Arpeggio:
Efficient Metadata-based Searching and File Transfer with DHTs

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

November 1, 2004

Abstract

Arpeggio *is a peer-to-peer file-sharing network based on the Chord distributed hash table. Queries for files whose metadata matches a certain criterion are performed efficiently by using a distributed keyword-set index, augmented with index-side filtering. We introduce* metadata gateways*, a technique for minimizing index maintenance overhead.* Arpeggio *also uses the DHT for indirect storage of file contents, maintaining pointers from content to the live peers that provide it. Finally, we introduce* post-fetching*, 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 of a peer-to-peer file-sharing network.*

1 Overview and Related Work

Recent years have seen a dramatic increase in the popularity of peer-to-peer file sharing systems. These 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 either 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 networks because they overcome the difficulties of quickly and correctly locating peers. However, the *lookup by name* primitives provided by DHTs are not immediately sufficient to perform complex *search by content* queries of the data stored in the network. It is not clear how to construct search interfaces without sacrificing scalability or query completeness. Indeed, when it comes to systems that search for documents, Li et al. [5] have shown that the obvious approaches to distributed document search do not scale well.

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. 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 primitives of the Chord distributed hash table [11] to support metadata search and file distribution. This design retains many of the advantages of a central index, such as completeness and speed of queries, while providing the scalability and other benefits of full decentralization. Arpeggio uses only a constant amount of space for each distinct file being indexed, and resolves any query with a constant number of Chord lookups. The system can consistently locate even rare files scattered throughout the network, thereby achieving perfect recall.

In addition to the search process, we consider the process of transferring found files to those who want them. We use the DHT to optimize content distribution. Rather than using it for direct storage, we use a layer of indirection: the DHT tracks which peers are sharing which content. As in traditional file-sharing networks, files may only be intermittently available. We propose an 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 files. Section 2 examines the problem of building and querying distributed search indexes. Section 3 examines how the indexes are maintained once they have been built. Section 4 turns to how the DHT can be leveraged to improve the transfer and availability of files. Finally, Section 5 reviews the novel features of this design.

2 Searching

A file-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. Specific types of files can have more descriptive metadata; for example, the song title and artist name can be extracted from MP3 music files via the ID3 tags.

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 [5]. However, peer-to-peer indexing for file sharing networks remains feasible, because searches are performed using only file metadata, not contents. The size of file metadata is expected to be on the order of a few keywords, much smaller than the text of an average Web page.

2.1 Background

A natural approach is to maintain a centralized index of the files in the network, as done by the Napster [8] file sharing network. 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 [11] provide a LOOKUP primitive that maps a key to the node responsible for its value. It does so efficiently, with 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 [2] and other distributed hash tables provide a standard GET-BLOCK/PUT-BLOCK hash table abstraction. However, this interface alone does not allow search based on metadata keywords.

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, thus distributing the index throughout the network. 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 (e.g. "the"), so retrieving the entire list will 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 (unlike in the general case of indexing entire documents), the entire contents of the metadata can be kept in the index with each item. To perform a query involving a metadata term, we can send the full query to the corresponding index node, and it can perform the filtering and return only relevant results. This dramatically reduces network traffic at query time, since only results relevant to the full query need to be transmitted. This is similar to the search algorithm used by the Overnet network [9], which uses the Kademlia DHT [6]. Note that index-side filtering extends the DHT abstraction beyond the simple GET-BLOCK request: the DHT is no longer a mere data structure; it now supports network-side processing.

Although index-side filtering reduces query bandwidth, machines responsible for the large inverted indexes of common search terms will have disproportionately large storage and query loads. As an initial fix for this problem, we *truncate* large inverted indices, keeping only those items that best-match the inverted index term. This limits storage load, but sacrifices correctness: multi-term queries directed to this index may have matches in the full list that are not in the truncated list, hiding some correct results.

To overcome this problem, we propose to build inverted indexes not only on keywords but also on keyword *sets*. For each file, an index entry is created for 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 [4]. 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, truncation may still be necessary. However, it seems likely that in most cases there will be a "rare" k-word subset of the query with a small inverted index. This index will suffer little or no truncation, and will thus be more likely to hold all relevant results.

Using keyword set indexes rather than keyword indexes increases the number of index entries for a file with m metadata keywords from $O(m)$ to $O\left(\min(m^k, 2^m)\right)$. However, it reduces the importance of single-keyword indexes: they will be used only to answer single-word queries, and the negative effects of truncation on these queries are minimized. Reynolds and Vahdat observed that less than 29% of web search queries included only one keyword [10]. Further analysis will be necessary to identify the optimum value for the parameter k; however, the average number of terms for web searches is approximately 2.53, so k is likely to be around 3 [10]. A major advantage of keyword-set indexing is that the query traffic to nodes responsible for popular keywords will be abated by queries going to more specific indexes.

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 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.

2.3 Index Structure and Algorithms

We now describe in detail the index representation used by *Arpeggio*. All index data is stored in a distributed hash table similar to DHash [2]. However, in addition to the standard GET-BLOCK and PUT-BLOCK hash table abstractions, it must also support a FILTERED-GET operation that returns all list entries that match certain criteria. Everything *Arpeggio* stores in the DHT is in the form of a block with a tag indicating the type of block.

A *metadata block* contains the keyword and filterable metadata for a file as well as information for obtaining the file contents. Each unique file has a corresponding metadata block that holds its metadata. When a node registers a file with the index (for instance, when it comes online), it inserts a copy of the file's metadata block under each of the file's keyword sets. The location information can take multiple forms, since our indexing system is independent of the content distribution system. For the content distribution system we describe in Section 4, the location is the block ID of the file block.

To perform a query, a node first selects some largest possible subset K (up to size k) of the keyword metadata in the query. It then sends a FILTERED-GET request with the query as parameter to the node responsible for the metadata blocks associated with the keyword-set K. The contacted node searches its index to find the entries that match the full query, and returns them to the querying node.

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 under such circumstance.

3.1 Metadata Expiration

Instead of polling for departures, or expecting nodes to notify us of them, we expire metadata on a regular basis. This guarantees that long-absent files will never be returned by a search. 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. Pointers to file blocks, on the other hand, are guaranteed to remain valid in the face of network topology changes because the DHT handles replication of blocks. To counteract expiration, we *refresh* metadata that is still valid, thereby periodically resetting its expiration counter.

While one might think that metadata should be expired frequently so that searches are up to date, we argue in Section 4.2 that there is actually great value in keeping old metadata around to match queries, even if the corresponding files are no longer available. In particular, we track attempts to access those missing files and *postfetch* them to other nodes in the DHT if they become available after the access attempt. This lets us identify files that are in demand but sporadically available, and replicate them to increase their availability.

3.2 Metadata Gateways

Having each node directly maintain its own files' metadata in the network will result in large numbers of redundant inserts and refreshes of metadata blocks representing very common files. Insertion to many distinct keyword search sets is relatively expensive, so this will also consume a large amount of bandwidth. The problem can be solved by introducing a *metadata gateway* node to the process of insertion.

A metadata gateway aggregates updates to metadata blocks. Gateways leverage the fact that individual nodes insert many copies of the same block and, for common files, many different nodes will also insert the same block. To use a metadata gateway, instead of directly inserting metadata blocks into the network, a peer sends a single copy of the block to the gateway responsible for the block (found via the block's hash). The gateway then inserts the metadata block under all of the appropriate keys in the network, 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 if the blocks in the network are going to expire soon, but the copy of the block held by the gateway has been more recently refreshed.

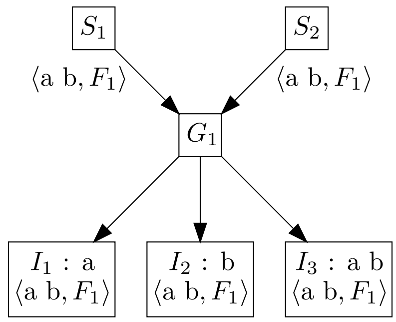

Figure 1: Two file source nodes inserting the same file F_1 via the $\langle \text{a b}, F_1 \rangle$ metadata gateway

Using gateways, insertion by content-holding nodes is no more costly than for a centralized index system, though the gateways themselves do consume bandwidth to insert into the index. Furthermore, gateways reduce the total network bandwidth incurred by managing metadata blocks from being proportional to the total number of files to being proportional to the number of unique files.

3.3 Index Replication

In order to maintain the index despite node failure, metadata block replication is also necessary. Because metadata blocks are small and reading from them must be low-latency, replication is used instead of erasure coding [2]. 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 directly 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 because metadata blocks will expire on all nodes simultaneously.

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. Alternatively, a content distribution network such as Coral [3] could be used. A DHT can be used for direct storage of file contents, such as in distributed storage systems like CFS [1]. However, because of the large file sizes and amount of churn in a typical file sharing network, *Arpeggio* uses *indirect storage*: the DHT stores pointers to each peer that contains a certain file.

A peer can query this index to identify all other peers that are sharing a file it wishes to obtain. The downloader may begin transferring the file from multiple sources in order to distribute the load and maximize transfer speed. Our algorithm is designed to maximize this effect.

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 pointers to *chunk blocks*, which in turn contain a list of the peers sharing each chunk, as in Figure 2. File and chunk block IDs are derived from the hash of the file or chunk 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. Second, multiple files may contain the same chunk, as Figure 2 demonstrates. A peer can obtain part of a file from peers that do not have an exactly identical file, but merely a *similar* file.

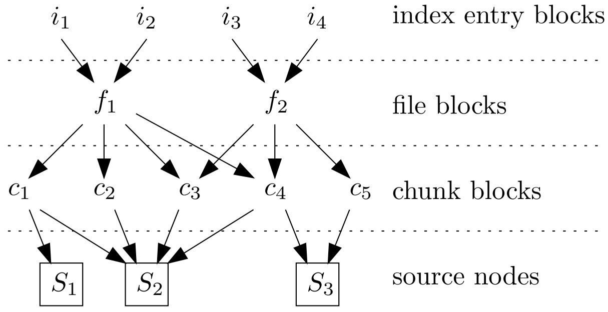

Figure 2: Division of files into chunks

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. 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 ID3 tag.

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. More generally, insertion or deletion of file content may negate 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 [7]. 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 file will still be largely comprised of identical chunks.

4.2 Postfetching

Additional novel properties can be garnered from *Arpeggio* by introducing caching into the network. Caching file chunks on nodes that would not usually be sharing the chunks provides new opportunities for increasing availability of files. Cached chunks are inserted into the index just like regular chunks, and therefore do not share the disadvantages of direct storage with regards to having to maintain the chunks despite topology changes. Furthermore, this insertion symmetry makes caching transparent to the search system. Also, 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 other nodes. These in turn register as sources for those chunks, thus 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* file 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, metadata gateways, and expiration are used to ameliorate the scalability problems inherent in distributed document indexing. We introduce a content-distribution system based on indirect DHT storage 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] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. SOSP '01*, October 2001.

[2] 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*, March 2004.

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

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

[5] 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*.

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

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

[8] Napster. `http://www.napster.com/`.

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

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

[11] 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.