B$^+$Trees on P2P: Providing content indexing over DHT overlays

Carlos Baquero* Nuno Lopes

Departamento de Informática, Universidade do Minho
{cbm,nuno.lopes}@di.uminho.pt

Abstract

The ability to search by content has been at the core of P2P data sharing systems and is a fundamental tool in the modern Web. However, currently deployed P2P search technology still suffers from either excessive centralization, abuse of network resources or low accuracy.

Efficient overlay structuring systems, like distributed hash tables (DHTs), provide adequate solutions to content location as long as unique identifiers are used. They cannot, however, directly support search without negative impacts on the load balance of data distribution among peer nodes.

We will show that DHTs can be used as a base for efficient content indexing by building a B$^+$Tree structure that coordinates the use of homogeneous size blocks, compatible with the DHT load balance assumptions. The remaining of the paper is dedicated to a discussion of some of the issues, problems and possible solutions, that need to considered when building complex data structures on top of a peer-to-peer DHT layer.

1 Introduction

In the last few years peer-to-peer (P2P) systems have been extensively used for cooperative file sharing among user nodes at the network edges. One of the basic required functionalities is the ability to search across shared contents. Search can target compact object descriptions, like a summary string or long file name, or extend to full text search in document objects. When a globally unique identifier is known we are no longer facing search but object location and/or retrieval.

Approaches to P2P search can be coarsely divided into two classes: those that rely on precomputed indexes [13, 11] and those that do not [6]. The first class pays an overhead when nodes join and update their contents, being influenced by the network dynamics. In the former case, overhead is at query time, and is often very significant in flood based strategies [14]. Also in the former case more reasonable query time overheads — for instance when using random walks instead of flooding [1, 10] — often leads to negative impacts on search accuracy. Determining which class exhibits the best overall behavior is still an open problem. The answer depends on a characterization of node uptimes and shared data stability that is not yet clear on present surveys [16].

In this position paper we will consider the case in which an inverted index of shared contents is to be kept along the P2P nodes. In contrast to the shared data itself, which is usually of a read-only nature, such an index is a dynamic structure whose mappings are added when new nodes connect and publish more contents, and are removed upon disconnections or content deletions.

Seminal work on P2P data structures introduced several efficient models of Distributed Hash Tables (DHTs) that cover a vast design space [17, 15, 12]. Designs span from systems where a node knows a constant number of nodes, $O(1)$, and lookups are $O(\log N)$ to the number of nodes N [9]; to systems where nodes keep knowledge of $O(\sqrt{N})$ nodes to achieve lookups in a fixed number of hops, $O(1)$ [7].

The effectiveness of P2P DHTs makes them obvious candidates for storing an inverted index that would support search. However, the data distribution patterns [5] that occur in such indexes break the assumptions made by DHT models on what

*Currently visiting Lancaster University Computing Department, under FCT grant BSAB/390/2003.

concerns load balancing among the nodes [2]. Inverted indexes map the most common keywords into a lengthy list of object locations, and if the keyword is to be used directly as a key in the DHT mapping, the whole list would be assigned to a single node. Load balancing approaches that split the contents into lists of fixed sized blocks are only adequate to immutable data, they cannot be used with mutable sets.

Our observation is that this problem can affect not only the construction of inverted indexes but also other mutable data structures, such as files or directories. Although the idea of using DHTs as building blocks for more complex structures is not new, the need to keep uniform loads on DHTs mappings is often overlooked [13]. However, DHTs operate correctly and are quite efficient when mapping keys to contents that exhibit an homogeneous scale.

In order to achieve a well balanced distribution of an inverted index and have a generic approach to the storage of mutable sets we have explored the construction of B^+Trees over DHT storage, taking advantage of the fixed overhead associated to B^+Trees [3]. This approach has a strong parallel to the use of inodes in the CFS storage system for read only data [4].

In the next session we give a brief description of the approach and then proceed to a more general discussion of the problems that arise when building such data structures over P2P DHTs.

2 Approach Synopsis

Our approach makes use of a DHT as a store for $(key \mapsto value)$ pairs. The DHTs uniform hash function ensures that pairs are evenly distributed across nodes. The B^+Tree blocks will be stored as values, assigning a uniform load to each pair.

While the reverse index stores information of the kind $(word \hookrightarrow location*)$ the B^+Tree will in fact be used to manage the set $location*$. For any given word in the inverted index the DHT is used to map that word into a B^+Tree root block. Each block can hold a given set of locations, as raw data, as well as other block identifiers for navigation in the B^+Tree. Since all blocks, from all B^+Trees, are stored in the same DHT, as well as word to root block associations, collisions must be prevented. To this end, the root words are used in conjunction with block positions within the tree in order to compose distinguished DHT keys.

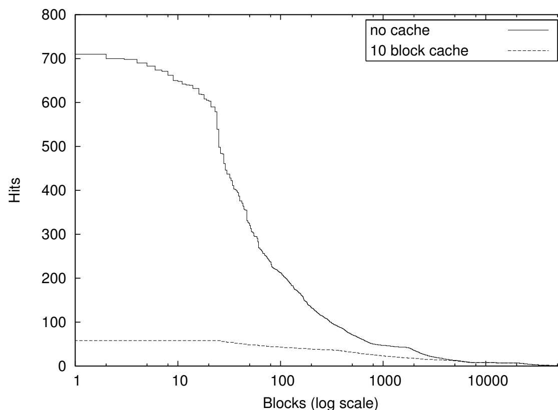

Figure 1: Number of accesses on individual blocks. With and without caching.

As a direct result from the use of B^+Trees, popular words will have its location set scattered along several blocks, and thus several nodes. The overhead for lookups is constant with respect to the number of nodes, and the average overhead for insertions and deletions is constant as well. The overhead will depend on the cardinality of the location set and grows in logarithmic proportion to its size. For most words, which are the non popular ones and therefore the most significant for searches, the overhead is non existent since the root block will act as a leaf block and hold the whole location set.

Load balancing of data storage across the nodes is achieved by the B^+Tree DHT combination and the spreading of block ids by the DHT hash function. However, unbalanced communication load can still be found on peers responsible for root and top level blocks of popular words. Fortunately those contention points are also the less subject to updates in the distributed structure and consequently are easy to overcome by simple caching techniques. Figure 1 depicts a simulation showing how a 10 block cache in each node can lead to a order of magnitude reduction in block accesses upon insert operations.

3 Discussion

Without having presented a complete analysis of the combined used of B$^+$Trees and DHTs, we hope to have established at this point the potential of combining traditional data structuring models with a base infrastructure of DHT mediated storage. In this section we will discuss some of the more general issues that come to surface in the process of making this combination.

3.1 DHT encapsulation

Keeping a rigid interface between the DHT middleware and the upper layers forces a client centered mode of operation. In this mode, the B$^+$Tree code is run at the node acting as client to the requested B$^+$Tree operation. Each block request must be served in turn by its DHT layer, meaning that although DHT routing keeps distributed, all the reconstruction of the stored set is done at the client.

In order to provide a distributed reconstruction of the stored sets, the B$^+$Tree algorithm must run across all nodes responsible for relevant blocks in the target B$^+$Tree. This can only be achieved by exposing the DHT routing to the B$^+$Tree algorithms, precluding the use of a simple *update(key,value), get(key), remove(key)* interface. Nevertheless, such approach would not be tied to any specific DHT routing algorithm, although the routing details would have to be considered for each case.

3.2 Data structures and code

The development of efficient distributed data structures over a P2P layer can only be achieved if the involved algorithms execute in the most appropriate nodes, with respect to the location of manipulated data [8]. A paradigmatic case is found when computing set intersections while searching for documents matching multiply query terms [13]. Retrieving both sets in the client node and them performing the intersection is clearly an inefficient approach.

For massively distributed data, operation shipping is often more efficient than data shipping. Consequently, the presentation of the distributed data middleware to its client code must accommodate programming models that provide an efficient manipulation of the distributed data structures. A good example of efficient manipulation of generic data structures can be found on the C++ STL programming model. It can constitute a good source of example both for algorithms and structures.

3.3 Cache Consistency

The use of caching techniques is crucial when taming communication contention zones due to: "flash crowds" for popular queries; and content hot spots on popular keywords. Two kinds of reference locality can be explored in the reverse index B$^+$Trees: one associated to blocks fetched due to client requests, and another linked to the requests originated from other nodes. The first case explores word locality within the contents that are exported in a given node, while the second case takes advantage of block correlations within the B$^+$Tree. It seems advisable to keep some separation between the two caches.

Regardless of the actual cache under consideration, it is not an option to propagate cache invalidations when changes occur. In order to avoid invalidations, block caching in B$^+$Trees is only applied to non-leaf blocks, so that stale entries can be detected and corrected, when fetching lower level blocks with newer versions. The cache control algorithm must then ensure that stale entries do not compromise the consistency of the stored data structure and that only performance is affected when using stale block references.

3.4 Garbage Collection

Content indexing over large volumes of data is often partially inconsistent with the data actually available. This happens in Web indexing, where the window of inconsistency is defined by the search cycle period. Recent contents are not present in the indexes and removed contents persist until the next cycle.

In our approach, contents are announced by nodes at join time, consequently reducing the window of inconsistency for new contents. Here the tradeoff was to favor communication at join time, reducing the load associated to queries.

Content removal on an announce based system cannot be simply approached by detecting a node

failure and asking all the nodes that index its contents to remove the appropriate entries. The main obstacle in this situation is the absence of a distributed direct index, mapping a node to its contents. These direct indexes are only available in the actual nodes at stake, and those can leave the system without prior notice.

One option to overcome this consists on having the nodes, that hold leaf blocks with locations, periodically check the availability of those locations and after given thresholds remove them, possibly propagating this removal along the B^+Tree. Another option is to associate a time-to-live tag to each content association and make sure that content holders initiate a refresh operation along the B^+Tree at appropriate times.

Both options would benefit from insight into the statistical distribution of node uptimes. From available data [16], it appears that the longer a given node has been connected, the more probable it is that it will remain connected. A complete characterization of uptimes will lead to an adequate calibration of the refresh, or polling, cycles that minimizes traffic while keeping a low level of outdated entries.

4 Conclusions

In this position paper we have brought attention to the inefficiency of DHTs when dealing with non-uniform mutable structures, of which a relevant example is found on inverted indexes used for announce based search. Next, we have shown that traditional data structuring tools, like B^+Trees, can be a solution to load balance data distribution. Simple caching techniques can also be applied in order to even communication loads.

We concluded with an analysis of the concerns that are at stake when building complex data structures on top of a DHT based storage substrate, and in particular B^+Tree based inverted indexes. One relevant finding is that the statistical properties of node uptimes and re-connection patterns are of primary importance for the determination of the adequate approaches to search in P2P and index garbage collection policies.

References

[1] Lada Adamic, Rajan Lukose, Amit Puniyani, and Bernardo Huberman. Search in power-law networks. *Physical Review E*, 64(046135), 2001.

[2] Carlos Baquero and Nuno Lopes. Towards peer-to-peer content indexing. *ACM Operating Systems Review*, 37(4):90, October 2003.

[3] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press, 1989.

[4] 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 System Principles*, Alberta, Canada, October 2001.

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

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

[7] Indranil Gupta, Kenneth Birman, Prakash Linga, Al Demers, and Robbert Van Renesse. Kelips: building an efficient and stable p2p dht through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Cambridge, USA, March 2003.

[8] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. Complex queries in dht-based peer-to-peer networks. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 242–259. Springer-Verlag, 2002.

[9] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Cambridge, USA, March 2003.

[10] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th international conference on Supercomputing*, pages 84–95. ACM Press, 2002.

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

[12] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of the ACM SIGCOMM'01 Conference*, pages 161–172, 2001.

[13] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, 2003.

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

[15] 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.

[16] 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.

[17] 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.