather than only the ones with the same version.

However, it is not sufficient to use a segmentation scheme that draws the boundaries between chunks at regular intervals. In the case of MP3 files, since ID3 tags are stored in a variable-length region of the file, a change in metadata may affect all of the chunks because the remainder of the file will now be "out of frame" with the original. Likewise, a more recent version of a document may contain insertions or deletions, which would cause the remainder of the document to be out of frame and negate some of the advantages of fixed-length chunking.

To solve this problem, we choose variable length chunks based on content, using a chunking algorithm derived from the LBFS file system [51]. Due to the way chunk boundaries are chosen, even if content is added or removed in the middle of the file, the remainder of the chunks will not change. While most recent networks, such as FastTrack, BitTorrent, and eDonkey, divide files into chunks, promoting the sharing of partial data between peers, our segmentation algorithm additionally promotes sharing of data *between files*.

4.3.2 Content-Sharing Subrings

To download a chunk, a peer must discover one or more sources for this chunk. A simple solution for this problem is to maintain a list of peers that have the chunk available, which can be stored in the DHT or handled by a designated "tracker" node as in BitTorrent [4]. However, the node responsible for tracking the peers sharing a popular chunk represents a single point of failure that may become overloaded.

We instead use *subrings* to identify sources for each chunk, distributing the query load throughout the network. The Diminished Chord protocol [38] allows any subset of the nodes to form a named "subring" and allows LOOKUP operations that find nodes in that subring in $O(\log n)$ time, with constant storage overhead per node in the subring. We create a subring for each chunk, where the subring is identified by the chunk ID and consists of the nodes that are sharing that chunk. To obtain a chunk, a node performs a LOOKUP for a random Chord ID in the subring to discover the address of one of the sources. It then contacts that node and requests the chunk. If the contacted node is unavailable or overloaded, the requesting node may perform another LOOKUP to find a different source. When a node has finished downloading a chunk, it becomes a source and can join the subring. Content-sharing subrings offer a general mechanism for managing data that may be prohibitive to manage with regular DHTs.

4.3.3 Postfetching

To increase the availability of files, our system caches file chunks on nodes that would not otherwise be sharing the chunks. Cached chunks are indexed the same way as regular chunks, so they do not share the disadvantages of direct DHT storage with regards to having to maintain the chunks despite topology changes. Furthermore, this insertion symmetry makes caching transparent to the search system. Unlike in direct storage systems, caching is non-essential to the functioning of the network, and therefore each peer can place a reasonable upper bound on its cache storage size.

Postfetching provides a mechanism by which caching can increase the supply of rare files in response to demand. *Request blocks* are introduced to the network to capture requests for unavailable files. Due to the long expiration time of metadata blocks, peers can find files whose sources are temporarily unavailable. The peer can then insert a request block into the network for a particular unavailable file. When a source of that file rejoins the network it will find the request block and actively increase the supply of the requested file by sending the contents of the file chunks to the caches of randomly-selected nodes with available cache space. These in turn register as sources for those chunks, increasing their availability. Thus, the future supply of rare files is actively balanced out to meet their demand.

Chapter 5

Implementation Notes

5.1 Architecture

Our implementation of Arpeggio is divided into three layers, as shown in Figure 5-1; for maximum flexibility, the core indexing functionality is separated from the underlying DHT implementation and from the user interface. At the lowest level, the routing layer encapsulates the functions provided by the DHT. The indexing layer implements the algorithms described in Chapter 3, making it possible to index and search file metadata. Finally, the user interface layer has several components, making it possible to access the indexing functionality through various means.

5.2 Routing Layer

The routing layer of Arpeggio provides the DHT's LOOKUP primitive. It is separated into its own layer, with an extremely narrow RPC interface, for two reasons. First, this separation makes it possible to substitute a different DHT instead of Chord, as the upper layers of the system are independent from the details of the DHT algorithm. Second, for ease of implementation, it allows the Arpeggio codebase to be separated from that of the DHT.

Our implementation of Arpeggio's routing layer consists of a daemon called `cd` (*Chord daemon*)[1]. This daemon is implemented in C++, using the `libasync` event-driven framework [48], and links against the MIT Chord codebase. It accepts remote procedure calls and presents the interface described in Table 5.1. This interface essentially consists of the bare LOOKUP primitive. Notably, it does not include the higher-level GET-BLOCK/PUT-BLOCK DHT storage layer functions; implementing a replicated storage layer is left to the indexing layer in order to enable techniques such as index-side filtering (Section 3.1.3).

In order to enable replication, the RPC interface also includes functions for obtaining the current successor and predecessor lists. These are used for implementing index replication, since in Chord a node will only replicate data that is stored on an adjacent node. For use with other DHTs, these functions could be replaced by their more general equivalent: finding the set of nodes on which data should be replicated.

[1]In retrospect, it should have been clear that choosing a name identical to that of a common UNIX shell command was not conducive to sanity.

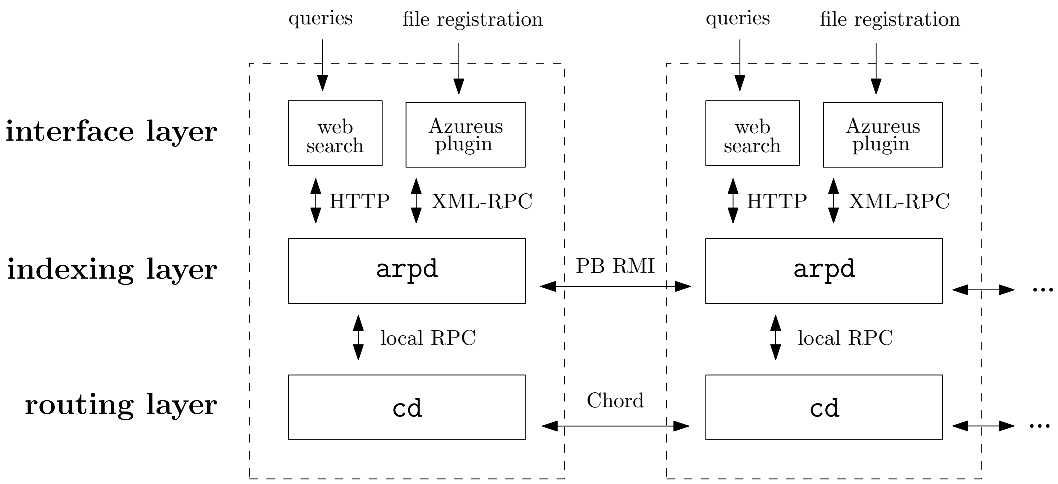


Figure 5-1: *Arpeggio implementation architecture: key modules*

Table 5.1: `cd` *RPC interface*

Function	Description
`newchord`	Create a Chord instance and join a ring
`unnewchord`	Shut down the Chord instance
`lookup`	Look up the successors of a key
`getsucclist`	Return the list of successors of our node
`getpredlist`	Return the list of predecessors of our node

5.3 Indexing Layer

The indexing layer consists of the Arpeggio daemon, `arpd`, which implements the Arpeggio indexing algorithms. `arpd` accepts high-level commands such as "add metadata to index" from the user interface, via a simple RPC interface described in Table 5.2.

`arpd` is implemented in Python, using the Twisted asynchronous framework [20]. Each `arpd` node presents several interfaces, as shown in Figure 5-1. `arpd` communicates with the Chord daemon, `cd`, via a local RPC connection over a Unix domain socket in order to perform Chord lookups; it communicates with other instances of `arpd` on other nodes in the network via Twisted's Perspective Broker remote method invocation system [68]. Finally, it operates a HTTP interface for the user interface.

To participate in Arpeggio indexing, each node in the network operates an `arpd` daemon. Upon startup, `arpd` starts the Chord daemon, `cd`, and establishes a local RPC connection to it over a Unix domain socket. `arpd` then creates a Chord node, and connects it to a user-specified well-known node for bootstrap purposes. Once the connection is established, the local `arpd` begins communicating with its neighbors in the Chord ring in order to begin

Table 5.2: `arpd` *RPC interface presented to UI*

Function	Description
`insert`	Add a metadata block to the appropriate indexes
`search`	Perform a search for files matching a given query

Table 5.3: *`arpd`–`arpd` RPC interface*

Function	Description
`add`	Add a metadata block to all appropriate indexes on this node
`search`	Search a particular index for metadata blocks matching a query
`gatewayInsert`	Request that an index gateway refresh indexes if necessary
`offerBlocks`	Inform a remote node about blocks that it should replicate
`requestBlocks`	Request a set of blocks (by hash) from a remote node

synchronizing the data it is now responsible for replicating.

Each node acts as a client, an index gateway, and an index server. Upon receiving a request to insert a file into the index, `arpd` performs a Chord lookup for the key corresponding to the file. It contacts the `arpd` instance on that node, which acts as an index gateway, generating the keyword sets and contacting whichever index servers are necessary. As an index server, the node receives inserted blocks, answers queries, and periodically checks for block expiration and participates in the index replication protocol.

Each index entry is stored on k replicas, where k is a parameter that can be adjusted based on a tradeoff between index data durability and maintenance cost; our implementation currently uses $k = 3$. Nodes periodically participate in a replication protocol to ensure that each of the replicas responsible for an index has all of the data associated with that index. This protocol operates as follows: periodically, or when churn is detected, a node contacts its k successors and predecessors and sends it a "Offer-Blocks" request, providing the hashes of every index entry it has that the other replica should have. Note that this can be computed locally, because of the use of consistent hashing to map indexes to responsible nodes. The remote node uses the hashes to see if any index entries are missing; if so, it sends a "Request-Blocks" RPC to the original node to request their contents. As an optimization, the expiration time for each block is included in the "Offer-Blocks" request so that if an index entry has simply been refreshed, its contents do not need to be retransmitted.

This replication protocol is greatly simplified compared to some alternatives, such as the Merkle-tree-based synchronization protocol used by DHash [10]. We have chosen this option primarily for simplicity of implementation.

5.4 User Interface

The implementation of Arpeggio is designed such that the user interface is separate from the indexing implementation. This means that a variety of user interfaces can be combined with the indexing core, similar in spirit to the giFT file sharing program [24]. This allows it to be used with different file sharing programs, or for general searching. For example, a set of command-line tools can be used to add files to the index or perform searches; these can coexist with plugins in BitTorrent or Gnutella clients.

At a high level, the interface requires two components: a means for the user to perform searches, and a mechanism for registering shared files with the index. Here, we present an interface that meets these requirements with a web-based search interface, and a plugin for a popular BitTorrent client that indexes shared torrents.

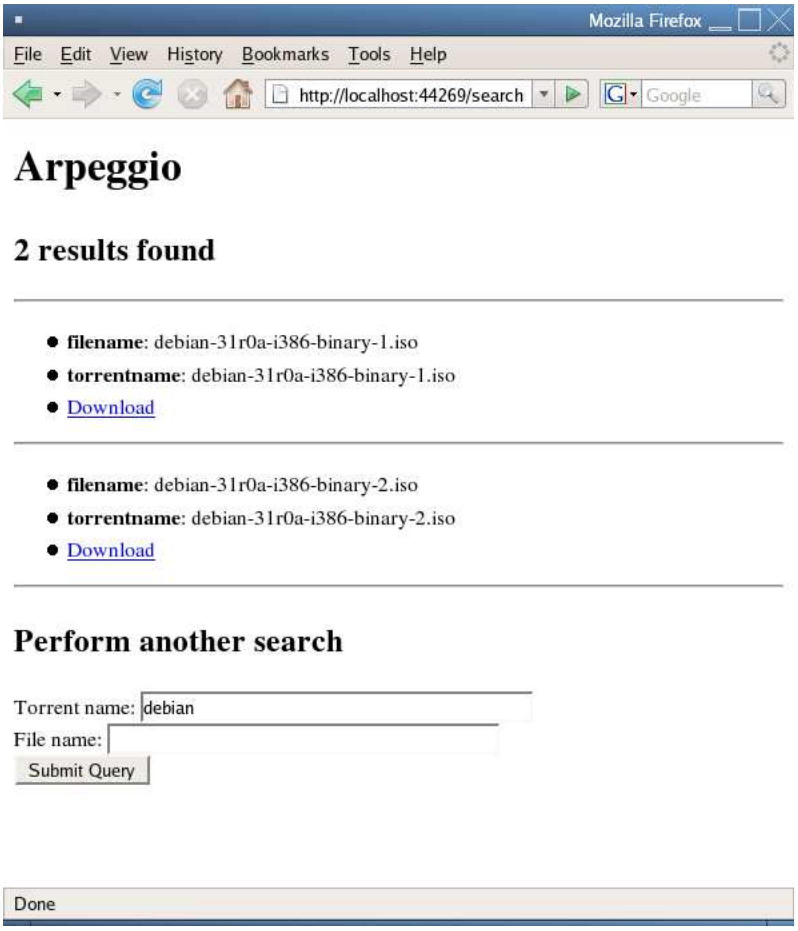


Figure 5-2: *Screenshot of web search interface*

5.4.1 Web Search Interface

By embedding a HTTP server into the `arpd` daemon, the web interface allows the user to perform searches simply by pointing a web browser at the local host on a certain port. The interface is very simple, and consists simply of a search form that allows keyword queries, as shown in Figure 5-2; the field names for metadata entries are configurable.

We choose to use a web interface because of its near-universality; it can be used in conjunction with any file sharing client, or by itself. In particular, users of BitTorrent clients are often accustomed to performing their searches on tracker websites, and many clients, such as `ktorrent` [40], include integrated web browsers for accessing search sites.

Each result returned via the web interface includes a link for downloading the file. As described in Section 4.2, this is a magnet link containing the file's hash, which many clients can use to initiate a download.

5.4.2 Azureus Indexing Plugin

Besides searching, the other necessary component of a user interface is a system for registering shared content with the index. This can be achieved by adapting a file sharing client to notify the Arpeggio indexing daemon of which files it has available. We have implemented

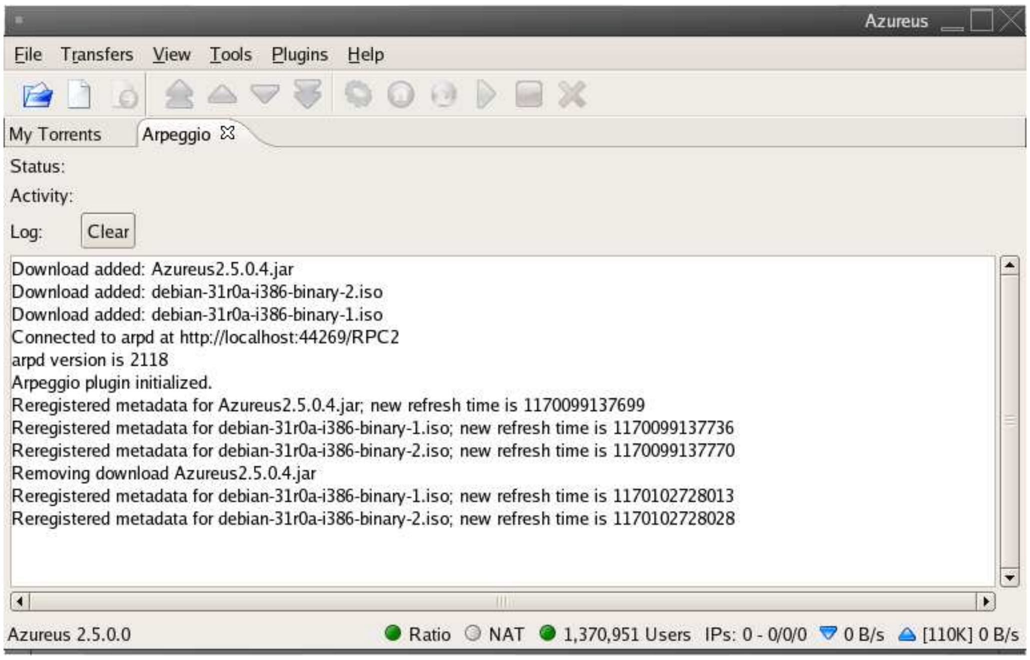


Figure 5-3: *Azureus indexing plugin interface*

this functionality as a plugin for the popular Azureus BitTorrent client.

The Arpeggio plugin for Azureus operates in the background, without user interaction; its only interface is a status display shown in Figure 5-3. The plugin consists of a set of *metadata extractors* that examine the torrent files and generate metadata entries, plus a core that combines the generated metadata and sends it to the local `arpd` daemon. These metadata extractor plugins can generate metadata that is always available, such as the keywords from the torrent's name and the files contained within or the hash of the file, or metadata entries for certain types of files, such as the ID3 tags of a MP3 file or their equivalent for other media files. The plugin communicates with `arpd` over a local XML-RPC [75] connection. It periodically sends requests to re-register the metadata for files being shared, in order to keep the index entries from expiring.

5.5 Deployability

Though Arpeggio has been implemented, ensuring that it is widely adopted requires some further issues to address. The system must be easy for end-users to use, and provide an incentive for users to use it over the current alternatives.

The current implementation of Arpeggio uses the MIT Chord implementation in its routing daemon (`cd`). This presents a problem as the MIT Chord implementation is a research prototype, and cannot easily be used by end users. It requires many dependencies and can be difficult to build; it also runs only on Unix-based systems, not under Windows. To address this problem, `cd` can be replaced with a more robust implementation, using a either a different implementation of Chord or a different distributed hash table, such as the Kademlia implementations currently used by a number of BitTorrent clients.

Since many BitTorrent clients already implement their own DHT, it is natural to wonder whether Arpeggio can use the same DHT, eliminating the need for clients to run two separate DHTs. However, it is not straightforward to do so because Arpeggio's indexing system requires the nodes to support network-side processing features such as index-side filtering. One option for integrating it with an existing DHT would be to use a protocol for subgroup lookup such as Diminished Chord [38], or OpenDHT's ReDiR service. With Diminished Chord's "lookup key in subgroup" primitive, which efficiently finds the immediate successor of a particular key among the set of nodes in a subgroup, a single DHT could be used for routing even if only some nodes are participating in our indexing system.

Though it violates the peer-to-peer nature of the system, it is possible to configure the user interface tools to contact a remote `arpd` daemon rather than one running locally. This could be useful if some users are unable to run their own indexing daemon.

In order for users to begin using Arpeggio, it must provide an improvement over existing services. As currently implemented, it provides largely the same functionality as existing BitTorrent websites. Its advantage is that it operates cooperatively, and hence can scale better and respond better to failures than a centralized solution. However, there is a network effect involved: if no users are registering files with the system, then no new users will have any incentive to use it for their searches. To aid in deployment, the index can be bootstrapped using data from existing indexes of files. For the purposes of evaluating our system, we developed a tool (Section 6.1.2) for periodically scanning the top BitTorrent search websites for new torrents; this could be adapted to register the torrents it discovers with the Arpeggio index, creating an initial index. Such a tool could be run by any user of the network, and would only need to be run by a single user. As users begin to use the system, the index registration plugins in their BitTorrent clients would register the files they are sharing, eventually eliminating the need for this measure.

Chapter 6

Evaluation

6.1 Corpora

In order to evaluate the effectiveness of Arpeggio for various distributed indexing applications, we use data from several sources.

6.1.1 FreeDB CD Database

FreeDB [21] is a database of audio CD metadata, based on the old CDDB service. It contains the artist, album title, and track titles for many CDs, using user-submitted data. It is commonly used in audio players to identify a CD and display title information. Though DHTs may be useful for this application [59], we do not propose Arpeggio as a replacement for FreeDB, we are merely using its database as an example corpus of the type of information that could be indexed by Arpeggio. An Arpeggio index would allow searches for songs whose title, artist, or album include various keywords.

This data is well-suited for Arpeggio's indexing because it consists of many files which have large (audio) content and only a few metadata keywords. For our evaluation, we obtained the complete FreeDB database, and extracted the keyword information. The database contains over 2 million discs, with a total of over 28 million songs. Each song has an average of 6.43 metadata keywords. For comparison, the actual data corresponding to these songs would make up approximately 214 years of audio, requiring about slightly over a petabyte of storage in CD format.

6.1.2 BitTorrent Search

A natural application for Arpeggio is in indexing shared files (torrents) in BitTorrent. Since BitTorrent itself does not include an indexing mechanism, the task of locating shared files falls on various index web sites, which presents scalability and reliability challenges. The DHT-based trackerless system addresses the related problem of finding sources for a particular file, but not searching for files by keyword. Arpeggio can be used for the latter purpose.

To evaluate Arpeggio's feasibility for indexing torrents, we obtained metadata for all torrents that were added to several torrent index sites over one week. This was done using an automated client that downloaded lists of torrents from the sites listed in Table 6.1, and extracted the file name and other metadata from each entry. The sites indexed included most of the top 10 most popular torrent sites (some did not publish a list of latest torrents,

Table 6.1: *BitTorrent search sites indexed*

Site	Content indexed
`http://isohunt.com/`	25 latest submitted torrents
`http://isohunt.com/`	25 latest indexed torrents
`http://www.bittorrent.com/`	10 most popular torrents
`http://www.mininova.org/`	All torrents added today
`http://thepiratebay.org/`	30 latest added torrents
`http://thepiratebay.org/`	100 most popular torrents
`http://www.bitenova.nl/`	50 latest added torrents
`http://www.torrentspy.com/`	All torrents added today
`http://www.torrentz.com/`	50 latest added torrents
`http://www.btjunkie.org/`	240 latest added torrents
`http://www.btjunkie.org/`	30 most popular torrents
`http://www.torrentreactor.net/`	All torrents added today
`http://www.torrentreactor.net/`	20 most popular torrents
`http://www.meganova.org/`	All torrents added today
`http://www.torrentportal.com/`	All torrents added today

so were not suitable for indexing). Over a period of 7 days, the sites were checked for new additions at least once every three hours, in order to obtain all torrents that were added during the sample period.

6.1.3 Gnutella

In terms of intended use, Arpeggio is quite similar to Gnutella, since both can be used for fully-decentralized peer-to-peer file sharing networks. Hence, we evaluated Arpeggio using a Gnutella-based workload.

Gnutella workloads have been measured and studied extensively [62, 76, 66, 28, 39], and, indeed, we use results from these studies to determine our user model in our analysis. However, the effectiveness of Arpeggio's indexing algorithms depends heavily on the nature of the metadata of the files, in particular the number of keywords per file and per query. Since no prior studies report on these properties, we conducted our own measurement study of Gnutella to obtain a sample of query strings and file names.

We conducted a measurement study using an instrumented version of the popular LimeWire Gnutella client [44][1]. The purpose of the study was solely to obtain information about file names and queries; we did not attempt to measure other properties, such as node lifetime, user behavior, etc. In order to do so, we modified the client to record query requests and results that passed through it, and added a programmatic interface for our measurement scripts to inject queries. The client was also modified to always act as an ultrapeer (supernode), which meant that it was able to observe both queries received from its client leaves as well as those routed from other ultrapeers in the network. Monitoring these queries gave a sample of the queries that users were performing. Our client reissued these queries (at a rate of approximately one query per second), causing it to receive lists of

[1]Though one would not expect the particular Gnutella client used to be particular important, incompatibilities between clients made it important to use a popular, well-supported client. In particular, our earlier attempt to index Gnutella using an instrumented version of Mutella [50] was foiled because LimeWire clients, which make up much of the network, were unable to connect to it.

all hosts with matching files. Whenever a query hit was received from a previously-unseen host, the indexer contacted that host directly with a "Browse-Host" request, prompting it to return all files it was sharing. These files were, in turn, used to generate additional queries: approximately once per second, a file was randomly selected and its keywords were set as a query, in order to discover and browse other hosts that had that file available.

Performing this sampling process for 24 hours, our client observed 2,661,219 queries[2] and 10,869,055 query results (including replies to both its queries and its browse requests). Because the queries and hosts are selected uniformly at random, we expect the results to be representative of Gnutella file names and queries as a whole. It is not, however, without bias: older clients that do not support the Browse-Host protocol are excluded from consideration, as are newer clients whose users have specifically disabled the feature. However, these clients should be rare, as the most two common Gnutella clients, LimeWire and BearShare, which comprise over 90% of the network [66], both support this feature and enable it by default. Note that hosts behind firewalls and NATs do not pose a problem, because our indexer uses the standard Gnutella hole-punching mechanisms in order to send its requests.

6.2 Index Properties

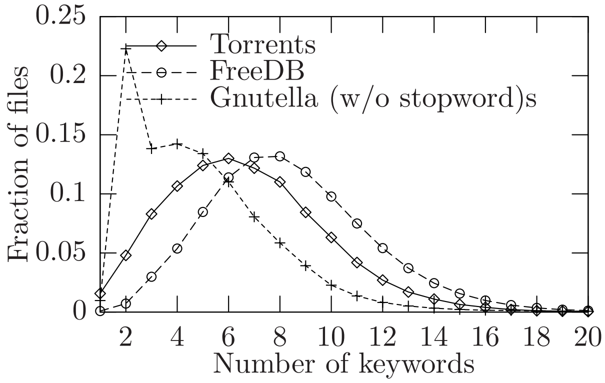
Since Arpeggio's algorithms rely upon the assumption that the amount of metadata associated with each file is small, it is important to investigate the number of metadata keywords associated with each file in the three corpora. Figure 6-1 shows the number of files in each corpus that have a certain number of keywords. Notably, the average number of keywords is higher for FreeDB: 8.40 for FreeDB vs. 6.72 for torrents and 5.33 for Gnutella. This difference occurs because each song in FreeDB has an artist, album title, and song title associated with it, whereas the other corpora include only metadata taken from file names. Also, each corpus includes a small number of files with many metadata keywords; these can have a disproportionately large effect on the total index size, particularly for larger values of the keyword-set size parameter K.

In order to reduce the number of keywords per file, we can exclude *stopwords*: common words that are not included in the index because they occur so frequently that they are not useful in searches. This technique is used frequently in search systems. The obvious downside of excluding stopwords is that queries containing only stopwords (such as "the") cannot return any replies; a more subtle point is that removing stopwords from queries may decrease the effectiveness of keyword-set indexing if, for example, only one keyword remains. For our purposes, we use a list of stopwords containing the standard common English words, words of two or fewer characters, and common file type extensions[3]. We present many of the following results both with and without stopwords excluded.

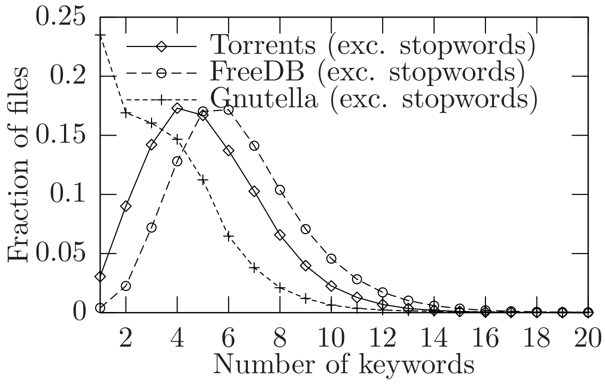
Figure 6-2 shows the distribution of number of keywords per Gnutella query (query information is unfortunately not available for the other two proposed applications). Most Gnutella queries include many keywords; the average query contained 5.33 keywords, and less than 1% contained only one. Even after discarding stopwords, the average query length remained over 3.72 keywords, with only about 20% contained only one keyword. In comparison, Reynolds and Vahdat observed that for web search engines, the average query length

[2]Most of which cannot be reprinted for reasons of decency.

[3]Users may wish to search for files of a given file type. Even with stopwords, Arpeggio still provides this capability by using index-side filtering to restrict the returned search results to files of the correct type.



(a) *No stopwords*



(b) *Excluding stopwords*

Figure 6-1: *Distribution of number of keywords per file, with and without stopwords*

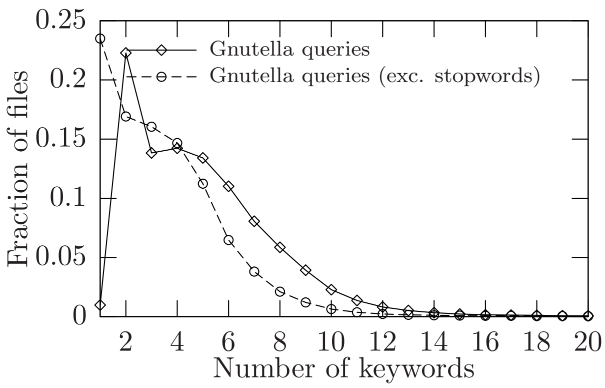


Figure 6-2: *Distribution of number of keywords per Gnutella query*

was 2.53 keywords, with 28% containing only one keyword, making them considerably less specific than our Gnutella queries [58].

6.2.1 Choosing K

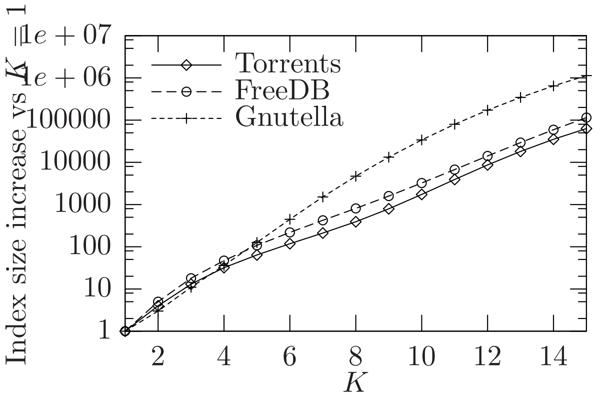
The effectiveness and feasibility of Arpeggio's indexing system depend heavily on the chosen value of the maximum subset size parameter K. Recall that K is the upper bound on keyword subset size that will be indexed; i.e. setting $K = 1$ creates a simple inverted index, $K = 2$ also includes keyword pairs, $K = 3$ also includes triplets, etc. If K is too small, then the KSS technique will not be as effective: there will not be enough multiple-keyword indexes to handle most queries, making long indexes necessary for result quality. If K is too large, then the number of index entries required grows exponentially, resulting in infeasible storage requirements and maintenance bandwidth requirements. Most of these index entries will be in many-keyword indexes that will be used only rarely, if at all, so little benefit is achieved.

The optimum value for the parameter K depends on the application, since both the number of metadata keywords for each object and the number of search terms per query vary. The Gnutella query trace showed an average of 3-5 keywords per query, depending on whether stopwords are included or excluded, which suggests that value can be obtained from increasing K to these amounts, though our experiments below show that the incremental value of increasing K above 3 is small.

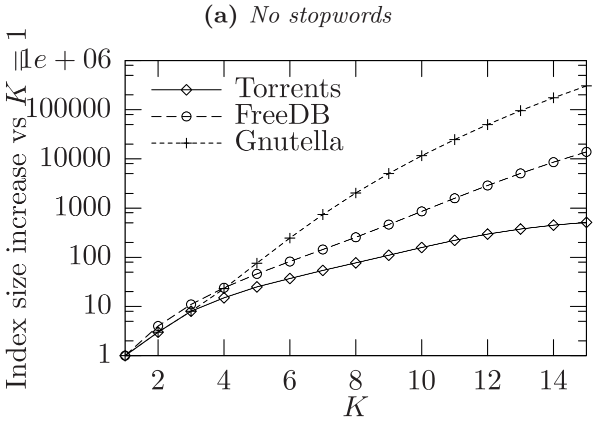
In the following sections, we analyze the costs required to build an Arpeggio index for varying values of K, as well as the benefits achieved in terms of improved load distribution.

6.3 Indexing Cost

We now consider the effects of keyword-set indexing on cost. There are two factors that we must consider: the increase in total storage costs that result from creating keyword set indexes, and the improvement in query load distribution that results from being able to send queries to more specific indexes.



(a) *No stopwords*



(b) *Excluding stopwords*

Figure 6-3: *Index size increase vs.* $K = 1$, *with and without stopwords*

6.3.1 Index Size

In order to accurately determine the storage costs imposed by Arpeggio's use of keyword-set indexes, we analytically computed the storage that would be required to index each of our three corpora. This was done by computing the number of keywords for each file, optionally excluding stopwords, and computing the number of index entries required for that file using the formula for $I(m, K)$ in Section 3.2.

The results are shown in Figure 6-3. The graph shows the total number of index entries relative to the $K = 1$ case, on a logarithmic scale. Recall that the $K = 1$ case corresponds to the standard inverted index case, so the graph shows the extra cost required to maintain keyword-set indexes. From this, we see that our proposed value of $K = 3$ or $K = 4$ is feasible, as it results in only an increase of one order of magnitude in index size. Note also that, even though the overall index size has increased, its distribution is improved because there are a larger number of keyword indexes that can be stored on different nodes.

6.3.2 Query Load

Next, we consider the benefits obtained from the use of keyword-set indexing. We will evaluate how the distribution of query load improves: whether the number of nodes that receive a disproportionately high number of queries is reduced. To do this, we simulate the Gnutella workload. We begin by creating 1000 Chord nodes; each operates 8 virtual nodes

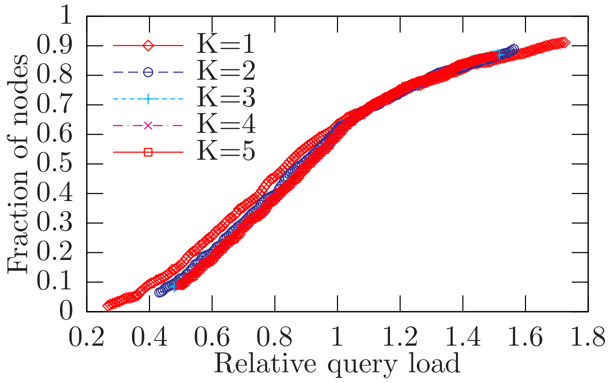


Figure 6-4: *Query load distribution for various K*

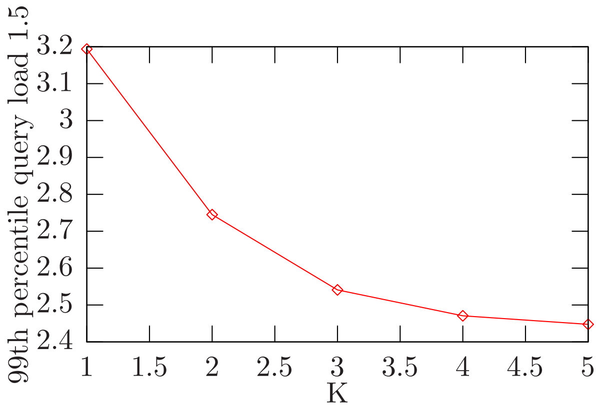


Figure 6-5: *99th-percentile query load vs. K*

with randomly-assigned hash IDs to improve load balancing. We then consider every query in our trace of 2.6 million Gnutella queries, compute its keywords, and select an appropriate keyword set index. The query load is computed in terms of number of queries.

The resulting distribution is shown as a CDF in Figure 6-4. The query load on the horizontal axis is normalized such that a value of 1 represents the average load (i.e. total queries/# nodes). An ideal distribution would be a steep curve around 1. We can see that the $K = 1$ case is significantly less balanced than the others, the $K = 2$ case improves somewhat, and the $K \geq 3$ cases are not easily distinguishable from each other. Perhaps more comprehensibly, Figure 6-5 shows that the 99th-percentile query load decreases as a result of increasing the value of K.

6.4 Maintenance Cost

Though index size is an important consideration for evaluating the feasibility of Arpeggio, it is an incomplete measure without also considering the costs of maintaining the index. Indeed, the principal bottleneck likely to be encountered is not storage space, but *bandwidth* requirements. These include the costs of satisfying INSERT and QUERY requests, replicating the index, and running the Chord protocol. To this end, we performed a packet-level simulation of the Arpeggio protocol under load, and evaluated the bandwidth usage.

Parameter		Client Class 1	Client Class 2
class probability	p_i	0.25	0.75
query idle time	$\lambda^{(i)}_{\text{idle}}$	30 s	300 s
departure probability	$p^{(i)}_{\text{leave}}$	0.1	0.2
shares files?		no	yes

Table 6.2: *Client model parameters*

6.4.1 Simulation Model

Our simulation was performed using the `p2psim` discrete-event simulator for peer-to-peer protocols [25]. We produced a simulator implementation of Arpeggio for `p2psim` that shares all the major features of the full-fledged implementation with the exception of a simplified metadata model; the simulator version handles only metadata derived from file names, and does not support additional fields for rich metadata. It includes KSS, index-side filtering, index gateways, and replication. The lookup layer is simulated using `p2psim`'s implementation of Chord.

Because nodes in peer-to-peer networks are highly heterogeneous with respect to their stability and capacity, our model takes this heterogeneity into account. We use a two-level structure inspired by Gnutella's ultrapeers: the core of the network is made up of a stable set of nodes that form a Chord ring and operate Arpeggio index servers, and a set of leaf nodes send requests to the core nodes but do not maintain indexes themselves. We assume that each core node supports 10 leaf nodes. This is consistent with Gnutella, where the fraction of ultrapeer nodes is in the 10–15% range [66]. The most common Gnutella client connects to three ultrapeers, each of which allow 30 leaf node connections; we neglect this leaf multihoming for simplicity. We use an exponentially-distributed core node lifetime of 106 minutes, following Loo et al's observation that this is the average lifetime of Gnutella nodes after excluding those with lifetimes less than 10 minutes [45]. For connectivity between core nodes, we use `p2psim`'s King network topology, which consists of an all-pairs matrix of latencies between DNS servers measured using the King tool [29].

Each of the ten leaf nodes connected to each leaf node is simulated as follows, using the model of [23]: the leaf is randomly assigned a class i (with probability p_i, which determines its behavior. Once connected, the node begins sending queries with an exponentially-distributed query inter-arrival time with mean $\lambda^{(i)}_{\text{idle}}$. Queries are randomly selected from our Gnutella query trace. After each query, the client has probability $p^{(i)}_{\text{leave}}$ of going offline. If so, the time until a new client arrives is exponentially-distributed with mean λ_{join}. After remaining online for five minutes, a client will begin to register its files' metadata with the index at a rate of one file per second.

Our model uses two client classes. Based on Gnutella measurements [62], 25% of clients are freeloaders (only downloading files, not sharing), and 75% share files. The parameters for these two classes are shown in Table 6.2. The mean time for new client arrival λ_{join} was set to 300 seconds. For non-freeloader peers, the number of files was chosen using a logarithmic distribution similar to that described in [62]. The files were selected using a Zipf popularity distribution, with keywords for each file randomly selected from our Gnutella trace.

Arpeggio was configured with two replicas per index, with $K = 3$, and performed replica

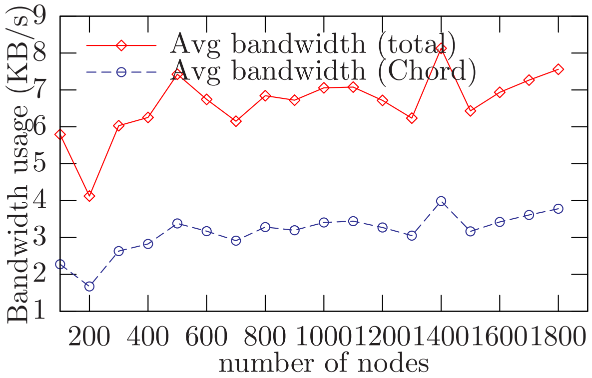


Figure 6-6: *Simulation results: average per-node bandwidth*

synchronization every five minutes.

6.4.2 Results

We ran the simulator for varying numbers of nodes, and simulated for 720 seconds each time. The graph of per-node bandwidth usage is shown in Figure 6-6. Note that the simulator is operating in a two-tier configuration with 1 core node per 10 leaf nodes; the number of clients shown is the number of leaf nodes, and the number of core nodes is $1/10^{\text{th}}$ that amount. Only bandwidth between core nodes is counted; we are neglecting the bandwidth between leaf nodes and their core node for this experiment. The subset of the bandwidth that is used for Chord lookups and routing table maintenance is shown separately to distinguish it from the costs of Arpeggio's indexing. The data is noisy due to the degree of randomness inherent in the client behavior in this experiment, but shows that the maintenance costs scale well, remaining approximately the same even as the number of index nodes and clients are increased by a factor of more than 10.

Chapter 7

Conclusion

We have introduced the Arpeggio content sharing system, which provides both a metadata indexing system and a content distribution system. Arpeggio differs from other peer-to-peer networks in that both aspects are implemented using a DHT-based lookup algorithm, allowing it to be both fully decentralized and scalable. We extend the standard DHT interface to support not only lookup-by-key but search-by-keyword queries. In order to make distributed indexing scale to the size of a peer-to-peer file sharing network, we introduce network-side processing techniques such as index-side filtering, index gateways, and expiration. For metadata indexing, where the average number of keywords per file is small, we improve query load balancing by indexing based on keyword sets rather than individual keywords.

For the content-distribution side of the system, we provide two options. We introduce a new content-distribution system based on indirect storage via Chord subrings that uses chunking to leverage file similarity, and thereby optimize availability and transfer speed. It further enhances availability by using postfetching, which uses cache space on other peers to replicate rare but demanded files. Alternatively, for ease of deployment, we present a simpler alternative that uses the trackerless BitTorrent system to handle much of the content distribution tasks.

Arpeggio has been implemented, providing a core indexing module that can be used with a variety of user interfaces. Currently available are a web interface that allows convenient searches for BitTorrent files, and a plugin for a popular BitTorrent client that automatically registers shared files with the distributed index.

Our simulation studies evaluate the feasibility of using Arpeggio to index files from Gnutella, various BitTorrent search sites, and the FreeDB CD database. We find that keyword-set indexing improves query load balancing, and Arpeggio is able to perform scalably under a Gnutella-based synthetic workload.

Bibliography

[1] ACM. *The 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[2] Azureus: Java BitTorrent client. Available at `http://azureus.sourceforge.net/`.

[3] Luiz Andre Barroso, Jeffrey Dean, and Urs Hoelzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, March 2003.

[4] BitTorrent protocol specification. `http://bittorrent.com/protocol.html`.

[5] BitTorrent software. Available at `http://www.bittorrent.com/download.html`.

[6] BitTorrent Community Forum. BitTorrent trackerless DHT protocol specifications v1.0. Experimental draft. Available at `http://www.bittorrent.org/Draft_DHT_protocol.html`.

[7] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[8] CacheLogic Ltd. Peer-to-peer in 2005, August 2005. Available at `http://www.cachelogic.com/home/pages/research/p2p2005.php`.

[9] Miguel Castro, Manuel Costa, and Antony Rowstron. Debunking some myths about structured and unstructured overlays. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)* [70].

[10] Josh Cates. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, May 2003.

[11] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making Gnutella-like P2P systems scalable. In *Proceedings of ACM SIGCOMM 2003*, Karlsruhe, Germany, August 2003. ACM.

[12] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)* [71].

[13] Austin T. Clements, Dan R. K. Ports, and David R. Karger. Arpeggio: Metadata searching and content sharing with Chord. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS '05)*, volume 3640 of *Lecture Notes in Computer Science*, pages 58–68, Ithaca, NY, USA, February 2005. Springer.

[14] Bram Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003.

[15] Drew Cullen. SuprNova.org ends, not with a bang but a whimper. *The Register*, December 19, 2004. Available at `http://www.theregister.co.uk/2004/12/19/suprnova_stops_torrents/`.

[16] 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)* [1].

[17] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)* [69].

[18] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the 11th ACM/SIAM Symposium on Discrete Algorithms (SODA '00)*, pages 743–752, San Francisco, CA, USA, January 2000. ACM/SIAM.

[19] John R. Douceur. The Sybil attack. In IPTPS '02 [33].

[20] Abe Fettig. *Twisted Network Programming Essentials*. O'Reilly Media, Inc., Sebastopol, CA, USA, 2006.

[21] FreeDB. Available at `http://www.freedb.org/`.

[22] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)* [69].

[23] Zihui Ge, Daniel R. Figueiredo, Sharad Jaiswal, Jim Kurose, and Don Towsley. Modeling peer-peer file sharing systems. In *Proceedings of IEEE INFOCOM 2003*, San Francisco, CA, USA, March 2003. IEEE.

[24] The giFT project: Internet file transfer. Available at `http://gift.sourceforge.net/`.

[25] Thomer Gil, Frans Kaashoek, Jinyang Li, Robert Morris, and Jeremy Stribling. p2psim: a simulator for peer-to-peer protocols. Available at `http://pdos.csail.mit.edu/p2psim/`.

[26] Omprakash D. Gnawali. A keyword set search system for peer-to-peer networks. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, June 2002.

[27] Gnutella protocol specification 0.4. `http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf`.

[28] Krishna P. 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 *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, USA, October 2003. ACM.

[29] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proceedings of the 2nd ACM SIGCOMM Internet Measurement Workshop*, Marseille, France, November 2002. ACM / USENIX.

[30] Gnutella web caching system. Available at `http://www.gnucleus.com/gwebcache/`.

[31] Tom Hargreaves. The FastTrack protocol. Available at `http://gnunet.org/papers/FAST-TRACK-PROTOCOL`, August 2003.

[32] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, M. Satyanarayanan, Gregory R. Ganger, and Erik Riedel. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04)*, San Francisco, CA, USA, March 2004. USENIX.

[33] *The 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, USA, February 2002.

[34] *The 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003.

[35] *The 3rd International Workshop on Peer-to-Peer Systems (IPTPS '04)*, San Diego, CA, USA, February 2004.

[36] M. Frans Kaashoek and David R. Karger. Koorde: a simple degree-optimal distributed hash table. In IPTPS '03 [34].

[37] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC '97)*, El Paso, TX, USA, May 1998. ACM.

[38] David R. Karger and Matthias Ruhl. Diminished Chord: A protocol for heterogeneous subgroup formation in peer-to-peer networks. In IPTPS '04 [35].

[39] Alexander Klemm, Christoph Lindemann, Mary K. Vernon, and Oliver P. Waldhorst. Characterizing the query behavior in peer-to-peer file sharing systems. In *Proceedings of the 4th Internet Measurement Conference (IMC '04)*, Taormina, Sicily, Italy, October 2004. ACM / USENIX.

[40] KTorrent: BitTorrent client for KDE. Available at `http://ktorrent.org/`.

[41] Jinyang Li, Boon Thau Loo, Joseph M. Hellerstein, M. Frans Kaashoek, David Karger, and Robert Morris. On the feasibility of peer-to-peer web indexing and search. In IPTPS '03 [34].

[42] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)* [70].

[43] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC '02)*, Monterey, CA, USA, July 2002. ACM.

[44] Lime Wire LLC. LimeWire: Open source P2P file sharing. Available at http://www.limewire.org/.

[45] Boon Thau Loo, Ryan Huebsch, Ion Stoica, and Joseph M. Hellerstein. The case for a hybrid P2P search infrastructure. In IPTPS '04 [35].

[46] MAGNET-URI draft specification. Available at http://magnet-uri.sourceforge.net/magnet-draft-overview.txt.

[47] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In IPTPS '02 [33].

[48] David Mazières. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, USA, June 2001. USENIX.

[49] Ralph C. Merkle. *Secrecy, authentication, and public key systems*. PhD thesis, Stanford University, Stanford, CA, USA, June 1979.

[50] Mutella. Available at http://mutella.sourceforge.net/.

[51] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* [1].

[52] Napster. http://www.napster.com/.

[53] National Institute of Standards and Technology. Secure hash standard. Federal Information Processing Standard 180-2, August 2002.

[54] Overnet. Available at http://web.archive.org/web/20060601060922/http://www.overnet.com/.

[55] Peer exchange. Available at http://www.azureuswiki.com/index.php/Peer_Exchange.

[56] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, April 1989.

[57] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM 2001*, San Diego, CA, USA, August 2001. ACM.

[58] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of the International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

[59] 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 ACM SIGCOMM 2005*, Philadelphia, PA, USA, August 2005. ACM.

[60] Christopher Rohrs. Query routing for the Gnutella network. Technical report, LimeWire LLC, New York, NY, USA, May 2002. Available at http://www.limewire.com/developer/query_routing/keyword%20routing.htm.

[61] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001. IFIP/ACM.

[62] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the 13th Multimedia Computing and Networking Conference (MMCN '02)*, San Jose, CA, USA, June 2002. SPIE/IS&T.

[63] Emil Sit and Robert Morris. Security concerns for peer-to-peer distributed hash tables. In IPTPS '02 [33].

[64] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):149–160, February 2003.

[65] Jeremy Stribling, Jinyang Li, Isaac G. Councill, M. Frans Kaashoek, and Robert Morris. OverCite: A distributed, cooperative CiteSeer. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)* [71].

[66] Daniel Stutzbach, Reza Rejaie, and Subhabrata Sen. Characterizing unstructured overlay topologies in modern P2P file-sharing systems. In *Proceedings of the 5th Internet Measurement Conference (IMC '05)*, Berkeley, CA, USA, October 2005. ACM / USENIX.

[67] Chunqiang Tang and Sandhya Dwarkadas. Hybrid global-local indexing for efficient peer-to-peer information retrieval. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)* [69].

[68] Twisted Matrix Labs. *Introduction to Perspective Broker*. Available at `http://twistedmatrix.com/projects/core/documentation/howto/pb-intro.htm%l`.

[69] USENIX. *The 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, USA, March 2004.

[70] USENIX. *The 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, USA, April 2005.

[71] USENIX. *The 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, CA, USA, May 2006.

[72] Ashlee Vance. MPAA closes Loki. *The Register*, February 10, 2005. Available at `http://www.theregister.co.uk/2005/02/10/loki_down_mpaa/`.

[73] Limin Wang, KyoungSoo Park, Ruoming Pang, Vivek S. Pai, and Larry Peterson. Reliability and security in the CoDeeN content distribution network. In *Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, June 2004. USENIX.

[74] Hakim Weatherspoon and John D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In IPTPS '02 [33].

[75] Dave Winer. XML-RPC specification. Available at `http://www.xmlrpc.com/spec`, June 1999.

[76] Shanyu Zhao, Daniel Stutzbach, and Reza Rejaie. Characterizing files in the modern Gnutella network: A measurement study. In *Proceedings of the 13th Multimedia Computing and Networking Conference (MMCN '06)*, San Jose, CA, USA, June 2006. SPIE/IS&T.