A Peer–to–Peer Inverted Index Implementation For Word-Based Content Search

Nuno Alberto Ferreira Lopes

October, 2003

1 Introduction

For the last few years Peer-to-Peer (P2P) systems have been used for cooperative file sharing among users. Napster was the first system that allowed the exchange of music files through the collaboration of it's peers. P2P systems make an effective use of the widespread availability of computing and storage resources that are present over the Internet, even though temporarily. The ability to search information is of great importance in order to take full advantage of such huge amount of shared information.

The first generation of P2P systems, like Napster [4] and Gnutella [3], uses a word based searching model. Users search for files that match a given word or list of words. Napster makes use of a central index for searching, despite file transfers being made only between peers. A centralized index creates a single point of failure on the system, which can be used to shutdown the system. Gnutella was the first system to use a completely decentralized networking model. Searching is made through neighborhood broadcasts, which create a considerable amount of network traffic. Therefore, this system does not scale to a very large number of users [6].

The second generation of P2P systems, like Chord [9] and Pastry [7], is based on the Distributed Hash Table (DHT) model. A DHT system forms an overlay network where each peer is given an unique identifier. The identifier has constant size and may be generated as the result of an hashing function. This identifier is also used as the key of the $(key, value)$ pair each peer will store. The distributed hash table is formed by all the pairs contained on peers. The overlay network is capable of locating any key of the hash table, and therefore the peer storing it, from any other peer in a scalable way. The routing procedure required for locating

keys makes use of very few communication and storage resources for each peer. Although these systems are very efficient on resource usage, they are not capable of searching information contents explicitly. The information that is stored as values on the DHT can only be accessed by a key, following a typical hash table functionality.

There are two basic models for object searching on a P2P network: local or global. The local model requires that the search must be sent to every other peer on the network. The final search result is the union of the individual peers result. As such, there is a tremendous overhead on the communication channels because each query requires a broadcast to all peers. This is the case for the Gnutella system. Since every query goes through every peer, that information could be used to passively learn the location of specific objects and, later on, use it to redirect queries [5]. Nevertheless, this technique would imply a startup period where each peer would learn the object distribution across the network and because peers are connected to the network for small periods of time in average [8], the system might not be able to stabilize due to constant peer renewal.

The global model assumes that a search is performed on some global shared index. Since the system cannot have any centralized structure, this index would have to be distributed evenly across the peers. To build such an index, each peer is required to announce it's contents, in the form $word \mapsto location$. The location points to a document stored on the peer that contains the given word. The inverted index is made by collecting all the announcements. Searching for a word on the index returns the list of all document locations that match that word.

However, the word occurrence distribution must be taken into account when using an inverted index. It is known that word presence in documents follows a Zipf distribution [2, 1]. This distribution forms an index with a wide frequency range. Words that are very popular have high occurrence rates and make only a very small part of the word set on the index. Most of the words on the index have in average a very low occurrence rate. The figure 1 shows the relation between words and the number of documents in which they occur.

Due to the inverted index requirements, a DHT system would be unsuitable for implementing an index directly. The inverted index requires that the word location image adapts to different sizes ranging from a single location entry to a number of entries close to the document count. DHT systems are not designed to support key images (values) that can get extremely large while being at the same constantly modified. In fact, DHTs offer a storage that is limited to the capacity of the peer for each key. It would be possible to split an oversized image into smaller pieces that could be stored on several peers. However, the management overhead resulting from a constant activity of changing the image contents, by adding and removing locations from it, would make the DHT inefficient.

This article discusses the feasibility of applying a complex data structure, B^+-

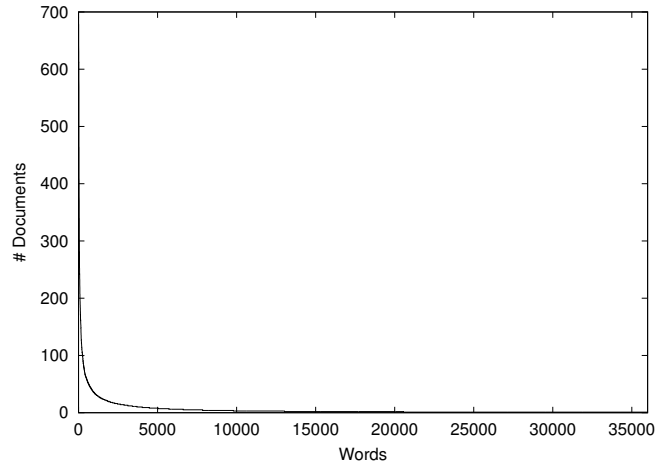

Figure 1: Number of documents associated to each word

tree, on top of a DHT to achieve a decentralized scalable inverted index implementation. Although DHTs are not appropriate for an index implementation, their scalability and efficient resource usage can still be used as a base platform for peer location. On the other hand, B^+-trees provide an efficient implementation of hash tables over secondary storage devices. The B^+-tree objective is to minimize I/O disk operations. The same idea is also present on network communications, where the objective is to reduce the number of messages transmitted between hosts.

2 The System Model

An inverted index is a mapping between words and document location sets ($word \mapsto \{document\ location\}_{SET}$). It's main purpose is to locate documents that contain a specific word. With a word-based location service available, it is possible to do contents search on shared documents. The index provides the following major operations:

- INSERT($word$,$reference$) – inserts a new document reference under the word set.
- REMOVE($word$,$reference$) – removes a reference the word set.
- HAS_REF($word$,$reference$):$bool$ – checks if a reference is present under a word.

- GET_REF($word$):$reference$ – retrieves the first reference of the word location set.
- NEXT_REF($word$,$reference$):$reference$ – iterates over the location set of a word.

Internally, the system makes use of the DHT as a key/peer locater that stores ($key \mapsto value$) data pairs. The keys have a constant size that results from applying an $hash$ function. This $hash$ function also distributes the keys uniformly over the key universe. Values associated to keys also have a fixed size and are treated as atomic objects. The DHT provides the following basic operations:

- INSERT(key_h,$value$) – associate a value to a key.
- GET(key_h):$value$ – retrieve the value associated to the key.

The B^+-tree is used to manage the set of locations for each word. A set implementation is obtained by using just the key field of the B^+-tree's internal structure (which would represent a single document location) and ignoring the $value$ field. Since any B-tree was designed to offer efficient insert and query operations with a large number of elements, it is specifically suited for an inverted index set implementation.

Each word association on the inverted index has it's own unique B^+-tree. As a consequence, any operation on an index word must retrieve the root block from the associated B^+-tree. From there on, the tree algorithm will select the other blocks necessary to the operation, if any. In order to access any block, the tree algorithm needs some kind of "block reference". Since all blocks of all trees are stored on the DHT, the DHT key is used as the block reference. The block's DHT key is created by taking the word to which the tree belongs and some other related block's information, in order to minimize the chance of collisions.

Popular index words will be subject of constant access for both location searching and inserting. Because every index operation must always access the root block of a word's tree, the peer responsible for such block will suffer from contention. To reduce contention, a block caching procedure is executed. The procedure caches exclusively blocks that are not leafs. B^+-tree operations start by modifying a leaf block first and only then cascade into upper level blocks, if necessary. By caching non-leaf blocks, the number of outdated blocks in cache will be small. Increasing the number of locations for a word will create more blocks on the respective tree, mostly leaf ones. Therefore, increasing the number of blocks also contributes to minimizing contention on a particular block.

3 Simulation

This system is composed by two distinct layers: DHT and B^+-tree. The simulator consists solely on the B^+-tree layer and does not take into account the DHT layer, which was substituted by a local hash table implementation. This decision focuses the simulation on message exchange between blocks exclusively. Messages contain requests for the index major operations and internal management operations also. Because the simulator considers that all peers have a single block and the DHT layer is not simulated, sending a message has an atomic cost. The effective cost of sending a message between two blocks would be, in this case, the cost of sending a message between any two peers on the DHT. Such cost depends on the DHT implementation but can be considered optimal. To simplify even further, messages are not restricted in size and are always delivered.

A peer can use two different connection models when acting as a client of the system by requesting an index operation: A cascading model where each peer delegates on another peer the operation it requests, recursively until the result is reached; A star-based model centered on the client where all messages are originated on the client and sent to each individual peer. Without excluding the cascading model as a possibility, the simulator uses the star model. This decision was influenced by the simulator's simple design, which does not consider each peer state individually and is not capable of simulating independent decisions based on different peers.

The simulation considers only a single client, from which index operations are requested by sending messages to blocks. A single index operation may require accessing several blocks, by sending one message to each peer. These messages may in turn trigger secondary block management operations. Because all request messages start from the client, the caching mechanism was made local to the client. This decision does not invalidate the caching simulation, as each peer would act as a client itself on a real system.

The data used for simulating was small news documents with in average 350 unique words each. About 1000 documents were processed with a total of 36499 unique words. Documents were grouped in sets with 100 elements each and inserted on the system separately. Each set represents the contents of an independent peer. A different cache state was used for each set, simulating the state each peer would acquire from it's interaction with the system. No search operation was simulated, at this stage, since it's block access pattern would be identical to the insert operation.

Figure 2 shows the number of accesses on all system blocks registered during the simulation, without using cache. It is possible to observe a large range of values, which is related with the word frequency. Namely, it is possible to see that few blocks, the root blocks for the popular words, had a very high access count.

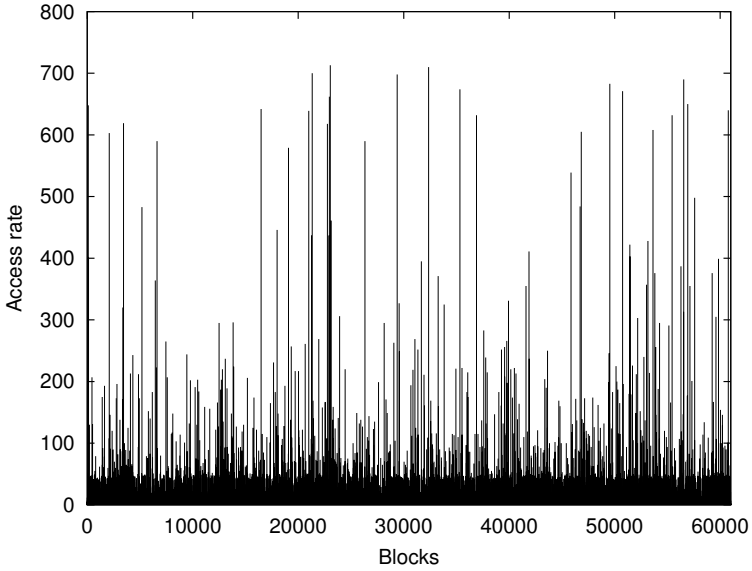

Figure 2: Block access rate without using cache.

Figure 3 shows the same simulation while using a caching mechanism. The access count range was reduced by a factor of 10, which contributed to flatten the access rate and reduce block contention.

When building a P2P distributed system, a key issue is both load balancing among the peers, as well as the minimization of the load in the P2P network. Our use of cached B^+-trees is consistent with these goals, but several other issues must still be explored, in order to fully assess the properties of this approach.

4 Current Work

We are currently working on the following issues:

- Measurement of the DHT system as a stand-alone implementation of an inverted index in order to compare it's performance with the system proposed in this article. We expect peers to be unbalanced in both storage and communication loads. We aim to prove that the B-tree will always be a more balanced solution on both storage and communication loads for any peer.

- Analysis of the block caching mechanism in order to determine the best caching size for different number of peers on the system.

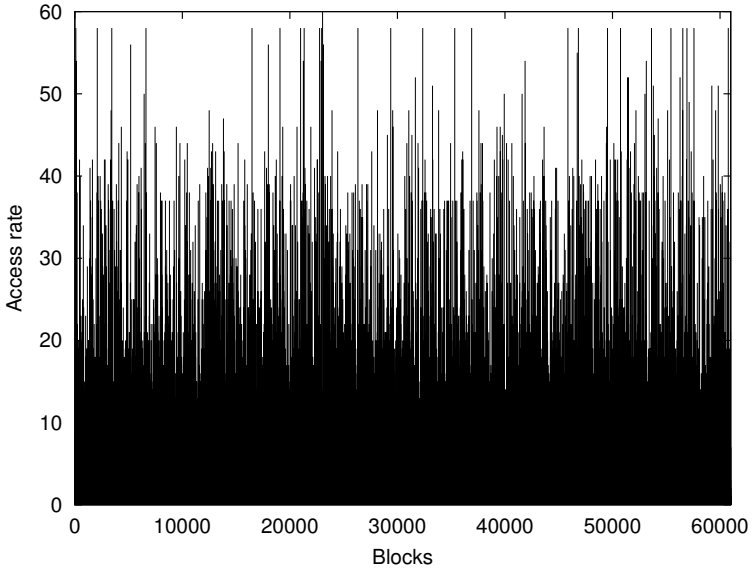

Figure 3: Block access rate using a cache mechanism (LRU).

- Implementation of the block to peer association in order to study the effective load by peers rather than by blocks. This would allow the simulation of peers with different block storage capacity. Another possible extension would be adding the message size parameter to the message cost procedure, allowing an effective cost analysis on peers.
- Implementation of the AND and OR search operators, increasing search flexibility. The measurement of the intermediary query processing load on peers should validate that no contention is created.

References

[1] R. Baeza-Yates and Ribeiro-Neto B. *Modern Information Retrieval*. ACM Press, NewYork, 1999.

[2] Zipf G. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, 1949.

[3] Gnutella website. http://gnutella.wego.com/.

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

[5] Fernando Pedone, Nelson Duarte, and Mario Goulart. Probabilistic queries in large-scale networks. In *Proceedings of the 4th European Dependable Computing Conference*, pages 209–226, 2002.

[6] Jordan Ritter. Why gnutella can't scale. no, really. http://www.darkridge.com/ jpr5/doc/gnutella.html, 2001.

[7] 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*, Germany, 2001.

[8] 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.

[9] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM'01 Conference*, pages 149–160, 2001.