Scalable Data Access in Peer-to-Peer Systems Using Unbalanced Search Trees

Karl Aberer
Swiss Federal Institute of Technology (EPFL)
Switzerland
email: karl.aberer@epfl.ch

June 28, 2002

Abstract

With the appearance of Peer-to-Peer information systems the interest in scalable and decentralized data access structures is attracting increasingly interest. We propose to that end P-Grid, a scalable data access structure resulting from the distribution of a binary prefix tree. When adapting the P-Grid structure to skewed data distributions one obtains unbalanced search trees. The key result of this paper shows that unbalanced trees do not harm as long as communication is considered as the critical cost and the access structures are constructed properly. Besides proving this result we propose the necessary distributed, randomized algorithms that allow to construct the P-Grid in a self-organized manner such that the tree structure dynamically adapts to the data distribution and the aforementioned result is applicable.
Keywords: Scalable Data Access Structures, Peer-to-Peer Systems, Self-organization

1 Introduction

With the appearance of Peer-to-Peer systems the interest in scalable and decentralized data access structures is attracting increasingly interest. Such access structures emerge from the interaction of multiple peers. Each peer is storing a small fraction of the data and maintains a routing table to neighboring peers to forward search requests that it cannot answer. Different organisations of such routing schemes have been proposed. Most frequently they are constructed based on an underlying abstract model of binary search trees. This approach has been pursued by us [1], but also in [5], [6] and [7]. As a result, queries can be answered with at most $O(\log N)$ messages, where N is the number of peers, since the trees are balanced and thus the path length is limited.

However, with skewed data distributions, this approach may lead to very unevenly distributed workloads for the peers. In classical approaches to database

indexing this problem is addressed by using balanced tree structures, like B*-Trees, rather than binary search trees. Such an approach applied to P2P environments might pose substantial problems in terms of coordination among peers.

In this paper we will demonstrate that the use of balanced tree structures for data indexing is not necessarily required in a P2P environment. We will construct binary search trees by using randomized algorithms such that the storage space required at each peer is balanced. As consequence the search trees will no longer be balanced if the data distribution is not uniform.

We make the key observation that in a P2P environment not the depth of a search tree is critical for search performance, but the number of messages exchanged for a search. Our assumption is that the cost of traversals of paths in trees that are made locally at one peer can be neglected as compared to the cost of message exchanges. In case this is not true, we still may consider to use any local indexing structure to speed up local traversal. We will show that even if the (virtual) search trees, as implicitely represented in the structure of the routing tables, are extremely unbalanced, the number of messages used to process a query still scales gracefully. The assumption to be made for this property to hold is fairly weak: the entries in the routing tables must be selected with uniform probability from the set of peers that qualify for routing a query.

This result shows that in distributed, and in particular decentralized data management, we have to rethink some of the very fundamental assumptions on data management, that have evolved in the context of mostly centralized databases.

In the following Section 2 we first introduce the system model and the P-Grid data access structure that serve as the basis for our analysis. In Section 3 we introduce the distributed algorithm that results in unbalanced data access structures. In Section 4 we give then the main result showing that the search cost in terms of communication cost scales well despite of the unbalanced nature of the search tree. Finally, we give some directions on future research in the concluding Section 5.

2 System Model and Data Access Structure

We give an informal introduction into the decentralized, scalable data access structure, P-Grid, on which our subsequent discussion is based. More details are found in [1, 2]. Similar structures have also been proposed in [6], [5] and [7].

We assume that a set of N peers $P = \{p_1, \ldots, p_N\}$ is given. A data object $d \in D$ is identified by a binary key $key(d) = b_1 \ldots b_{k_d}, b_i \in \{0,1\}$, where the length k_d of the data key is constant. Each peer $p \in P$ has associated a binary key $key(p) = b_1 \ldots b_{k_p}, b_i \in \{0,1\}$, $k_p \leq k_d$. A peer p stores data objects d, where $key(p)$ is a prefix of $key(d)$. Thus each peer is responsible for the storage of part of the data. Note that the peers' key length may vary for different peers.

For each prefix $b_1 \ldots b_l$ of $key(p)$ of length l, $l = 1, \ldots, k_p$, the peer p maintains a routing table. It consists of a *reference list* to other peers p', that have

the same prefix of length l, but a different value at position $l+1$. We will call these references to other peers p's references at level l and denote it as $refs(p,l)$. These references are used to route search requests for data keys with prefix $b_1 \ldots b_l$ that do not match at position $l+1$ with $key(p)$.

We illustrate the data access structure in Figure 1. The position in the virtual search tree of *peer4* in the example P-Grid shows that it is responsible for all data objects with prefix '10', thus stores them. The keys implicitly partition the search space and define the structure of the virtual binary search tree. As can be seen from the figure, multiple peers may be responsible for the same path (e.g. *peer1* and *peer6* in Figure 1). This improves robustness and responsiveness, when peers are not online all the time but with a certain, possibly low, probability.

A search request can start at any peer and proceeds in the obvious way: starting from the first bit the search key is compared bit by bit to the peers' keys. When the local key matches the next bit is compared, if it does not match the query is forwarded to one of the peers from the routing table. For example, a query for '100' sent to *peer6* is forwarded to *peer5* at the first level (there is no common prefix between the path of *peer6* and the key, so that the peer has to directly forward the request to a peer he knows in the other half of the tree). Upon receiving the query, *peer5*, responsible for prefix '11', finds out that the query is to be forwarded to *peer4* (as the longest prefix between *peer5* and the key is of size one, the peer forwards the request to the other branch of the tree at the second level). *Peer4*, being responsible for keys beginning with '10', may finally deliver the content to the peer having issued the query. Note that in a real setting, multiple peers would be listed for each level in the routing tables, and one of them would be selected randomly when forwarding the request.

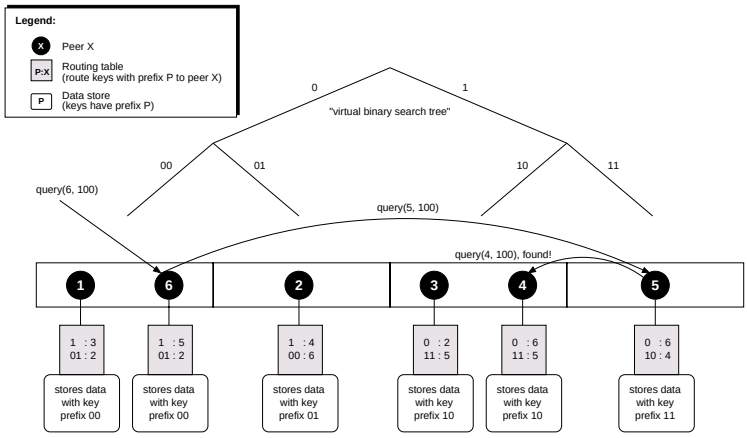

Figure 1: Sample P-Grid

In [1] we have shown that there exists an efficient, randomized and dis-

tributed algorithm for constructing such a data access structure based on random interactions among the peers. The basic idea is that whenever peers meet they refine their keys into opposite directions. If a global maximal keylength k_{max} is given, this algorithm results in a uniform distribution of keys over the peers. Each key will be associated on average with $\frac{N}{2^{k_{max}}}$ peers. From a global viewpoint the resulting access structure consists of fully balanced binary search trees, where each peer supports the search along one path from the root to a leaf of the tree. Thus the total amount of resources scales linearly in number of peers and the search cost, both in time and number of messages generated scales logarithmically.

3 Constructing the Data Access Structure

With skewed data distributions, as they can be expected in realistic applications, using a balanced binary tree for data distribution and search would imply an unevenly distributed workload among the peers. Therefore we use a modified version of the construction algorithm, that has been introduced in [1]. This modified algorithm results in unbalanced search trees. We assume that the peers store already some data that reflects the actual data distribution. Throughout the construction process, whenever peers meet, they not only extend their corresponding keys, but they also exchange their data correspondingly. In that way the data is globally disseminated and the peers use the data to guide the construction process. We no longer make use of a maximal keylength parameter k_{max} to control the termination of the construction process as in [1]. Rather peers extend their key only if sufficient data, i.e. a minimal number of data items, is known to them to justify a further extension of their key.

The algorithm is given as pseudcode in Algorithm 1. It consists of the following main steps:

- Exchange of references: This leads to an increasingly uniform distribution of references in the reference lists of the different peers. Note that during an exchange no references are lost.
- Exchange of data: This leads to an increasing distribution and replication of data among the peers. This step also ensures that eventually all data objects are stored at a peer that is responsible for them, i.e. where the peer's key is a prefix of the data object key.
- Extension of the key associated with a peer: This step is only executed if the peer has after the exchange a sufficient number of data objects related to the extended key. In this way the lengths of the peers' keys flexibly adapt such that each peer stores on average the same number of data objects.

We explain the different steps of the algorithm in more detail: the algorithm is executed each time two peers p_1 and p_2 meet (they meet either actively in order to construct the P-Grid or as side-effect of other activities, such as pinging

Algorithm 1 Exchange algorithm two peers p_1 and p_2 perform in order to construct the P-Grid structure

$exchange(p_1, p_2, r)$

1: **if** $length(key(p_1)) > length(key(p_2))$ **then**
2: swap p_1 and p_2 { make sure that p_2 has the longer key }
3: **end if**
4: lc = length of common prefix of $key(p_1)$ and $key(p_2)$;
5: **if** $lc > 0$ **then**
6: exchange references between $refs(p_1, lc)$ and $refs(p_2, lc)$;
7: **end if**
8: $l_1 = length(key(p_1)) - lc$; $l_2 = length(key(p_2)) - lc$;
9: **if** $l_1 > 0$ and $l_2 > 0$ **then**
10: the peers select from each other data stores the data that belong to their key; {peers may store data that does not correspond to their key }
11: now forward the peer with the shorter path (which is p_1) to another peer using $refs(p_2, l_1)$;
12: **end if**
13: **if** $l_1 = 0$ and $l_2 = 0$ **then**
14: $k_1 = key(p_1)$ extended by a random bit;
15: $k_2 = key(p_2)$ extended by the inverse of the random bit;
16: $d_1 = SelectData(data(p_1) \cup data(p_2)), k_1)$; $d_2 = SelectData(data(p_1) \cup data(p_2)), k_2)$; $\{SelectData(d, k)$ selects from the set d the data objects with a key of which k is prefix$\}$
17: ld_1 = number of data items in d_1; ld_2 = number of data items in d_2; { now proceed differently depending on the number of data objects each peer holds; $minstorage$ is the minimal number of data items required in order to extend the key of a peer by an additional bit }
18: **if** $ld_1 > minstorage$ and $ld_2 <= minstorage$ **then**
19: $key(p_1) = k_1$; update reference lists and exchange data objects;
20: **end if**
21: **if** $ld_1 <= minstorage$ and $ld_2 > minstorage$ **then**
22: $key(p_2) = k_2$; update reference lists and exchange data objects;
23: **end if**
24: **if** $ld_1 > minstorage$ and $ld_2 > minstorage$ **then**
25: $key(p_1) = k_1; key(p_2) = k_2$; update reference lists and exchange data objects;
26: **end if**
27: **if** $ld_1 <= minstorage$ and $ld_2 <= minstorage$ **then**
28: $data(p_1) = data(p_2) = d_1 \cup d_2$;
29: **end if**
30: **end if**
{ now we treat the cases where the peers have paths of different length; extend the path of peer p_1 if it has already a sufficient number of data objects corresponding to the extended key }
31: **if** $l_1 = 0$ and $l_2 > 1$ **then**
32: k_1 is extended by the inverse bit from $key(p_2)$ at position $lc + l_1$;
33: ld_1 = number of data objects matching key k_1 ;
34: **if** $ld_1 > minstorage$ **then**
35: $key(p_1) = k_1$; update reference lists and exchange data objects;
36: **end if**
37: **end if**

each other). First they decide which peer has the longer key and determine the common prefix of their keys (line 1-4). Then they exchange at the level of their common prefix references from their routing tables (line 5-7). This step is required in order to disseminate and uniformly distribute references across all routing tables. Then the peers continue depending on how their keys are related (line 8). If the keys extend the common key prefix in different ways (line 9-12), the peers exchange data objects that the other peer might (wrongly) store (line 10). Then the peer with the shorter part uses the routing table of the other peer in order to locate another peer, in order to perform the exchange algorithm with it (line 11). This ensures that each exchange initiates some extension of keys if possible and the data access structure is constructed quickly. The next case is where the keys of the peers are equal (line 13-30), and it is the main construction step. The peers extend their keys by opposite bits which are randomly selected (line 14-15) and then determine the number of data objects they know for the two new keys from their both two stores (line 16-17). Depending on whether enough data objects are present (*minstorage*) they will decide to adopt the new key (line 18-26). When a peer extends its key, it has to obtain all the data objects that belong to the new key, and to create a new reference list for the new routing level, which at the beginning will only contain a reference to the peer it currently encounters. Only later the reference list will be filled with other entries, due to the reference exchanges (line 5-7). If both peers decide not to extend their keys they exchange all their data objects (line 27-29). In this way data is replicated, and in later encounters with other peers enough data objects may accumulate in order to allow an extension of the key. The last case to consider is where the key of one peer is a prefix of the key of the other peer (line 31-36). All the steps performed here are essentially the same. Only the peer with the shorter key is now forced to extend its key opposite to the key of the second peer.

We have simulated the algorithm in order to demonstrate its effectiveness in constructing a binary prefix tree of which the depth is correlated to the data distribution. In this simulation we have used 1024 peers. The data that they store follow a Zipf distribution, i.e. it is heavily skewed. The value *minstorage* is selected such that peers would hold at least $6\frac{2}{3}$ replicas of the data objects on average. An average replication factor of approximatively $11\frac{1}{3}$ results after the simulation converged to a stable state. On average 16 exchanges were executd by each peer, which shows that the algorithm scales well. An average key length of approximately $4\frac{1}{2}$ was reached by the peers. A substantial number of peers has reached a key length of 5. For those we compared the number of peers associated with each of the $2^5 = 32$ possible keys, to the number of data objects with a data key of which the peers' key is the prefix. The result is shown in Figure 2. It shows for each key the percentage of peers respectively data objects associated with the key. The result shows nicely that the distribution of the peers and data objects are closely correlated.

In summary, this distributed and randomized algorithm for constructing a P-Grid has the following effects:

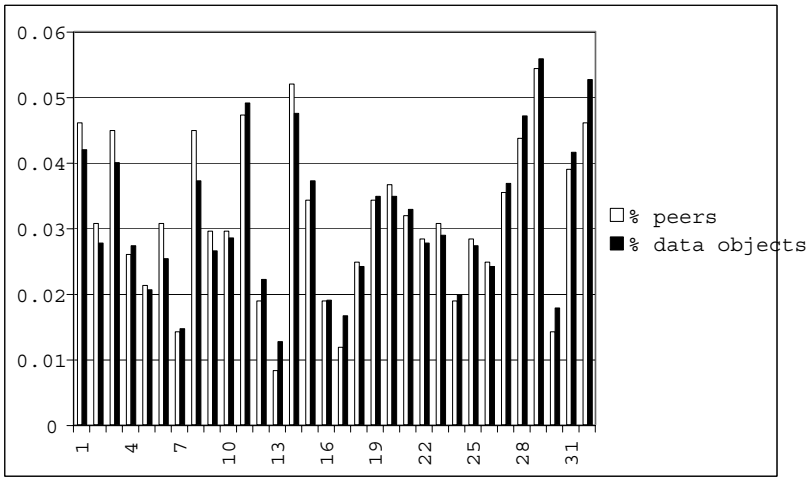

Figure 2: Comparison of peer distribution to data distribution

1. Each peer will store on average the same number of data items and thus require comparable resources for storage.

2. The lengths of the keys that are associated with the peers in a key interval are related to the number of data objects occuring in that interval. For key intervals where many data items occur, the keys tend to grow longer than in key regions with fewer data items. Therefore the (virtual) tree that is constructed is unbalanced.

3. No global parameter k_{max} is required as in the earlier version of this algorithm. It is replaced by the global data distribution, which we can assume to be available. This is a more natural form of global knowledge to exploit in the search tree construction than global knowledge on the maximal keylength, which encodes essentially knowledge on the global number of data objects in relation to the local storage capacity of a peer. Thus we further reduce global knowledge required for index construction and increase the degree of self-organization.

Constructing an access structure that is based on an unbalanced tree poses however another problem: it is no longer guaranteed that search paths remain short. In fact, the trees may now contain paths that in the worst case have length $O(N)$ which would defeat the purpose of having a search tree. This is the problem we will address in the following section.

4 Analyzing Search Cost

At this point we have to consider the physical characteristics of P2P systems. Actually we are not so much interested in the absolute length of the search path from the root to the leaf of a search tree, but rather in the number of messages that need to be exchanged in order to find a data object. This number will be lower than the path length if we can traverse multiple tree nodes at the same peer. Local processing of the traversal of multiple tree nodes is not critical, as we may assume that the cost of local processing is by orders of magnitude faster than message exchanges between peers. Even if the local processing cost were non-negligible it could be sped up by using a local indexing structure.

Therefore, in the following we consider only the number of message exchanges in a search process. The main finding is: even in the case a search tree is extremely unbalanced, the number of messages will remain low, i.e. $O(\log N)$, assuming the peer distribution found in the reference lists is the same as the global distribution of peers having the corresponding key.

More precisely, for a reference list $ref(p,l)$ of a peer associated with key $key(p) = b_1 \dots b_{k_p}$ there exists a set of peers $p' \in \hat{P}$, that qualify to occur within this list. These are the peers p' with key $key(p') = b_1 \dots b_{l-1} b_l^{-1}$, where $b_l^{-1} = (1 - b_l)$. We need to make the assumption that the probability to occur in $ref(p,l)$ is the same for each peer $p' \in \hat{P}$, i.e. $Prob(p' \in ref(p,l)) = const$ for all $p' \in \hat{P}$. This property is ensured by the step of merging the reference lists of different peers in the exchange algorithm (see Algorithm 1, line 5-7).

We have not described in detail this step yet. In this step two peers p_1 and p_2 with reference lists $r_1 = ref(p1,l)$ and $r_2 = ref(p_2,l)$ construct from the two lists two new lists by randomly exchanging the entries of r_1 and r_2. This needs to be done however with care in order to arrive at the desired distribution, where each entry occurs with the same probability. Informally, the peers first each randomly distribute the entries from the set $r_1 \cup r_2$ among each other, by maintaining the relative order of the entries. If after that step further entries can be added, the peers select from each others reference lists as many additional references as possible, starting from the beginning of the list. Thus some of the references are replicated. Observing this order of selecting the additional, replicated entries, implicitely ensures that always the less frequently occurring entries are replicated first. Therefore the algorithm tends to balance the global distribution of references.

In order to demonstrate that the algorithm achieves the desired behavior we give a simluation result. We assume that a population of 32 peers interacts in exchanging their references. This would correspond to the exchange of references at level $l = 5$ when having a total of 1024 peers. Furthermore we assume that there exist 64 peers that may be referenced, i.e. that 64 peers occur in the union of all reference lists of the 32 peers. Note, that this is a part of the search tree that is particularly deep; in a balanced tree one would expect only 32 peers at level 5. The reference lists themselves have maximal length 24. The mean frequency of the 64 referenced peers possibly occuring in a reference list

is then 12 (also counting references with frequency zero). We start with a Zipf distribution of the reference frequencies. After about 10 exchange operations executed by each peer, we arrive at a stable state where the standard deviation of the frequencies of peers occuring in the reference lists does not further decrease. This result is shown in Figure 3. One can see that at the beginning the standard deviation is around 15 and then it drops quickly to a value of around 2.7. More importantly, the distribution of frequencies of peers occurring in reference lists has changed from a Zipf distribution to a distribution that closely resembles to a binomial distribution with mean value 12 (Figure 4). In fact, the standard deviation for the binomial distribution $B(n; prob)$ with $n = 32$ and $prob = \frac{24}{64}$ is 2.74, which is very close to the value obtained in the simulation. $prob = \frac{24}{64}$ is the probability for one of the 64 possible references to occur in a reference list of length 24 and the binomial distribution gives thus the distribution of frequencies assuming that each reference appears with constant probability.

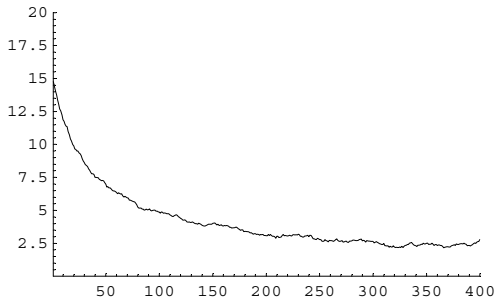

Figure 3: Decrease of standard deviation of frequencies of references in the reference lists

We give now the proof of our main claim. For doing this we will make use of the fact that the peers in the reference lists have constant probability for occurrence, which as we could see is fairly well achieved by the exchange algorithm for reference lists.

For proofing the claim we analyze the search process. We consider one specific key at a leaf node of the search tree. A search for this key starts at some peer. In each step (going from one bit of the key to the next) either the peer itself has a matching bit, and therefore no message is required, or a message has to be sent to another peer, following one of the references stored in the routing table at the corresponding level. There the same process continues until the leaf level is reached.

Let us look at the search paths and the number of peers responsible for a fixed key. At level 0 we have $n_0 = N$ peers, namely all peers are responsible for the empty key. Of these peers n_1 do not match the first bit, whereas $n_0 - n_1$ do. Thus, with probability $\frac{n_1}{n_0}$ a message is required, in order to arrive at a peer

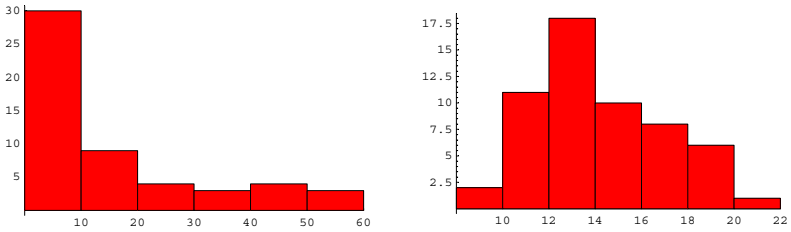

Figure 4: Frequency distribution of peers in reference lists before and after 400 exchanges

with the first bit matching if a randomly selected peer of the peer population is selected.

Once this peer is found we look at the next bit. Again n_2 peers, among the $n_0 - n_1$ remaining peers that match the first bit, will not match the second bit. Since we can assume that the distribution of peers within reference lists is the same as the global distribution of peers, in other words each peer occurs with the same constant probability, with probability $\frac{n_2}{n_0 - n_1}$ a message is sent to find a peer matching the bit. We can continue now this process for k steps if the key has length k. The whole process is illustrated in the Figure 5.

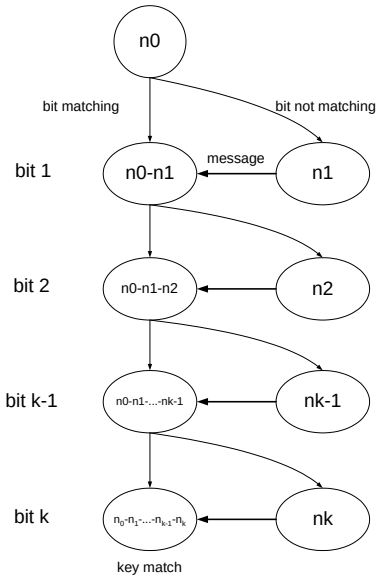

Figure 5: Search process

Adding up the expected number of messages in each step results in the expected total number of messages for the search process:

$$\sum_{i=1}^{k} \frac{n_i}{n_0 - \ldots - n_{i-1}}$$

for an arbitrary sequence of positive numbers $n_1, \ldots, n_k$ with $\sum_{i=1}^{k} n_i < n_0$. In order to estimate the value of the expected number of messages we can proceed as follows. Each term can be considered as the area of a rectangle. We have arranged these rectangles as in Figure 6.

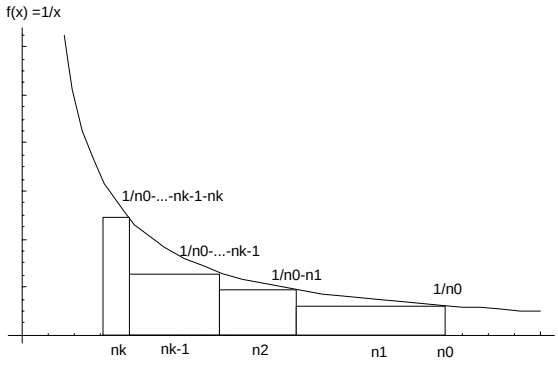

Figure 6: Deriving an upper bound for the number of expected messages

One can see from this figure, that the rectangles fall exactly under the curve of the function $f(x) = \frac{1}{x}$. Therefore the total area of the rectangles is smaller than

$$\int_{r}^{n_0} \frac{1}{x}\, dx = \log n_0 - \log r$$

where $r = n_0 - \sum_{i=1}^{k} n_i \geq 1$. This results in a bound

$$\sum_{i=1}^{k} \frac{n_i}{n_0 - \ldots - n_{i-1}} < \log n_0 = \log 2 \ \log_2 N.$$

which shows that independently of how the search tree is unbalanced the number of messages will always remain of order $O(\log N)$. This bound is in fact very tight. For example, it is close to $\frac{1}{2} \log_2 N$, which is the expected number of messages if the search tree were exactly balanced.

5 Conclusion

We have shown in this paper that the property of search trees being balanced is of a quite different relevance if we look at decentralized system environments and take into account the physical characteristics of them. The result is not contradicting findings on the impossibility of constructing distributed balanced search trees [4], rather it relativizes the importance of the question of having balanced search trees.

Questions for further investigation relate in particular to practical, algorithmic problems. For large datasets the algorithm for adapting the tree depth could operate on representative samples of data objects rather than the whole dataset a peer holds. This would reduce the bandwidth consumed during the tree construction phase. Such an approach would however require complementary methods for disseminating quickly all data objects to their corresponding peers once the access structure is constructed. Also dynamic settings, where the tree structure adapts to changing data distributions, are an interesting question for further study. In this work we have assumed that the tree structure adapts to the data distribution. However, it would be also of interest to study methods for adapting to the distribution of the access frequencies, such as proposed in [3], by using dynamic replication of data objects. This in turn has an impact on the analysis of the expected number of messages required for searching data objects.

We are currently integrating the mechanism described in this paper into the implementation of our prototype P2P content sharing system, Gridella [2], which we intend to evolve over time into a completely decentralized full-fledged data management system.

References

[1] K. Aberer, *P-Grid: A self-organizing access structure for P2P information systems* Proceedings of the Ninth International Conference on Cooperative Information Systems (CoopIS 2001), Trento, Italy, 2001.

[2] K. Aberer, M. Punceva, M. Hauswirth, R. Schmid, *Improving Data Access in P2P Systems*, IEEE Internet Computing, Jan./Feb. 2002.

[3] P. Cudre-Maroux, K. Aberer, *A Decentralized Architecture For Adaptive Media Dissemination*, Proceedings IEEE International Conference on Multimedia and Expo, Lausanne, Switzerland, 2002.

[4] B. Kröll, P. Widmayer *Balanced Distributed Search Trees Do Not Exist* WDAS 95, p 50-61, 1995.

[5] C. G. Plaxton, R. Rajaraman, A. W. Richa, *Accessing Nearby Copies of Replicated Objects in a Distributed Environment*, Proceedings of the 9th Annual Symposium on Parallel Algorithms and Architectures, pp. 311-20, 1997.

[6] S. Rhea et al. *Maintenance-free Global Data Storage*, IEEE Internet Computing, vol.5, no.5, Sept./Oct.2001, pp. 40-49.

[7] I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan: *Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications* Proceedings of the ACM SIGCOMM, 2001.