

P-Grid: A self-organizing access structure for P2P information systems

Karl Aberer

Department of Communication Systems
Swiss Federal Institute of Technology (EPFL)
1015 Lausanne, Switzerland
karl.aberer@epfl.ch

Abstract

Peer-To-Peer systems are driving a major paradigm shift in the era of genuinely distributed computing. Gnutella is a good example of a Peer-To-Peer success story: a rather simple software enables Internet users to freely exchange files, such as MP3 music files. But it shows up also some of the limitations of current P2P information systems with respect to their ability to manage data efficiently. In this paper we introduce P-Grid, a scalable access structure that is specifically designed for Peer-To-Peer information systems. P-Grids are constructed and maintained by using randomized algorithms strictly based on local interactions, provide reliable data access even with unreliable peers, and scale gracefully both in storage and communication cost.

Keywords. Peer-To-Peer computing, Distributed Indexing, Distributed Databases, Randomized Algorithms.

1 Introduction

Peer-To-Peer (P2P) systems are driving a major paradigm shift in the era of *genuinely* distributed computing. Major industrial players believe “P2P reflects society better than other types of computer architectures. It is similar to when in the 1980’s the PC gave us a better reflection of the user” (www.infoworld.com).

In a P2P infrastructure, the traditional distinction between clients and back-end (or middle tier application) servers is simply disappearing. Every node of the system plays the role of a client and a server. The node *pays* its participation in the global exchange community by providing access to its computing resources. Gnutella is a good example of a P2P success story: a rather simple software enables Internet users to freely exchange files, such as MP3 music files.

In the current P2P file-sharing systems, like Gnutella [1], no indexing mechanisms are supported. Search requests are broadcasted over the network and each node receiving a search request scans its local database (i.e. its file system) for possible answers. This approach is extremely costly in terms of communication and leads to high search costs and response times. For supporting efficient search, however, appropriate access structures are prerequisite.

Access structures in distributed information systems have been addressed mostly in the area of distributed and parallel databases. Different approaches to data access in distributed environments have been taken there. We mention some of the principal approaches that can be found.

- The distribution of one copy of a search tree over multiple, distributed nodes is a technique that has been investigated in [7]. The same authors have shown that, under certain assumptions, in that with that approach balanced search trees do not exist [8].

- The replication of the complete search structure is an approach that underlies the RP^* -Trees proposed in [9]. In [11] a mechanism is proposed that leads eventually to the replication of the search structures.
- Scalable replication of a search tree (more precisely B-Tree) is proposed in [6] (dB-Tree) and [12] (Fat-BTree). With scalable replication each node stores a single leaf node of the search tree, the root node of the search tree is replicated to every node, and the intermediate nodes are replicated such that each node maintains a path from the root to its own leaf.
- No search structures: in these approaches operation messages are broadcasted to all participating nodes. E.g. with RP_N^* [9] the data is range partitioned as in B-Trees but no index exists and a multicast mechanism is used. In current P2P file sharing systems like Gnutella the P2P network is used to propagate search requests to all reachable peers.

Most of these approaches assume a reliable execution environment, require some centralized services, are designed for a fairly small numbers of nodes (hundreds) or focus on deterministic execution guarantees. In this paper we would like to take a different approach and investigate the question of how an access structure can be built in a community consisting of a very large number of unreliable peers without any central authority, that can provide still a certain level of reliability of search and scales well in the number of peers, both in storage and communication cost. In order to obtain scalability we use the approach of scalable replication of tree structures, as proposed in [6, 12]. To build and use these search structures we use randomized algorithms that are exclusively based on local interactions among peers. The idea is that by randomly meeting among each other the peers successively partition the search space and retain enough information in order to be able to contact other peers for efficiently answering future search requests. The resulting distributed access structure we call *P-Grid* (Peer-Grid). As this investigation is intended to clarify whether such an approach is principally feasible we make the simplifying assumption that data distribution is not skewed. Thus the use of binary search trees over a totally order domain of keys is sufficient (as e.g. also used in [7]). Similarly, for the analysis and simulation of P-Grids, we make uniformity assumptions on the behavior of the peers. Though definitely in a next step skewed data distributions and extended methods for balancing the search trees (as well known from B-Tree structures) are required, even the basic methods proposed in this paper could be extremely beneficial to improve the efficiency of current file sharing applications.

The main characteristics of P-Grids are

- they are completely decentralized, there exists no central infrastructure, all peers can serve as entry point to the network and all interactions are strictly local.
- they use randomized algorithms for constructing the access structure, updating the data and performing search; probabilistic estimates can be given for the success of search requests, and search is robust against failures of nodes.
- they scale gracefully in the total number of nodes and the total number of data items present in the network, equally for all nodes, both, with respect to storage and communication cost.

In the next section we introduce our system model and the structure of P-Grids. In Section 3 we describe the distributed, randomized algorithm that is used to construct P-Grids. In Section 4 we give some analysis on basic properties of P-Grids. Section 5 contains extensive simulation results, that demonstrate the feasibility of the P-Grid algorithms and exhibit important scalability properties of P-Grids.

2 System Model and Access Structure

We assume that a community of peers P is given that can be addressed using a unique address $addr : P \rightarrow ADDR$. For an address $r \in ADDR$ we define $peer(r) = a$ iff $addr(a) = r$ for $a \in P$. The peers are online

with a probability $online : P \rightarrow [0, 1]$. Peers that are online can be reached reliably using the underlying communication infrastructure by means of their address.

Every peer stores information items from a set DI that are characterized by an index term from a set K . The set of index terms is totally ordered, such that a search tree can be constructed in the usual way. In the following, we assume that the index terms are binary strings, built from 0's and 1's and that a key $k = p_1 \dots p_n$, $p_i \in \{0, 1\}$ corresponds to a value $val(k) = \sum_{i=1}^n 2^{-ip_i}$ and an interval $I(k) = [val(k), val(k) + 2^{-n}[\subseteq [0, 1[$.

Now we define the access structure. The goal is to construct an access structure, such that

- The search space is subsequently partitioned into intervals of the form $I(k)$, $k \in K$. Every peer takes over responsibility for one interval $I(k)$. As each key corresponds to a path in the binary search tree we will also say that the peer is responsible for the *path* k .
- Taking over responsibility for an interval $I(k)$ means, that a peer should provide the addresses of all peers that have an information item with a key value k_{query} that belongs to $I(k)$, i.e. $val(k_{query}) \in I(k)$.
- For each prefix k_l of k of length l , $l = 1, \dots, length(k)$ a peer a maintains references to other peers, that have the same prefix of length l , but a different value at position $l + 1$, for the key they are responsible for. We will call these references to other peers, a 's references at level $l + 1$. These references are used to route search requests for keys with the same prefix k_l , but a continuation that does not match the own key, to other agents.
- A search can be started at each peer.

Before giving the formal definition of the access structure, we show in Figure 1 a simple example to illustrate the idea. The different levels relate to the different levels of the binary search tree. The intervals relate to the nodes of the search tree. We indicate the key values that correspond to the intervals (i.e. 0 and 1 at level 1, 00, 01, 10, and 11 at level 2). At the lowest level we have entered 6 peers, indicated by the black circles, into the leaf nodes corresponding to the keys for which they are responsible for. One can see that multiple peers can be responsible for the same key. We will call these later also *replicas*. We also entered the agents into all intervals on the path from the root to their leaf node. At the leaf nodes we show the pointers to specific data items that have a key of which the key related to the leaf node is a prefix.

One can see that at level zero every peer is associated with the whole interval, in other words, it stores a root node of the search tree. At level 1 every peer is associated with exactly one of the two intervals. At level 2 every peer is associated with exactly one interval.

The connectors from one level to the next are the references that each peer maintains to cover the other side of the search tree. For example, at level 0 peer 1 has a reference to peer 3, and at level 1 peer 1 has a reference to peer 2.

When a search request is issued it is routed over the responsible peers. There are two possibilities, either at the next level the peer itself is responsible, then it can further process the request itself, or, the request needs to be forwarded to another peer. For illustration we have included into the Figure the processing of two queries. In the first example the query 00 is submitted to peer 1. As peer 1 is responsible for 00 it can process the complete query. In the second example the query 10 is submitted to peer 6. Using its reference at level 0 it contacts peer 5, which in turn contacts at level 1 peer 4, who is responsible for the key corresponding to the query.

We define now formally the data structure for peers that allows to represent the P-Grid. Every peer $a \in P$ maintains a sequence

$$(p_1, R_1)(p_2, R_2) \dots (p_n, R_n),$$

where $p_i \in \{0, 1\}$ and $R_i \subset ADDR$. We define $path(a) = p_1 \dots p_n$, $prefix(i, a) = p_1 \dots p_i$ for $0 \leq i \leq n$ and $refs(i, a) = R_i$. In addition the number of references in R_i will be limited by a value $refmax$.

The sets R_i , $1 \leq i \leq n$ are references to other peers and satisfy the following property:

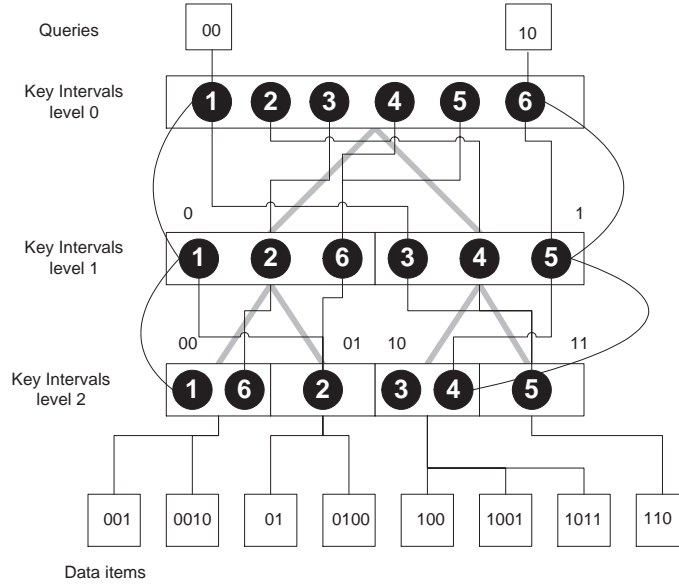


Figure 1: Example P-Grid

$$r \in refs(i, a) : prefix(i, peer(r)) = prefix(i-1, a)p_i^-$$

where $path(a) = p_1 \dots p_n$ and p^- is defined as $p^- = (p + 1) \text{ MOD } 2$. In addition, each peer maintains a set of references $D \subset ADDR \times K$ to the peers that store data items indexed by keywords k for which $path(a)$ is a prefix. In other words, at the leaf level the peer knows at which peers data items corresponding to the search keys that it is responsible for, can be found.

This gives rise to a straightforward depth first search algorithm using the access structure:

```

query(a, p, l)
{
found = FALSE;
rempath = sub_path(path(a), l+1, k);
compath = common_prefix_of(p, rempath);
IF length(compath) = length(p) THEN result = a ELSE
  IF length(path(a)) > l + length(compath) THEN
    querypath = sub_path(p, length(compath) + 1, length(p));
    refs = refs(l + length(compath) + 1, a);
    WHILE |refs| > 1 AND NOT found
      r = random_select(refs);
      IF online(peer(r))
        found = query(peer(r), querypath, l + length(compath));
RETURN found;

/* Comment: sub_path(p1...pn, l, k) := p1...pk
common_prefix_of(p1...pk pk+1...pn, p1...pk qk+1...ql = p1...pk)
random_select(refs) returns a random element from refs
and removes it from refs */
}

```

Given a P-Grid, a query p can be issued to each peer a through a call $query(a, p, 0)$.

3 P-Grid Construction

Having introduced the access structure and the search algorithm, the main question is now, of how a P-Grid can be constructed. As there exists no global control this has to be done by using exclusively local interactions. The idea is that whenever peers meet, they use the opportunity to create a refinement of the access structure. We do not care at this point why peers meet. They may meet randomly, because they are involved in other operations, or because they systematically want to build the access structure. But assuming that by some mechanisms they meet frequently, the procedure works as follows.

Initially, all peers are responsible for the whole search space, i.e. all search keys. At that stage, when two peers meet initially, they decide to split the search space into two parts and take over responsibility for one half each. They also store the reference to the other peer in order to cover the other part of the search space. The same happens whenever two peers meet, that are responsible for the same interval at the same level.

However, as soon as the P-Grid develops, also other cases occur. Namely, peers will meet

1. their keys share a common prefix,
2. their keys are in a prefix relationship.

In case 1, what the peers can do is to initiate new exchanges, by forwarding each other to peers they are themselves referencing. In case 2 the peer with the shorter key can specialize by extending its key. In order to obtain a balanced P-Grid it will specialize in the opposite way the other peer has already done at that level. The other peer remains unchanged.

These considerations give rise to the following algorithm that two peers $a1$ and $a2$ execute when they meet.

```
exchange(a1, a2, r)
{
  commonpath = common_prefix_of(path(a1), path(a2));
  lc = length(commonpath);
  IF lc > 0
    (* exchange references at the level where the paths agree *)
    commonrefs = union(refs(lc, a1), refs(lc, a2));
    refs(lc, a1) = random_select(refmax, commonrefs);
    refs(lc, a2) = random_select(refmax, commonrefs);
    l1 = length(sub_path(path(a1), lc + 1, length(path(a1))));
    l2 = length(sub_path(path(a2), lc + 1, length(path(a2))));
    (* Case 1: if both remaining paths are empty introduce a new level *)
    CASE l1 = 0 AND l2 = 0 AND length(commonpath) < maxlength
      path(a1) = append(path(a1), 0);
      path(a2) = append(path(a2), 1);
      refs(lc + 1, a1) = {a2};
      refs(lc + 1, a2) = {a1};
    (* Case 2: if one remaining path is empty split the shorter path *)
    CASE l1 = 0 AND l2 > 0 AND length(commonpath) < maxlength
      path(a1) = append(path(a1), value(lc+1, path(a2))^-);
      refs(lc + 1, a1) = {a2};
      refs(lc + 1, a2) = random_select(refmax, union({a1}, refs(lc+1, a2)));
    (* Case 3: analogous to case 2 *)
    CASE l1 > 0 AND l2 = 0 AND length(commonpath) < maxlength
```

```

    path(a2) = append(path(a2), value(lc+1, path(a1)))^-;
    refs(lc + 1, a2) = {a1};
    refs(lc + 1, a1) = random_select(refmax, union({a2}, refs(lc+1, a1)));
    (* Case 4: recursively perform exchange with referenced peers *)
    CASE l1 > 0 AND l2 > 0 AND r < recmax,
        refs1 = refs(lc+1, a1) \ {a2};
        refs2 = refs(lc+1, a2) \ {a1};
        FOR r1 IN refs1 DO
            IF online(peer(r1)) THEN exchange(a2, peer(r1), r+1);
        FOR r2 IN refs2 DO
            IF online(peer(r2)) THEN exchange(a1, peer(r2), r+1);

/* Comment: random_select(k, refs) returns a set with k random elements from refs
    append(p1...pn, p) = p1...pn p
    value(k, p1...pn) = pk
    p^- = 1+p MOD 2 */
}

```

A few remarks are in place. A measure to prevent overspecialization of peers is to bound the maximal length of paths that can be constructed to *maxlength*. Simulations show that this results in a more uniform distribution of path lengths among peers and better convergence of the P-Grid. Also such a bound is needed when a certain degree of replication at the lowest grid level is to be achieved. The disadvantage is that some global knowledge is used, namely the maximal path length, which not always might be locally available or easily derivable. In practical applications, one possible indication that a path has reached *maxlength* could be that the number of data items belonging to the key is falling below a certain threshold.

An alternative would be to avoid overspecialization by taking another approach in Case 2 and Case 3, where one path is subpath of the other and the peer with shorter path chooses to specialize differently than the other peer. Here one could shorten the longer path if the difference in length is greater than 1, such that both resulting paths have the same length. However, this would require additional means to maintain consistency of references as peers give up responsibility for keys by generalizing and could possibly specialize at a later stage differently and slows down convergence. So we omitted this possibility here.

The recursive executions of the exchange function by using the locally available references have an important influence on the performance of the method. These executions are more promising of ending up in a successful specialization as they are already targetted to a more specific set of candidates. On the other hand the recursive executions might lead to a quick overspecialization of the P-Grid in subparts of it. Therefore, we bound the recursion depth up to which the exchange function is called by the value *recmax*. This value has a very strong influence on the global performance of the algorithm, as we will see later.

So far, we have only considered the construction process of the access structure itself. At the leaf level the peers need also to know the data items, respectively the peers storing those data items, that correspond to their responsibility. As many peers can be responsible for the same key the general problem is of how to find all those peers in case of an update. Different strategies are possible:

- Randomly performing depth first searches for peers responsible for the key multiple times and propagating the update to them
- Performing breadth first searches for peers responsible for the key once and propagating the update to them
- Creating a list of buddies for each peer, i.e. other peers that share the same key, and propagate the update to all buddies.

We will not give the detailed algorithms here as they are quite obvious. But in Section 5 we will identify by using simulations, which is the most efficient method.

4 Analysis of Search Performance

We want to analyze the question of how probable it is to find a peer that is responsible for a specific search key starting the search at an arbitrary peer. This analysis allows to give rough estimates on the sizing of the P-Grid parameters that are required to achieve a desired search reliability in a given setting. We perform the analysis for an idealized situation, where for all peers the parameters of the P-Grid, like keylength and number of peers responsible for the same key, are uniformly distributed. Though such a distribution will not occur in practice it gives a good estimation the quantitative nature of a P-Grid.

The following parameters determine the problem. The number of peers N and the number of data objects each peer can store d_{peer} determine the total number of data objects that can be stored in the network as $d_{global} = N * d_{peer}$. The size of a reference r and the amount of space each peer is willing to make available for indexing purposes s_{peer} determines the possible number of references that can be stored at each peer $i_{peer} = \frac{s_{peer}}{r}$.

Now we determine the number of entries required for a certain grid organisation. The length of a key required to differentiate data items located at different peers is given by

$$k \geq \log_2 \frac{d_{global}}{i_{leaf}} \quad (1)$$

where i_{leaf} is the number of references to data items each peer stores at the leaf level.

Then the total number of index entries stored at a peer is given by $i_{leaf} + k * refmax$, where $refmax$ is the multiplicity of references used to build the grid structure. Thus we obtain the constraint

$$i_{leaf} + k * refmax \leq i_{peer} \quad (2)$$

which determines the value of i_{leaf} . In order to allow at the lowest level of the grid the support for $refmax$ alternative peers, references to data items need to be replicated with a factor of $refmax$. This is only possible if

$$\frac{d_{global}}{i_{leaf}} * refmax \leq N \quad (3)$$

i.e. there must be sufficiently high numbers of peers available, such that each interval at the lowest grid level is supported by at least $refmax$ peers.

Given a constant probability p that a peer is online we are now interested in the question what is the probability of performing a successful search for a peer that is responsible for the query key. In the worst case at each level of the grid a new peer needs to be contacted, that is selected out of the available references. At each level of the grid, the probability of reaching a peer at the next level is $1 - (1 - p)^{refmax}$, i.e. one minus the probability that all $refmax$ referenced peers are offline. Since the search tree is of depth k , then the probability of performing a successful search for a key is

$$(1 - (1 - p)^{refmax})^k \quad (4)$$

We give now an example, to illustrate what a P-Grid would cost in terms of space for a practical setting.

Example. Let us consider the setting P2P file sharing as it currently is found with Gnutella. We use some rough estimates of the actual parameters that are observed for this application. Assume that $d_{global} = 10^7$ data objects (files) exist, that a reference costs at most $r = 10$ Bytes of storage (the path plus the IP address) and that every peer is willing to provide $s_{local} = 10^5$ Bytes of storage for indexing (which is in fact far less than the size of an average media file). Furthermore we assume that peers are online with probability 30%.

Let us now analyze how large a community for supporting the 10^7 files must be in order to ensure that search requests for files are answered with a probability of 99%.

Each peer can store at most $i_{peer} = 10^5$ references. If we "guess" a value of $i_{leaf} = 10^4 - 200$, we see that inequality (1) is satisfied for a value of $k = 10$. For a value of $refmax = 20$ then, according to (4),

the probability of successfully finding a peer for a given key is larger than 99%. The storage required is due to our good initial guess exactly $i_{peer} = 10^5$. The number of peers required to support this grid has to be according to inequality (3) larger than 20409. This is a very reasonable number compared to the size of the actual Gnutella user community.

5 Simulation

For performing simulations we have implemented the algorithm for constructing P-Grids using the mathematical computing package Mathematica (www.wolfram.com). With these simulations we intend to obtain results on the following questions.

- How many communications in terms of executing the exchange algorithm are required for building a P-Grid ?
- What is the influence of the recursion factor *recmax* in the exchange algorithm on the efficiency of the P-Grid construction ?
- Is the resulting structure reasonably well balanced with respect to distribution path lengths and number of replicas per path ?
- How reliable can data be found using the P-Grid ?

First we analyze the convergence speed when the P-Grid is constructed. In this simulation we varied the number of peers from 200 to 1000. The peers meet randomly pairwise and execute the exchange function. The path length was bounded by $maxlength = 6$. As a criterion for a P-Grid to be built we assumed that the average length of the keys that the peers are responsible for reaches a certain threshold t , i.e. $\frac{1}{N} \sum_{a \in P} length(path(a)) < t$. In the following simulation the number of calls to the exchange function were counted till an average path length of 5.94 (99% of $maxlength$) was reached. The simulation was performed with a recursion depth *recmax* of 0 and 2. The value of *refmax* was set to 1, i.e. only one reference to another peer is stored. This influences only the performance in the case where $recmax > 0$.

The results indicate that a linear relationship exists between the number of peers (N) and the total number of communications (e) needed in building the P-Grid. As a consequence, every peer has to perform on average a constant number of exchanges to reach its maximal path length independent of the total number of peers involved.

N	recmax = 0		recmax=2	
	e	$\frac{e}{N}$	e	$\frac{e}{N}$
200	15942	79.71	4937	24.68
400	27632	69.08	10383	25.95
600	43435	72.39	15228	25.38
800	59212	74.01	18580	23.22
1000	74619	74.61	25162	25.16

The next simulation shows how the choice of a maximal path length $maxlength$ influences the number of exchanges $e_{maxlength}$ required. The simulation is done for $N = 500$ peers. The results indicate that the number of communication grows exponentially in the path length (to a basis 2) when no recursion is used. With a recursion bound $recmax = 2$ the convergence speeds up substantially.

<i>maxlength</i>	<i>recmax</i> = 0			<i>recmax</i> = 2		
	<i>e</i> _{<i>maxlength</i>}	$\frac{e_{maxlength}}{N}$	$\frac{e_{maxlength}}{e_{maxlength-1}}$	<i>e</i> _{<i>maxlength</i>}	$\frac{e_{maxlength}}{N}$	$\frac{e_{maxlength}}{e_{maxlength-1}}$
2	4893	9.78		5590	11.18	
3	9780	19.56	1.998	7289	14.57	1.303
4	18071	36.14	1.847	8215	16.43	1.127
5	35526	71.05	1.965	13298	26.59	1.618
6	72657	145.31	2.045	17797	35.59	1.338
7	171770	343.54	2.364	27998	55.99	1.573

The following table shows that the recursion depth *recmax* has substantial influence on the convergence speed. When using recursive calls to the exchange function the probability that a random meeting leads to a successful exchange in the P-Grid increases. However, if recursion is not constrained this can lead to negative effects as the peers tend to overspecialize quickly. The result shows that for 500 peers and maximal path length 6 the optimal recursion depth limit is 2.

<i>recmax</i>	<i>e</i>	$\frac{e}{N}$
0	35436	70.87
1	15377	30.75
2	12735	25.47
3	16595	33.19
4	18956	37.91
5	22426	44.85
6	25130	50.26

If *refmax* > 1, i.e. peers maintain more than one reference to other peers at each level, i.e. there exist more possibilities to make recursive calls. Thus if *recmax* > 0 this should have an influence on the number of exchanges that are performed when constructing the P-Grid. Note that this additional effort is rewarded by a higher density of the P-Grid.

We analyzed this with the following simulation with *N* = 1000 peers, a recursion depth limit *recmax* = 2 and using maximal path length of 6.

<i>refmax</i>	<i>e</i>	$\frac{e}{N}$
1	25285	25.28
2	39209	39.20
3	72130	72.13
4	125727	125.72

As one can see the number of exchanges grows exponentially, which is undesirable. In fact, this turned out to be a weakness in the algorithm we proposed. However, there exists a simple way to fix this. One limits the number of referenced peers with which exchanges are made throughout recursion to a low number. Then the results become very stable as the following simulation with the modified algorithm shows, where recursive calls are only made to 2 randomly selected referenced peers.

<i>refmax</i>	<i>e</i>	$\frac{e}{N}$
1	23826	23.826
2	37689	37.689
3	40961	40.961
4	43914	43.914

The following simulations are based on a configuration that is similar to the one resulting from our analysis in Section 4. This confirms that the algorithms scale well for realistic parameter settings. We are using 20000 peers that build a P-Grid with binary keys of maximal length 10. The maximal number of references *refmax* at each level is limited to 20. The online probability of peers is 30%.

Building a P-Grid of that size within a simulation environment is starting to consume considerable resources. Within approximately 10 hours of running time on a Pentium III processor the P-Grid built up

to an average depth of 9.43. During that time 1250743 exchanges among peers were taking place, i.e. 62 per peer. Based on the P-Grid constructed this way we made the following analyses.

A first question of interest is, how balanced the P-Grid is with respect to the distribution of keys, i.e. how many peers are responsible for the same keys. We will call such peers replicas. A simple, intuitive argument shows that the exchange function inherently tends to balance the distribution of keys. Let us look at the top level, i.e. at the decision whether the key of a peer starts with a 0 or a 1. A peer will decide upon this when it meets the first time another peer that has already taken this decision or also needs to decide on the top level, and he will decide for the opposite value. Thus, if there exists an imbalance such that one of the two values dominates this leads immediately to a compensation effect that the values occurring with less frequency will more likely be chosen. In that way the algorithm is self-stabilizing.

The following Figure 2 shows the distribution of the number of peers responsible for the same key. The x-axis indicates the replication factor and the y-axis the number of peers that have this replication factor. The average number of replicas for a peer is 19.46.

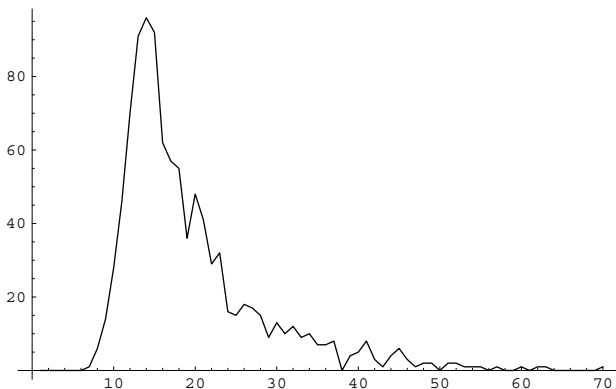


Figure 2: Replica distribution

Next we would like to experimentally confirm the analysis on the probability of finding a peer that is responsible for a given key. We made a simulation where we searched 10000 times for a random key of length 9. Remember that only 30% of the peers are online. In 99.97% of the cases the search was successful and a search required on average 5.5576 messages among peers. We were counting as messages the successful calls of the query operation to another peer. This shows that searches can be performed reliably.

Now we turn to the question of how reliably updates can be performed. The problem with an update, in contrast to a search, is that we have to find all replicas of a path, not just one. Therefore we analyze how efficiently a large fraction of all replicas can be found. We compare three approaches.

1. Repeated depth first searches
2. Repeated depth first searches including all buddies that have been identified throughout index construction
3. Repeated breadth first searches

We repeatedly searched for a random key of length 9 and then computed the fraction of replicas identified to the actual number of existing replicas. Figure 3 shows the result. The x-axis shows the number of messages used in the insertion process, and the y-axis the percentage of successfully identified replicas. Without giving the detailed simulation setup, one can see that clearly the strategy of using breadth first searches is by far superior, while the two other methods perform comparably.

One can see also from these simulations, that for achieving a high update reliability a fairly large number of messages are required (in the order of hundreds per replica to be updated). So this approach is acceptable

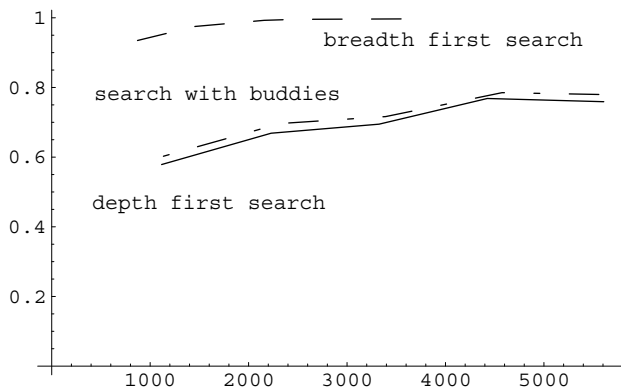


Figure 3: Finding all replicas

in the case where updates are rare as compared to queries. However, the relevant problem is not so much to achieve high reliability in keeping the replicas consistent, but rather high reliability in obtaining correct query results. Thus we can follow a different approach. We update a sufficiently high number of replicas using fewer messages, and then use repeated query operations, till a query result is multiply confirmed. Obviously, if more than half of the replicas are correct by repeating queries arbitrarily high reliability can be achieved. Thus, by proceeding in this manner we increase slightly the query cost and in turn reduce drastically the update cost. There is another factor that is helpful in that context. Not all replicas are as likely to be found. This implies that replicas that are found during updates are also more likely to be found during queries.

The following simulations illustrate the tradeoff among update and query cost and indicate what advantageous combinations of update and query strategies exist. 100 updates were performed and each updated data item was searched 10 times, thus 1000 queries were performed in each configuration. Each update was performed by a breadth first search where at each level *recbreadth* references are followed. The breadth first search was repeated during update *repetition* times. The *successrate* is the fraction of successfully answered queries after update. The cost is again in terms of number of messages.

with repetitive search					
	recbreadth	repetition	successrate	query cost	insertion cost
	2	1	1	137	78
	2	2	1	34	147
	2	3	1	17	224
	3	1	1	112	637
	3	2	1	13	1434
	3	3	1	13	2086
without repetitive search					
	2	1	0.65	5.5	72
	2	2	0.85	5.6	145
	2	3	0.89	5.4	212
	3	1	0.95	5.5	734
	3	2	0.98	5.5	1363
	3	3	0.994	5.4	2080

The main result here is that the approach of using repeated searches to achieve query reliability pays off dramatically. The configuration (*recbreadth* = 3, *repetition* = 3) without using repeated search, which achieves only 99,4% reliability, would require at least a ratio of 160 queries per update in order reach the

break-even point with the configuration ($recbreadth = 2, repetition = 3$) with using repeated search, but which offers practically 100% reliability.

6 Discussion

This paper introduced P-Grid a first step towards developing scalable, robust and self-organizing access structure for Peer-To-Peer information systems. In order to better understand the effectiveness of a P-Grid we can compare it to the use of centralized replicated servers. Assume D is the number of data items and N the number of peers. For storage we consider the number of references that need to be stored at the nodes ignoring any cost for local indexing. For querying we consider the number of messages exchanged at the nodes assuming that each node creates a constant number of queries per time unit. Then a solution using centralized (possibly replicated) servers compares to the P-Grid as follows.

	P-Grid	Central Server
Storage	peers: $O(\log D)$	server: $O(D)$ client: constant
Query	peers: $O(\log N)$	server: $O(N)$ client: constant

One can see that both storage and communication cost scale well for the P-Grid. For a server solution in particular the linear growth of communication cost in terms of number of clients is to be considered critical as servers tend to become bottlenecks. This shows that, besides other considerations motivating the use of P-Grids like decentralization or robustness, also scalability is one of the benefits.

The emphasis of this paper was on distributed, randomized algorithms which allow the peers to cooperatively partition the search space in an efficient way. The approach in this paper is limited to uniform data distributions.

For uniformly distributed key values the P-Grid can be immediately applied. For prefix search on text the algorithm can be adapted by extending the $\{0, 1\}$ alphabet. This would allow to support trie search structures. However a worthwhile investigation would be to extend the method by adapting more sophisticated text search structures, like in [3, 4].

An obvious continuation of this research is to develop P-Grids that can adapt to skewed data distributions. To that extent throughout the construction process the actual data distribution needs to be taken into account and the structures have to continuously to adapt to updates. However, we see no principal difficulties in implementing such an approach.

Another natural extension of the approach would be to take other known parameters, like known reliability of peers, knowledge on the network topology or knowledge on query distribution into account throughout P-Grid construction and updates. To achieve load balancing a computational economy can be imposed, as already investigated for distributed data mangement in [10, 5].

We see the P-Grid, as it is presented in this paper, as a first representative of access structures for Peer-To-Peer information systems, for which we expect to see in the future many variations, that are adapted to the specific requirements of various Peer-To-Peer application domains.

7 Acknowledgements

I would like to thank Manfred Hauswirth for carefully reading and commenting the manuscript. I would also like to thank Magdalena Puceva and Rachid Guerraoui for many helpful discussions. This work also greatly benefited from the inspiring working environment that is provided by the newly founded Communication Systems Department at EPFL.

References

- [1] E. Adar and B. A. Huberman: *Free riding on Gnutella* Technical report, Xerox PARC, 10 Aug. 2000.
- [2] D. Clark. *Face-to-Face with Peer-to-Peer Networking*. IEEE Computer, January 2001.
- [3] Y. Chen, K. Aberer: *Layered Index Structures in Document Database Systems* Proceedings of the 1998 ACM CIKM International Conference on Information and Knowledge Management, Bethesda, Maryland, USA, November 3-7, ACM Press, p. 406-413, 1998.
- [4] Y. Chen, K. Aberer: *Combining Pat-Trees and Signature Files for Query Evaluation in Document Databases* DEXA 99, Florence, Italy.
- [5] D.F. Ferguson, C. Nikolaou and Y. Yemini *An Economy for Managing Replicated Data in Autonomous Decentralized Systems* Proc. Int. Symp. on Autonomous Decentralized Sys. (ISADS'93), Kawasaki, Japan, 1993.
- [6] T. Johnson, P. Krishna *Lazy Updates for Distributed Search Structure* SIGMOD Conference 1993: 337-346
- [7] B. Kröll, P. Widmayer *Distributing a Search Tree Among a Growing Number of Processors*. ACM SIGMOD Conference 1994: 265-276
- [8] B. Kröll, P. Widmayer *Balanced Distributed Search Trees Do Not Exist* WADS 1995: 50-61
- [9] W. Litwin, M. Neimat, D. A. Schneider *RP*: A Family of Order Preserving Scalable Distributed Data Structures*. VLDB 1994: 342-353
- [10] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, Jeff Sidell, Carl Staelin, Andrew Yu: *Mariposa A Wide-Area Distributed Database System* VLDB Journal 5(1): 48-63, 1996.
- [11] R. Vingralek, Y. Breitbart, G. Weikum *SNOWBALL: Scalable Storage on Networks of Workstations with Balanced Load* Distributed and Parallel Databases Vol 6(2), Kluwer Academic Publishers, 1998
- [12] H. Yokota, Y. Kanemasa, J. Miyazaki *Fat-Btree: An Update-Conscious Parallel Directory Structure* ICDE 1999: 448-457