Arpeggio: Metadata Searching and Content Sharing with Chord

Austin T. Clements, Dan R. K. Ports, David R. Karger *
MIT Computer Science and Artificial Intelligence Laboratory
{aclements, drkp, karger}@mit.edu

January 21, 2005

Abstract

Arpeggio *is a peer-to-peer file-sharing network based on the Chord lookup primitive. Queries for data whose metadata matches a certain criterion are performed efficiently by using a* distributed keyword-set index*, augmented with index-side filtering. We introduce* index gateways*, a technique for minimizing index maintenance overhead. Because file data is large,* Arpeggio *employs subrings to track live source peers without the cost of inserting the data itself into the network. Finally, we introduce* postfetching*, a technique that uses 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, and a content distribution system tuned to the requirements and capabilities of a peer-to-peer network.*

1 Overview and Related Work

Peer-to-peer file sharing systems, which let users locate and obtain files shared by other users, have many advantages: they operate more efficiently than the traditional client-server model by utilizing peers' upload bandwidth, and can be implemented without a central server. However, many current file sharing systems trade-off scalability for correctness, resulting in systems that scale well but sacrifice completeness of search results or vice-versa.

Distributed hash tables have become a standard for constructing peer-to-peer systems because they overcome the difficulties of quickly and correctly locating peers. However, the *lookup by name* DHT operation is not immediately sufficient to perform complex *search by content* queries of the data stored in the network. It is not clear how to perform searches without sacrificing scalability or query completeness. Indeed, the obvious approaches to distributed full-text document search scale poorly [8].

In this paper, however, we consider systems, such as file sharing, that search only over a relatively small amount of *metadata* associated with each file, but that have to support highly dynamic and unstable network topology, content, and sources. The relative sparsity of per-document information in such systems allows for techniques that do not apply in general document search. We present the design for *Arpeggio*, which uses the LOOKUP primitive of Chord [13] to support metadata search and file distribution. This design retains many advantages of a central index, such as completeness and speed of queries, while providing the scalability and other benefits of full decentralization. *Arpeggio* resolves queries with a constant number of Chord lookups. The system can consistently locate even rare files scattered throughout the network, thereby achieving near-perfect recall.

In addition to the search process, we consider the process of distributing content to those who want it, using subrings [7] to optimize distribution. Instead of using a DHT-like approach of storing content data directly in the network on peers that may not have originated the data, we use indirect storage in which the original data remains on the originating nodes, and small pointers to this data are managed in a DHT-like fashion. As in traditional file-sharing networks, files may only be intermittently available. We propose an

*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.

architecture for resolving this problem by recording in the DHT requests for temporarily unavailable files, then actively increasing their future availability.

Like most file-sharing systems, *Arpeggio* includes two subsystems concerned with searching and with transferring content. Section 2 examines the problem of building and querying distributed keyword-set indexes. Section 3 examines how the indexes are maintained once they have been built. Section 4 turns to how the topology can be leveraged to improve the transfer and availability of files. Finally, Section 5 reviews the novel features of this design.

2 Searching

A content-sharing system 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. Each file shared on the network has an associated set of metadata: the file name, its format, 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, ID3 tags on MP3 music files.

Analysis based on required communications costs suggests that peer-to-peer keyword indexing of the Web is infeasible because of the size of the data set [8]. However, peer-to-peer indexing for metadata remains feasible, because the size of metadata is expected to be only a few keywords, much smaller than the full text of an average Web page.

2.1 Background

Structured overlay networks based on distributed hash tables show promise for simultaneously achieving the recall advantages of a centralized index and the scalability and resiliency attributes of decentralization. Distributed hash location services such as Chord [13] provide an efficient LOOKUP primitive that maps a key to the node responsible for its value. Chord uses at most $O(\log n)$ messages per lookup in an n-machine network, and minimal overhead for routing table maintenance. Building on this primitive, DHash [3] and other distributed hash tables provide a standard GET-BLOCK/PUT-BLOCK hash table abstraction. However, this interface alone is insufficient for efficient keyword-based search.

2.2 Distributed Indexing

A reasonable starting point is a *distributed inverted index*. In this scheme, the DHT maps each keyword to a list of all files whose metadata contains that keyword. To execute a query, a node performs a GET-BLOCK operation for each of the query keywords and intersects the resulting lists. The principal disadvantage is that the keyword index lists can become prohibitively long, particularly for very popular keywords, so retrieving the entire list may generate tremendous network traffic.

Performance of a keyword-based distributed inverted index can be improved by performing *index-side filtering* instead of joining at the querying node. 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 is similar to the search algorithm used by the Overnet network [11], which uses the Kademlia DHT [9]. Note that index-side filtering breaks the standard DHT GET-BLOCK abstraction by adding network-side processing, demonstrating the utility of direct use of the LOOKUP primitive.

2.3 Keyword-Set Indexing

While filtering reduces network usage, query load may be unfairly distributed, overloading nodes responsible for popular keywords. To overcome this problem, we propose to build inverted indexes not only on keywords but also on keyword *sets*. 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 [5].

Essentially, this scheme allows us to precompute the full-index answer to all queries of up to K keywords. For queries of more than K keywords, the index for a randomly chosen K-keyword subset of the query can be filtered. 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.

Since the majority of searches contain multiple keywords [12], large indexes are no longer critical to result quality as most queries will be handled by smaller, more specific indexes. 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.

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. Furthermore, if m is small compared to K (as for files with few keywords), then $I(m)$ is no worse than exponential in m. Further analysis will be necessary to identify the optimum value for the parameter K[1]; however, the average number of terms for web searches is approximately 2.53 [12], so K is likely to be around 3, and thus $I(m)$ will be a low-degree polynomial in m.

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 can further constrain a search. Index-side filtering allows for more complex searches than KSS alone. A user may only be interested in 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.

[1] We are currently investigating the effectiveness of methods for splitting indexes into more specific indexes only when necessary (essentially, adapting K per index).

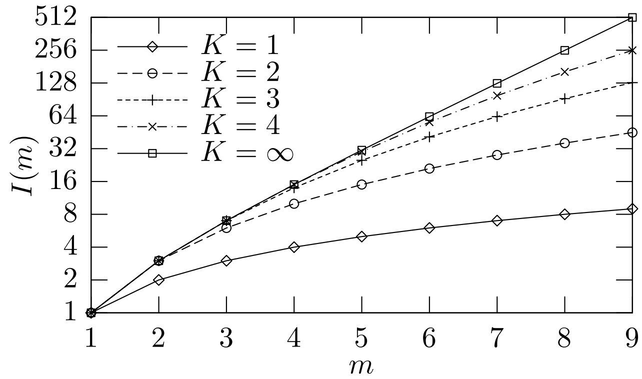

Figure 1: Growth of $I(m)$ for various K

3 Index Maintenance

Peers are constantly joining and leaving 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.

3.1 Metadata Expiration

Instead of polling for departures, or expecting nodes to notify us of them, we expire metadata on a regular basis so that long-absent files will not be returned by a search. Nevertheless, blocks 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, we *refresh* metadata that is still valid, thereby periodically resetting its expiration counter. We argue in Section 4.3 that there is 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.

3.2 Index Gateways

If each node directly maintains its own files' metadata in the distributed index, 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. The total cost for inserting the file is therefore $\Theta(sI(m))$ 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 must repeatedly renewing its insertions to prevent their expiration.

To minimize this redundancy, we introduce an *index gateway* node that aggregates index insertion. Index gateways are not required for correct index operation, but they increase the efficiency of index insertion. 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 LOOKUP of the block's hash). The gateway then inserts the metadata block into all of the appropriate indexes, but *only* if necessary. If the block already exists in the network and is not scheduled to expire soon, then there is no need to re-insert it into the network. 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 $\Theta(sI(m))$ to $\Theta(s + I(m))$.

3.3 Index Replication

In order to maintain the index despite node failure, index replication is also necessary. Because metadata blocks are small and reading from indexes must be low-latency, replication is used instead of erasure coding [3]. 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.

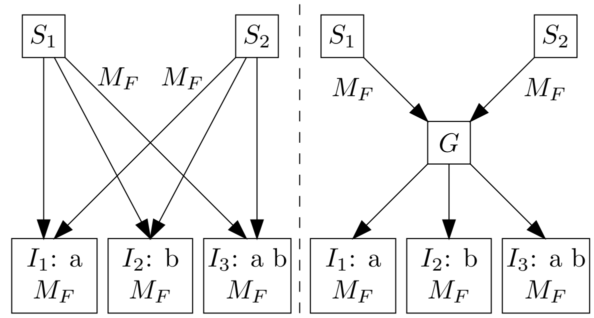

Figure 2: Two source nodes $S_{1,2}$, inserting file metadata block M_F to three index nodes $I_{1,2,3}$, with (right) and without (left) a gateway node G

4 Content Distribution

The indexing system we describe above simply provides the ability to search for files that match certain criteria. It is independent of the file transfer mechanism. Thus, it is possible to use an existing content distribution network in conjunction with *Arpeggio*. A simple implementation might simply store a HTTP URL for the file in the metadata blocks, or a pointer into a content distribution network such as Coral [4]. A DHT can be used for direct storage of file contents, as in distributed storage systems like CFS [2]. For a file sharing network, direct storage is impractical because the amount of churn [6] and the content size create high maintenance costs.

Instead, *Arpeggio* uses *indirect storage*: it maintains 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

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

Table 1: Layers of lookup indirection

highly dynamic network topology, but may sacrifice content availability.

4.1 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, *Arpeggio* 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 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 one with the same version, if necessary.

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 [10]. 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, *Arpeggio*'s segmentation algorithm additionally promotes sharing of data *between files*.

4.2 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 [1]. However, the node responsible for tracking the peers sharing a popular chunk that represents a single point of failure may become overloaded.

We instead use *subrings* to identify sources for each chunk, distributing the query load throughout the network. The Diminished Chord protocol [7] 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 Postfetching

To increase the availability of files, *Arpeggio* can cache 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.

5 Conclusion

We have presented the key features of the *Arpeggio* content sharing system. *Arpeggio* differs from previous peer-to-peer file sharing systems in that it implements both a metadata indexing system and a content distribution system using distributed hash tables. We extend the standard DHT interface to support not only lookup by key but complex search queries. Keyword-set indexing and extensive network-side processing in the form of index-side filtering, index gateways, and expiration are used to address the scalability problems inherent in distributed document indexing. We introduce a content-distribution system based on indirect storage via subrings that uses chunking to leverage file similarity, and thereby optimize availability and transfer speed. Availability is further enhanced with postfetching, which uses cache space on other peers to replicate rare but demanded files.

References

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

[2] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. SOSP '01*, Oct. 2001.

[3] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proc. NSDI '04*, Mar. 2004.

[4] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. NSDI '04*, Mar. 2004.

[5] O. Gnawali. A keyword set search system for peer-to-peer networks, June 2002. Master's thesis, Massachusetts Institute of Technology.

[6] K. P. Gummadi, R. J. Dunn, S. Sariou, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. SOSP '03*, Oct. 2003.

[7] D. R. Karger and M. Ruhl. Diminished Chord: A protocol for heterogeneous subgroup formation in peer-to-peer networks. In *Proc. IPTPS '04*, Feb. 2004.

[8] J. Li, B. T. Loo, J. M. Hellerstein, M. F. Kaashoek, D. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *Proc. IPTPS '03*, Feb. 2003.

[9] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. IPTPS '02*, Mar. 2002.

[10] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proc. SOSP '01*, Oct. 2001.

[11] Overnet. `http://www.overnet.com/`.

[12] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proc. Middleware '03*, June 2003.

[13] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.