Optimizing and Balancing Load in Fully Distributed P2P File Sharing Systems

Anh-Tuan Gai, Laurent Viennot

Optimizing and Balancing Load in Fully Distributed P2P File Sharing Systems

(Scalable and Efficient Keyword Searching)

Anh-Tuan Gai
INRIA Rocquencourt
anh-tuan.gai@inria.fr

Laurent Viennot
INRIA Rocquencourt
laurent.viennot@inria.fr

Abstract

A peer-to-peer file sharing system includes a lookup and a content distribution protocol. Very efficient peer-to-peer protocols exist for content distribution, but efficient indexing is still an open problem. Numerous work on structured overlay networks such as distributed hash tables offer a promising framework. However, balancing the load of publishing, storing indexes and answering request still remains a challenging task. We sketch a general architecture framework for solving these problems in the case of a file sharing application. Our design goals include reducing the work of file providers (they should not bare all the publishing process) and enabling keyword searching based on the assumption that few words are associated with each file.

1 Introduction

Peer-to-peer file sharing system [9, 4, 6, 3] has become an attractive alternative to client-server content distribution. Instead of uploading data from a unique source, the upload cost is redistributed among peers. Before downloading, the first problem to solve while searching a data is to identify precisely the data we are interested in and then to find peers sharing it. In this paper we are particularly concerned with fully distributed file sharing systems. In such environments, the participants give some of its resources in exchange for using the service. They would thus naturally expect a workload proportional to their use of the service. In particular, we cannot expect some of the peers to bear the load of a dedicated server.

Unfortunately, conventional lookup protocols are inherently not well matched to a fully distributed environment. In flooding based lookup protocols, the workload is not optimized since the systems may not return addresses of peers sharing a file until the whole network has been explored. In routing based solutions, such as distributed hash tables (DHT), few messages are exchanged but some nodes responsible for widely spread words may have to store many associations and to answer many requests. This conflicts with the expectation that all members should share the workload.

Distribution systems such as eDonkey [4] release on powerful servers. If some node play the role of these powerful servers this conflicts with the expectation that all member should share the tracking load.

We introduce a general framework which enables efficient keyword searching. We propose solutions to optimize the associations storage load and distribute it among participating nodes. Moreover, we distribute among interested nodes the association lists upload cost.

The key idea is to combine a resilient DHT and an efficient distribution protocol such as BitTorrent [3] both for lookup and distribution. We use the distribution protocol to redistribute the load of large association lists. We use the lookup protocol to find trackers that help users downloading the same file meeting each other. The key challenge is to efficiently redistribute the lookup workload, and the workload of tracking a file, among peers sharing interest.

The rest of this paper is organized as follows. A brief description of BitTorrent, and a recall on distributed hash tables (DHT) and bloom filters is given in Section 2. We introduce solutions to redistribute and optimize the publishing load, the storage load and the request load in Sections 3, 4 and 5 . Section 6 describes related work and Section 7 concludes.

2 Background

2.1 Distributed Hash Tables

Distributed hash tables (DHTs) are a class of decentralized distributed systems which partition ownership of a set of keys among participating nodes; they can efficiently route messages to owners of any given key. Each node is responsible for a part of a hash table. DHTs are typically designed to scale to a large number of nodes and to handle continual node arrivals and failures. This infrastructure can be used to build more complex services, such as distributed file systems, peer-to-peer file sharing systems, cooperative web caching, multicast, anycast, and domain name services. Many DHTs have been recently proposed. Table 1 represents the different characteristics of some DHTs. All DHTs usually follow the same framework: nodes maintain neighborhood links for safe routing and local republishing. Long links enable small graph diameter. The k nodes with IDs close to the file or word ID u for a given metric are responsible for u associations.

protocol	topology	diameter	contacts	k
Chord [16]	hypercube	$O(\log n)$	$O(\log n)$	1
Pastry [14]	hypercube	$O(\log n)$	$O(\log n)$	1
De-Bruijn [12, 10, 7, 1]	De Bruijn graph	$O(\log n)$	$O(1)$	1
Kademlia [11]	hypercube	$O(\log n)$	$O(k \log n)$	k
Broose [8]	De Bruijn	$O(\log n)$	$O(k)$	k

Table 1. characteristics of some DHTs. k is the redundancy parameter, it represents the number of contacts a peer can choose at each lookup step.

2.2 BitTorrent

BitTorrent [3] has quickly become one of the most used peer-to-peer file-sharing system in terms of traffic. When a file is made available using HTTP, all upload cost is placed on the hosting machine. With BitTorrent, when multiple people are downloading the same file at the same time, they upload pieces of the file to each other. This redistributes the cost of upload to downloaders, thus making hosting a file with a potentially unlimited number of downloaders affordable. BitTorrent is different from other peer-to-peer systems because it does not include any lookup protocol. Downloaders find ".torrent" files on websites using any search engine like Google or dedicated websites. ".torrent" files are small metadata files containing information about length, name, hashing information and tracker of a given file. Trackers are responsible for helping the torrent downloaders finding each other (files downloaded via BitTorrent are called torrents). The latest version of BitTorrent also includes a DHT in case of tracker failure. Since peers downloading a file have already retrieved the unique file ID from the ".torrent", a node responsible for the file ID may replace the tracker.

2.3 Bloom filters

A bloom filter [2] is a hashed-based data structure that summarizes membership in a set. More precisely, a bloom filter is an array of bits initially all equal to 0. To encode a set, each element is hashed by the same set of functions. When an element is hashed, the result correspond to a bit position of the bloom filter which is changed to 1. The membership test consist in hashing with the same hash functions set an element. If all resulting bit are equal to 1 in the bloom filter the test is positive. Nevertheless, the membership test returns false positives with a tunable, predictable probability and never forgets true members. Given optimal choice of hash functions number, the probability of a false positive is $p_{fp} = .6185^{s/e}$ where s is Bloom filter's size in bits and e is the number of elements in the set. Thus, to maintain a fixed probability of false positives, the size of the Bloom filter must be proportional to the number of elements represented.

2.4 Associations

In all peer-to-peer file sharing systems, some information about files and peers are distributed among participating nodes. These informations are necessary to enable efficient keyword searching. For a given keyword all systems should return files containing the keyword in the file name or in the file descriptors (author, album, or year for example). Then for a given file all systems should return addresses of peers sharing and downloading it. We thus distinguish two associations classes: keyword and file. A keyword association associates a keyword (or it's ID) to a file ID. A file association associates a file ID to addresses of peers downloading the file. Furthermore, we propose that both associations contain the file name and the descriptor list (see Sections 3 and 4 for more details).

3 Publishing Load

Less than 10 percents of peers share the majority of files available in file sharing systems and 70 percents of peers do not share files [15]. Moreover, the few percents of generous nodes share more than 1000 files each. In conventional routing based systems, a node publishing a file performs a lookup for each keyword of each file it publishes. A node sharing files will thus performs $F * D$ lookups, where F is the number of file it shares and d the average number of keywords per file. Ideally, a peer-to-peer file sharing system should give incentive to share files and should redistribute the publishing load among peers. To redistribute the publishing load, we now introduce the 2-levels publishing scheme.

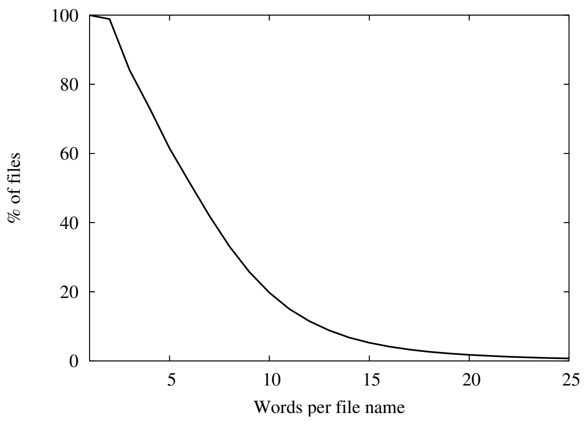

Figure 1. distribution of number of words per file name

3.1 2-levels publishing scheme

Similarly to associations classes, we distinguish two publication classes: file and keyword. Like in conventional publishing scheme, in the 2-levels publishing scheme a node sharing a file is responsible for the file association publication (file info associated to the sharing node address). On the other hand, to redistribute the keyword association publishing load, the nodes responsible for the file association are responsible for keyword associations publications (file info associated to the file ID). A node sharing a file thus needs only one lookup to publish it and keyword publications cost is redistributed over the system since the files IDs are themselves balanced. Figure 1 illustrates the distribution of words contained in the file name over one million of distinct eDonkey files. The average number of words per file is 5, and less than 1 percent of the files contain more than 25 words.

Using 2-levels publishing scheme also optimizes the publishing traffic. If a file has been already published, nodes responsible for the file ID do not republish keyword associations each time a node publishes an association for this file. Notice that the distribution of copies per file may be unbalanced and thus nodes responsible for a spread file ID will significatively reduce their keyword publishing load. In the whole system, the number of lookups for keyword publications is thus divided by C, where C is the average number of copies per file in the system.

Furthermore, to optimize the republishing load, a node sharing a file does not republish the file association, it is regularly contacted by nodes responsible for the file ID. It thus replies to a ping instead of performing a lookup. Similarly, nodes responsible for keyword associations regularly contact nodes responsible for the file ID. Finally, if the node sharing a file (resp. responsible for the file ID) is not contacted by nodes responsible for the file association (resp. keyword association) after a given timeout period, it republishes the file association (resp. keyword association). Using 2-levels publishing scheme together with pings, the $F * C * D$ lookups in conventional system per republishing time period, are replaced by $F * C + F * D$ pings, where F is the number of distinct files and D is the average number of keyword per file in the system.

4 Associations Storage Load

Key collisions introduced by widely spread keywords (”mp3” for example) requires some balancing mechanisms. Otherwise, a node responsible for a widely spread keyword will be overloaded in storage capacity (key collision hotspot) and in bandwidth capacity (request hotspot). Figure 2 illustrates the keyword occurrences distribution over two millions eDonkey files.

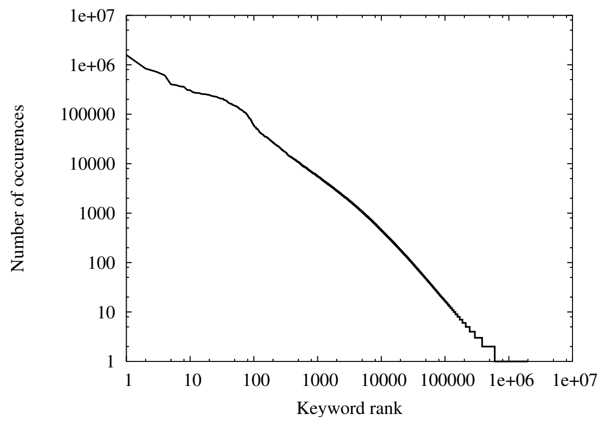

Figure 2. keywords ranked by number of occurences

In conventional routing based systems, associations are stored on the k closest nodes to a given ID. Let l be the maximum number of associations a node stores for a given keyword. If a keyword appears in more than l associations, we would like to redistribute the storage load among participating nodes. The main challenge is to find a rule to ensure that each association is stored by $O(k)$ nodes. Ideally, we also would like these nodes to be close (in number of hops) to the original ID.

4.1 Mixed Hashing

Similarly to double hashing, in case of key collisions, the system must provides an alternative ID to store associations. The key idea is to store keyword associations of spread keywords on peers close to a mix of most of the keyword ID digits and some of the file ID digits. If the nodes responsible for a spread keyword ID v already store l associations, further associations will be stored on nodes responsible for ID u composed with one digit of the associated file ID and digits of the keyword ID. Then, if the nodes responsible for u also store more than l associations for v, further associations will be stored on nodes close to ID composed with two digits of the associated file and digits of the keyword ID, and so on... If the mixed ID is well constructed, retrieving nodes storing associations consists in pushing further ahead the lookup (instead of performing another lookup). Moreover, using bi-directional protocols such as Broose [8], nodes storing associations may be on the routing path to the word ID (in Broose, associations should be stored on nodes close in number of hops to the keyword ID using right-shifting and lookups for request should use left-shifting[1]). Notice that finding associations on the routing path may be useful to redistribute the request load (see section 5 for more details).

4.2 Smaller Keyword Associations

Another challenge is to optimize association size. If a keyword association is reduced to the couple $\{$word ID, file ID$\}$, it's size is typically 320 bits (2 SHA-1 160 bits identifiers). To obtain smaller list, the file ID can be reduced to the first 40 or 50 bits. In a system with less than 2^{30} users, the k closest node to the real file ID is are not the k closest node to the ID composed with the first 50 file ID bits completed with 0, with probability less than $\frac{k}{2^{50-30}}$. Association storage for keyword u is reduced to $(160 + 50\ m)$ bits instead of $(160 + 160\ m)$ bits, where m is the number of associations for u. When m is large, the keyword association storage can thus be reduced by a factor 3.2.

5 Request load

Some association lists of spread keywords contain more than one million associations (see Figure 2). Using mixed hashing introduced in Section 4 redistributes the storage load of large associations list. Nevertheless, nodes responsible for frequently requested list will be overloaded in download by requests and in upload by association list distribution. Similarly, nodes responsible for meta-info files (".torrent") distribution will be overloaded if the associated file is frequently requested or if the ".torrent" is large. The main breakthrough of BitTorrent is to redistribute the file upload cost among peers sharing or downloading a file. We propose to use a protocol similar to BitTorrent to redistribute the association list and the meta-info file (".torrent") load. Moreover, we optimize the size of association list returned to save network bandwidth.

5.1 Request Hotspot

We propose to use a solution similar to the the solution introduced in Kademlia [11] to alleviate request hotspots. To redistribute the load of receiving request, answers should be cached on nodes likely to be on the path of further requests for the same ID such that nodes close to the requested ID are not overloaded by receiving request. To avoid over caching, associations are cached during a period of time exponentially inversely proportional to the number of hops between the current node and the node responsible for the key.

A complementary solution consists in distributing all these informations for frequently requested files or keywords associations list by all peers in the system or all peers sharing same interest (publish-subscribe) using a protocol similar to BitTorrent. We propose to introduce a global torrent of very requested file and keyword IDs. When a node is overloaded for a given file or keyword ID, it adds it to the global torrent and shares a new bloc containing informations it is frequently requested for.

5.2 Associations list Upload Cost

We would like to redistribute a large requested list upload cost among peers downloading it by sharing it. If many nodes want the same large association list they should retrieve it in a cooperative way and thus use a distribution protocol such as BitTorrent. When a peer looks for files relevant for a given keyword, it performs a lookup over the keyword ID. If the nodes responsible for the keyword ID are not overloaded, they act as trackers and as seeds for the keyword association list. If the nodes responsible for a given keyword u are overloaded, nodes storing associations for u act as trackers for the whole list and as seeds for a bloc of the association list. Furthermore, nodes on the routing path to u cache addresses of peers interested by the association list and thus also act as tracker. However, to enhance connectivity and to redistribute the tracking load, peers downloading the association list exchange their contact list.

Furthermore, If a keyword is widely spread and frequently asked by a peer, this peer should store the corresponding association list and regularly update it. Adding the first publication date to an association enables a peer to ask only for association published after it's last update. Moreover, since this peer has already many associations it will act as seed for old associations.

[1]In De Bruijn lookup based protocols, the contact list of node u contains nodes with prefix ID obtained by shifting u prefix address and inserting new digits.

5.3 Smaller Association Lists Uploaded

In this section we introduce solutions to optimize the traffic generated by the association list upload cost. If word associations only contain a word ID associated to a file ID, a node has to retrieve the association list for each words appearing in the request before it can choose a file in the lists intersection. Since bandwidth is the most expensive resource (compared to storage), we propose to add the filename and the file info (author, byterate, year, etc...) to the keyword association. When a node is looking for files relevant for several keywords, it only performs one lookup for one of the keyword (preferentially the rarest). Nodes responsible for the word ID only return associations where all requested keywords appear. The global torrent described in Section 5.1, may be useful to learn which words are widely spread.

Including file descriptions induces both storage and bandwidth overheads for keyword associations. We now introduce another solution to optimize the size of keyword association. For each file, a Bloom filter over its words is computed. Recall a Bloom filter is a hashed-based data structure that summarizes membership in a set. This Bloom filter replace the filename and the file infos of word associations. With a 20 bits per keyword Bloom filter the probability of false positive is less than 10^{-4} instead of 80 bits for a 10 characters keyword. This can reduce the keyword list size by a factor approximately 4.

Reynolds and Vahdat [13] have shown that more than 70% of requests contain two or more keywords. Another solution to reduce the size of associations list returned to requests could be the following. Nodes responsible for a file ID, also publish associations of keyword couple $\{$keyword ID, keyword ID, file ID$\}$. The storage load would also be $O(F * D^2)$ but with $O(F * D^2)$ publish lookups instead of $O(F * D)$ with the previous solution proposed.

5.4 Hashes Distribution

To download a file with BitTorrent the client needs the ".torrent" file. Some recent studies [5] have shown that the download is faster and fairer if the file is divided into small chunks (to enhance exchanges). Having many chunks implies having many hashes and thus a big ".torrent" file. In centralized systems, powerful users upload the ".torrent" files from servers.

To redistribute the upload cost of the ".torrent" to downloaders, we introduce the ".metatorrent" file (".torrent" of the ".torrent"). The trackers (nodes responsible for the file ID) store the ".metatorrent" file. A ".metatorrent" is simply a ".torrent" of the ".torrent". It contains all information necessary to start downloading the ".torrent" and the file. More precisely it contains the ID of the desired file, the size of the ".torrent", the size of the file, the size of chunks, and the hashes of the ".torrent" chunks. To start downloading the file quickly, BitTorrent verification procedure should be modified. A client can download file chunks even if it does not have the hashes of these chunks. With the size of the file and the size of a file chunk contained in the ".metatorrent" the node can already ask for and download some file chunks. The file chunks verifications with hashes are performed when the ".torrent" downloaded has been verified with the hashes contained in the ".metatorrent".

6 Related Works

Many routing based peer-to-peer systems have been proposed recently, e.g. [16, 14, 12, 10, 7, 1, 11, 8, 3].

Several systems based on DHT enable to find the closest nodes to a given ID [16, 14, 12, 10, 7, 1, 11, 8]. Chord [16], Pastry [14] and first generation De Bruijn graph based DHT [12, 10, 7, 1] need a strict topology maintenance. This implies many messages exchanges and low resilience to node departures. Kademlia [11] and Broose [8], are more resilient to node failure and churn but do not resolve the key collision hotspots.

BitTorrent [3] offers an efficient solution to file distribution but the content lifetime and availability rely on powerful servers. Moreover, in the solution presented the upload cost of the ".torrent" is redistributed among peers downloading the file.

A solution using Bloom filters over files membership [13] is efficient when keywords appear in few associations. If keywords may appear in 10000 associations, the optimal size of Bloom filters is 10 KB. Figure 2 illustrates that some words like "mp3" have much more occurrences. Furthermore, in the solution introduced in [13], nodes responsible for requested associations exchange messages to limit false positive associations returned. Thus, nodes responsible for many associations answer a lot of requests on the one hand and exchange messages to limit false positive on the other hand. However, we introduce a solution to optimize associations lists. Bloom filters computed over descriptor considerably reduce the descriptor association size and the upload associations lists cost can be distributed on peers interested in retrieving the list.

7 Conclusion

We have introduced a general framework for fully distributed file sharing system. The multi-level publication scheme introduced significatively reduces the publication traffic since keyword associations are published only once. Using pings instead of lookups reduces the republishing load by a factor $O(\log n)$. The mixed hashing scheme allows to redistribute the storage load on nodes close (in number of hops) to nodes responsible for the original ID. The solutions provided enable efficient keyword searching without any flooding mechanism and support inherent unbalanced associations distribution due to spread words. Using Bloom filters over keywords and smaller file ID allow to reduce the size of keyword association lists. Moreover, we are able to distribute the query load for spread keywords on peers interested by the corresponding keyword association lists. Finally, the solutions provided are designed for distributed hash tables but could be adapted to skip graph or other lookup protocols.

References

[1] I. Abraham, B. Awerbuck, Y. Azar, Y. Bartal, D. Malkhi, and E. Pavlov. A generic scheme for building overlay networks in adversial scenarios. In *17th International Parallel and Distributed Processing Symposium (IPDPS'2003)*, april 2003. Nice.

[2] A. Broder and M. Mitzenmacher. Networks applications of bloom filters: A survey. In *Allerton Conference*, 2002.

[3] Bram Cohen. Incentives build robustness in bittorrent, 2003.

[4] http://www.edonkey2000.com/.

[5] P. Felber and E.W. Biersack. Self-scaling networks for content distribution. In *International Workshop on Self-* Properties in Complex Information System (Self-*)*, May-June 2004. Bertinoro.

[6] F. Le Fessant and S. Patarin. Mldonkey, a multi-network peer-to-peer file-sharing program. Technical Report RR4797, INRIA, April 2003.

[7] P. Fraigniaud and P. Gauron. The content-addressable network d2b. Technical Report LRI 1349, Univ. Paris-Sud, 2003.

[8] A.T. Gai and L. Viennot. Broose: A practical distributed hashtable based on the de-bruijn topology. In *International Conference on Peer-to-Peer Computing (P2P)*, August 2004. Zurich.

[9] http://www.gnutella.com/.

[10] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[11] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[12] M. Naor and U. Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures (SPAA)*, 2003.

[13] Patrick Reynolds and Amin Vadhat. Efficient peer-to-peer keyword searching. In *Proceedings of the ACM/IFIP/USENIX Middleware conference*, June 2003.

[14] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, LNCS 2218, pages 329–350, 2001.

[15] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[16] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.