The Quest for Balancing Peer Load in Structured Peer-to-Peer Systems[*]

Karl Aberer, Anwitaman Datta, Manfred Hauswirth

Distributed Information Systems Laboratory

Ecole Polytechnique Fédérale de Lausanne (EPFL)

CH-1015 Lausanne, Switzerland

Abstract

Structured peer-to-peer (P2P) systems are considered as the next generation application backbone on the Internet. An important problem of these systems is load balancing in the presence of non-uniform data distributions. In this paper we propose a completely decentralized mechanism that in parallel addresses a local and a global load balancing problem: (1) balancing the storage load uniformly among peers participating in the network and (2) uniformly replicating different data items in the network while optimally exploiting existing storage capacity. Our approach is based on the P-Grid P2P system which is our variant of a structured P2P network. Problem (1) is solved by directly adapting the search structure to the data distribution. This may result in an unbalanced search structure, but we will show that the expected search cost in P-Grid in number of messages remains logarithmic under all circumstances. Problem (2) is solved by a dynamic, reactive balancing method based on sampling the P-Grid structure. Through simulations we show that our solution provides a scalable approach to these load balancing problems. Finally we discuss issues that had to be addressed beyond the theoretical aspects when implementing our approach as part of a practical P2P system.

Keywords: C.2 [**Communication/Networking and Information Technology**]: Distributed Systems—*Peer-to-Peer Systems*; C.2.4 [**Distributed Systems**]: Distributed Databases—*Peer-to-Peer, Load Balancing*; C.2.4 [**Distributed Systems**]: Distributed Applications—*Peer-to-Peer, Load Balancing*; E.1 [**Data Structures**]: Distributed data structures—*Distributed Hash Tables*; H.3 [**Information Storage and Retrieval**]: Indexing Methods—*Load Balancing, Distributed Hash Tables*

[*]The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

1 Introduction and motivation

Structured peer-to-peer (P2P) systems are considered as the next generation application backbone on the Internet. They solve key-based lookup of data, a basic problem of data access, using a decentralized approach. Most of the current systems of this class are based on a variant of the distributed hash table (DHT) approach [20].

A practical problem encountered by these systems is load balancing. Load balancing is critical to support high scalability, availability, accessibility, and throughput. Poor load balancing may in fact gradually transform a P2P system into a backbone-based system as it was observed for Gnutella [7].

For systems supporting equality-based lookup of data only, the problem of non-uniform workloads may be circumvented by applying randomized hash functions to the data keys, thus uniformly distributing workload, both for storage and query answering. In combination with using balanced search structures, i.e., balanced distributed search trees, such an approach leads to uniform load distribution among the participating peers. However, this approach is limited if further semantics of the data keys is exploited, for example, in the simplest case when the ordering of data keys is used in lookups to support prefix or range queries.

Another aspect of load balancing is uniform replication of data to support uniform availability. Typically this problem is tackled in current structured P2P systems by controlled replication, where a globally constant replication factor is assumed. Besides introducing global knowledge into the systems, which is undesirable from the viewpoint of decentralization and peer autonomy, this approach also lacks the ability to adaptively exploit existing storage resources in an optimal manner.

In this paper we will introduce an approach that tackles the two load balancing problems identified for structured P2P systems from the perspective of storage load balancing: we aim at distributing storage load uniformly among peers and to exploit existing storage capacity optimally by uniformly replicating data until the available storage is used optimally. The approach relies exclusively on completely decentralized and thus self-organized methods. Randomization is an essential element in the proposed solution. By solving this important subproblem of load balancing (we are, for example, in this paper not considering non-uniform query loads or non-uniform distribution of peer resources) we demonstrate the feasibility of a complex load balancing strategy in a completely decentralized setting and provide a working solution applicable to many practical settings.

Our approach is based on a fundamental observation that we make on distributed hash tables. The cost of lookup measured in terms of messages used remains logarithmic even if the underlying search tree structure is not balanced. Thus, a balanced distributed hash table is a sufficient condition for scalable search, but not a necessary one. This result relies essentially on the randomization inherent in the routing tables used to implement distributed hash tables. We exploit this property by adapting the structure of the DHT structure such that storage load is balanced also for non-uniform data key distributions. The resulting DHT may be arbitrarily unbalanced. This will solve the problem of balancing storage load among peers. In addition, allowing for unbalanced search trees, we can maintain the ordering of keys within the tree structures. Thus prefix and range searches can be directly supported, which constitutes an important element in exploiting semantics of search keys for queries.

This approach is different, both from standard DHT approaches and classical database indexing. In standard DHT approaches uniform hashing leads to the construction of approximately balanced search structures and thus to an upper bound on search cost derived from the depth of the search tree underlying the DHT construction. However, any ordering among the keys is lost. In database indexing, balanced data structures such as B-Trees are used for indexing to provide efficient search and maintain key ordering. However, adopting distributed versions of such search structures appears to be unsuitable for P2P environments, since in the worst case operations could affect the whole network (e.g., splitting the root of a search tree).

For uniform replication we introduce an adaptive mechanism to globally balance workload. Different to storage load, peers cannot detect non-uniform replication of data locally. Therefore we introduce a sampling-based method to detect imbalance and to dynamically adapt replication. Thus data will be dynamically replicated while peers aim at using their storage capacity optimally. An important aspect is the mutual dependency among storage load balancing and uniform replication. When peers attempt to locally balance their storage load they may compromise globally uniform replication. Therefore a main contribution of the paper is to achieve both load balancing goals in conjunction.

Our approach for load balancing is based on P-Grid [1, 6], our variant of a DHT. It essentially differs from other approaches in the way peers can dynamically adopt and decide during operation which search space they are responsible for, independent of their physical identity. In contrast to the standard assumption in structured P2P systems, that peers adopt their identity and thus the search space they are responsible for before they enter the network, in P-Grid peers can decide dynamically during interactions with other peers which data they become responsible for. Thus peers also can change their "search space assignment" in order to perform load balancing operations.

P2P systems are complex computational systems. In order to establish properties of such systems one has to rely on various methods. Ideally analytical results allow to establish their properties in full generality. However, for realistic systems such results are difficult to obtain and thus analytical results frequently have to be limited to simplified situations.

In this paper we will establish a key result on efficient search in unbalanced P-Grids theoretically. Also some of the load balancing heuristics that we introduced will be generalizations of algorithms motivated by analytical models for simplified situations. The general validity of our algorithms will, however, be verified by simulation results due to the complexity of the realistic target setting. Finally, we will also take the step from theoretically sound algorithms to practical implementations which will uncover some of technical issues to be encountered upon implementing a real-world structured P2P system.

Only few approaches on load balancing in structured peer-to-peer systems are currently reported in the literature. They typically start from standard DHT approaches and introduce algorithms and auxiliary data structures a-posteriori to remedy load imbalance. Though a complete comparison is not feasible at this point we believe that such approaches run the risk to sacrifice the efficiency of the original access structure in the presence of frequent changes in the data distribution as all structural adaptations are additive in nature.

Another class of approaches tries to tackle the problem through the application of economic models. These approaches are based on some form of virtual currency that can be earned through service/resource provision and is spent if services/resources are consumed. We consider this class of approaches as complementary to our work and we envisage to employ economic models in our approach for the case of non-uniform peer resources and non-cooperative environments.

Conceptually closest to our approach we consider Freenet [9, 10]. It exhibits also dynamic load balancing strategies, efficient search and provides a practical implementation. The main difference is the lack of any theoretical foundations for the heuristics used in Freenet and potential performance problems that (probably) only can be solved by non-scalable resource consumption [5].

This paper is structured as follows: Section 2 gives the basic theoretic definitions of P-Grid's structure. Section 3 proves that P-Grid will work efficiently even for skewed data distributions and provides a worst-case scenario and bound. Section 4 defines the problems of storage load and replication balancing theoretically and gives a motivating example to explain the problems and our goals. Section 5 describes the storage load balancing algorithm and Section 6 presents the algorithm to achieve uniform replica distribution. Section 7 briefly describes P-Grid's basic operations and Section 8 then puts everything together and discusses the results of our simulations for the integrated algorithm which justify our claims. Since P-Grid exists as software we provide some of our experiences in mapping our theoretical approaches onto a running implementation in Section 9. Section 10 relates our approach to other approaches in the area and we finally draw our conclusions in Section 11.

2 The P-Grid data structure

P-Grid is a distributed data structure based on the principles of distributed hash tables (DHT) [20]. As any DHT approach P-Grid is based on the idea of associating peers with data keys from a key space $\mathcal{K}$. Without constraining general applicability we will only consider binary keys in the following. In contrast to other DHT approaches we do not impose a fixed or maximal length on the keys, i.e., we assume $\mathcal{K} = \{0, 1\}^*$.

In the P-Grid structure each peer $p \in Peers$ is associated with a binary key from $\mathcal{K}$. We denote this key by $path(p)$ and will call it the path of the peer. This key determines which data keys the peer has to manage, i.e., the keys in $\mathcal{K}$ that have $path(p)$ as prefix. In particular the peer has to store them. In order to ensure that the complete search space is covered by peers we require that the set of peers' keys is *complete*.

The set of peers' keys is complete if for every prefix s_{pre} of the path of a peer p there exists a peer p' such that $path(p') = s_{pre}$ or there exist peers p_0 and p_1 such that $s_{pre}0$ is a prefix of $path(p_0)$ and $s_{pre}1$ is a prefix of $path(p_1)$. Naturally one of the two peers p_0 and p_1 will be p itself in that case. Completeness needs to be guaranteed by the algorithms that construct and maintain the P-Grid data structure and will be introduced in Section 5.

We do not exclude the situation where the path of one peer is a prefix of the path of another peer. This situation will occur during the construction and reorganization of a P-Grid. However, ideally this situation is avoided and any algorithm for maintaining a P-Grid should eventually converge to a state where the P-Grid is *prefix-free*, i.e., for peers p_0 and p_1 we have $path(p_0) \not\subset path(p_1) \wedge path(p_1) \not\subset path(p_0)$, where $s \subset s'$ denotes the prefix relationship among strings s and s'.

We also allow multiple peers to share the same paths, in that case we call the peers replicas. The number of peers that share the same path is called the *replication factor* of the path. Replication is important to support redundancy and thus robustness of a P-Grid in case of failures and to distribute workload when searching in a P-Grid.

For enabling searches peers maintain *routing tables* which are an essential constituent of the P-Grid structure. The routing tables are defined as (partial) functions $ref : Peers \times N \to \{Peers\}$ with the properties

1. $ref(p,l)$ is defined for all $p \in Peers$ and $l \in N$ with $1 \leq l \leq |path(p)|$
2. $ref(p,l) \subseteq Peers_{s_1 s_2 \ldots s_{l-1}(1-s_l)}$ with $path(p) = s_1 s_2 \ldots s_{l-1} s_l \ldots s_k, k \geq l$

where $Peers_t = \{p \in Peers | t \subseteq path(p)\}$ for $t \in \mathcal{K}$.

For the same association of peers with paths, different P-Grids can be obtained depending on the choice of $ref(p,l)$. An important observation of which we will make use later relates to the fact that in this definition the choices of the sets $ref(p,l)$ are independent for different $p \in Peers$ and $l \in N$.

Having multiple references at each level l again is necessary to guarantee robustness of the data structure. We denote by $refmax$ the maximum number of references maintained at each level. The search algorithm for locating data keys indexed by a P-Grid is defined as follows: Each peer $p \in Peers$ is associated with a location $loc(p)$ (in the network). Searches can start at any peer. Peer p knows the locations of the peers referenced by $ref(p,l)$, but not of other peers. Thus the function $ref(p,l)$ provides the necessary routing information to forward search requests to other peers in case the searched key does not match the key space the peer is responsible for. Let $t \in \mathcal{K}$ be the searched data key and let the search start at $p \in P$. Then the following recursive algorithm performs the basic search.

```
$search(t, loc(p))$ := **if** $path(p) \subseteq t$ **then** $return(loc(p))$
                       **else**
                           determine maximal $l$ such that $t_1 \ldots t_{l-1}(1-t_l) \subseteq path(p)$;
                           $r$ = randomly selected element from $ref(p,l)$; $search(t, loc(r))$;
```

The algorithm $search(t, loc(p))$ always terminates successfully: due to the definition of ref the function $search$ will always find the location of a peer at which the search can continue (use of completeness). With each invocation of $search(t, loc(p))$ the length of the common prefix of $path(p)$ and t increases at least by one. Therefore the algorithm always terminates.

In case of an unreliable network it may occur that a search cannot continue since the peer r selected from the routing table is not available. Then alternative peers can be selected from the routing table to continue the search.

We illustrate the P-Grid data structure by means of a simple example. In Figure 1 we show 8 peers $P1, \ldots, P8$. The paths of the peers are indicated by their positions in the (virtual) binary tree. In order not to overload the figure we show examples of routing table entries only for peer P_3 at level 1 and peer P_4 at level 2.

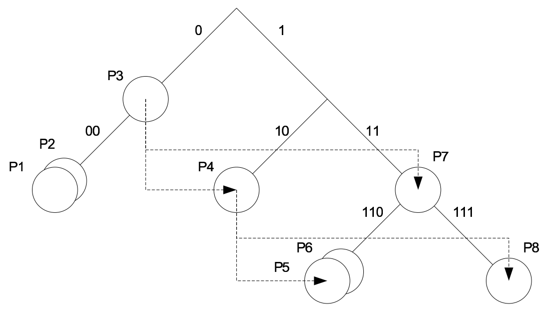


Figure 1: P-Grid data structure example

We can make the following observations:

- The P-Grid is not balanced (e.g., peers $P1$ and $P8$ have paths of different length)
- The P-Grid is complete. For all prefixes of peer paths there exist continuations of the paths for 0 and 1, with the exception of the path 0. The path 0 is however associated with peer $P3$ so that the corresponding subspace is also covered
- The P-Grid uses replication. For certain paths (00, 110) multiple peers are associated with, therefore data pertaining to these paths will be replicated (replica consistency is maintained using [11]).
- The P-Grid is not prefix-free, since, e.g., the path of peer $P7$ is a prefix of the path of peer $P8$.

For illustration we have also included references for some selected peers. For peer $P3$ we have included 2 references at level 1, which refer to peers with paths starting with bit 1. For peer $P4$ we have included two references at level 2, which refer to peers with paths starting with 11. Using these references we can illustrate a sample search. Assume a search for path 111 is submitted to peer $P3$. Since the path of $P3$ does not share a prefix with the search key $P3$ uses its references at level 1 to forward a search message, let us assume to peer $P4$. The path of peer $P4$ shares the first bit with the searched key, thus it uses its references at level 2 to forward a search message. Assume it selects peer $P8$. In this case the search terminates successfully after two messages have been sent.

3 Efficient Search in Unbalanced P-Grids

As we do not require that the P-Grid is balanced, i.e., that all paths associated with peers are of equal length, search cost may be non-logarithmic in the number of messages needed to locate a peer holding the searched data item. In this section we will show that due to the probabilistic nature of our approach this does not pose a problem. We show that the expected cost of searches measured in the number of messages required to perform the search remains logarithmic, independent of how the P-Grid is structured. We will see that even if peers' paths are linear in length in the total number of peers participating in the network, the expected search cost remains logarithmic.

For simplicity but without constraining general applicability we provide this result only for a special class of P-Grids, namely prefix-free P-Grids without replication of paths and of references. We will show by simulations that the result also applies in the more general cases (see Section 8.6). For the restricted class of prefix-free P-Grids without replication we can identify peers by their path unambiguously. In the following when we refer to peers we therefore do this directly via their paths, i.e., by using the set $\mathcal{S} = \{path(p), p \in Peers\} \subseteq \mathcal{K}$.

The set of P-Grids that can be constructed based on the set of peer identifiers $\mathcal{S}$ we denote by $\mathcal{P}_{\mathcal{S}}$. Different P-Grids are distinguished by their different choice of the references, which we denote by $ref_{\mathcal{S}}^{P}$ for a P-Grid $P \in \mathcal{P}_{\mathcal{S}}$. Since no replication of references is used, we assume that the function $ref_{\mathcal{S}}^{P}$ has the signature $ref_{\mathcal{S}}^{P} : \mathcal{S} \times N \to \mathcal{S}$, i.e., it is single-valued.

In a distributed environment the relevant cost measure for an algorithm is the number of messages that are exchanged. Each invocation of $search$ corresponds to forwarding the search task to a different peer, i.e., a message. Therefore we define the search cost in a P-Grid $P \in \mathcal{P}_{\mathcal{S}}$ for a data key $t \in \mathcal{S}$ starting at $s \in \mathcal{S}$ as the number of invocations of the function $search(t, loc(s))$. We denote this cost by $\sigma_P^s(t)$.

The definition of P-Grid does not exclude the case where the length of the paths is up to linear in the size of $\mathcal{S}$. Therefore searches can require a linear number of messages in the worst case which would make the access structure non-scalable. In the following we show that the expected average search cost is logarithmic, however.

Theorem 2. The expected search cost $\sigma_P^s(t)$ for the search of a specific key $t \in \mathcal{S}$ using a P-Grid $P \in \mathcal{P}_{\mathcal{S}}$, that is randomly selected among all possible P-Grids, starting at a randomly selected peer $s \in \mathcal{S}$ is less than $\log(|\mathcal{S}|)$.

A formal proof of this theorem is given in [2]. Here we limit us to provide the intuition which is underlying the proof.

For showing the claim we analyze a search process. We consider one specific peer associated by its key with a leaf node of the search tree. Assume a search for this key starts at some randomly selected peer. We determine the number of messages required to resolve all bits of the query correctly. If $n_0 = |\mathcal{S}|$ is the total number of peers and n_1 is the number of peers not matching the first bit of the query and if we assume that the initial peer is randomly selected resolving the first query bit will require a message with probability $\frac{n_1}{n_0}$. Now that the first bit is resolved the query is to be processed by one of the $n_0 - n_1$ peers with the first bit matching. At this point we have to assume that this peer is uniformly randomly selected among all peers matching in the first bit. This assumption is satisfied in the theorem, since we assume that the P-Grid is randomly selected among all possible P-Grids. Therefore the routing table entries are uniformly randomly selected among all peers that qualify since the entries of routing tables at different levels are independently chosen.

If we assume that the peer matching the first bit is uniformly randomly selected, we have n_2 peers among those $n_0 - n_1$ peers matching the first bit that do not match the second bit. Therefore with probability $\frac{n_2}{n_0 - n_1}$ a message will be required to resolve the second bit. Now the argument continues analogously till all bits are resolved. The process is illustrated in Figure 2. Note that the length of the key k is completely independent of the number of peers, thus the argument we have given applies for all $1 \leq k \leq |\mathcal{S}|$.

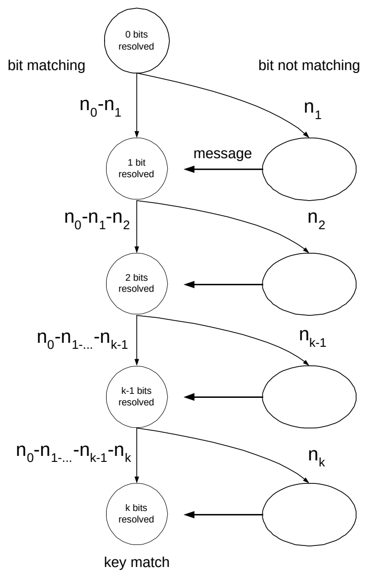


Figure 2: Search process

Adding up the expected number of messages for resolving each bit 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 a sequence of positive numbers $n_1, \ldots, n_k$ with $\sum_{i=1}^{k} n_i = n_0 - 1$. We can bound this sum as follows:

$$\begin{aligned}
\sum_{i=1}^{k} \frac{n_i}{n_0 - \ldots - n_{i-1}} &= \sum_{i=1}^{k} \int_{n_0-\ldots-n_{i-1}-n_i}^{n_0-\ldots-n_{i-1}} \frac{1}{n_0 - \ldots - n_{i-1}} dx \\
&\leq \sum_{i=1}^{k} \int_{n_0-\ldots-n_{i-1}-n_i}^{n_0-\ldots-n_{i-1}} \frac{1}{x} dx \\
&= \int_{1}^{n_0} \frac{1}{x} dx = \log n_0
\end{aligned}$$

This results in a bound $\sum_{i=1}^{k} \frac{n_i}{n_0-\ldots-n_{i-1}} < \log n_0$ as stated in the theorem.

Another result shows that also in the case where the search tree is not of logarithmic depth, the number of P-Grids for which a logarithmic search cost is not achieved is extremely small.

Theorem 3. The probability that a search in a P-Grid $P \in \mathcal{P}_{\mathcal{S}}$ for a key $t \in \mathcal{S}$ of length d starting at a randomly selected peer $s \in \mathcal{S}$ does not succeed after k steps is smaller than $\frac{\log(n)^{k-1}}{(k-1)!}$.

The detailed proof of this theorem and a discussion of some of its consequences can be found in [2].

Putting everything together we have shown that the search cost measured as communication cost is always small on average, independent of the specific shape of the P-Grid. Furthermore one can show that the cases where the cost deviates from the average are rare. Thus we can assume in the following that P-Grids may be unbalanced without affecting search performance and we will exploit this property in order to adapt the shape of the P-Grid to the data distribution for storage load balancing purposes .

4 The Problems of Load Balancing

In the following we introduce the load balancing issues that we will be considering for P-Grid in the subsequent sections. Basically any DHT-based access structure faces similar problems. Given a P-Grid structure we can identify two main issues to be addressed:

1. *Storage load balancing*. Given a peer $p \in Peers$ we define its storage load $load(p)$ as the number of data keys that pertain to the search space associated with the peer through its path. Balancing $load(p)$ among different peers is important to avoid throughput bottlenecks by overloading a small set of peers while under-utilizing the rest of the system. The resulting load balancing problem is a *local* load balancing problem, as peers can recognize locally whether they are underloaded or overloaded and thus that load is not distributed uniformly in the system.

2. *Uniform Replication.* Given a path $s \in \mathcal{K}$ we define the replication factor $rep(s)$ as the number of peers p with $path(p) = s$. Balancing $rep(s)$ is important in order to ensure that all subspaces of the key space are covered uniformly by peers and thus data items are uniformly replicated. This is of interest both from the viewpoints of failure resilience in the presence of unreliable peers and of workload balancing for searches and updates. The resulting load balancing problem is a *global* load balancing problem, as peers cannot determine locally how many replicas of their path exist in the network.

We assume that the data keys are in general distributed non-uniformly. This assumption has to be made whenever data keys are indexed that bear "semantics" and are not used purely as identifiers. In the latter case random hash functions can be applied to enforce a uniform key distribution and only trivial search predicates, i.e. equality, can be applied. Examples of data keys bearing semantics are natural language terms or sensory data and typically non-trivial search predicates are applied, such as sub-string or range queries.

If we require storage load balancing the resulting P-Grid structure will be unbalanced, since in regions of the key space where data items are more frequent, more peers will specialize and thus have longer paths. This explains the importance of our result from Section 3. Any method not taking into account data distribution, such as the standard method in DHT approaches, which randomly associates peers with paths, would in general result in non-uniform storage loads.

We will see in the following that methods for (local) storage load balancing and (global) uniform replication can be in conflict because balancing storage load locally may compromise global uniform replication. An important goal of this paper is to demonstrate that it is possible to devise completely decentralized algorithms that can overcome this conflict.

Our requirement that data items are uniformly replicated corresponds to the underlying assumption that the request rates for different data items are uniformly distributed. We will provide solutions to the problem of non-uniform request distribution in the future, based on standard techniques of variable replication (e.g., using the square-root rule [19]) and the results on the combined local and global load balancing presented in this paper.

4.1 Motivating Scenario

In the following we illustrate by means of an elementary example the typical operations performed when constructing and maintaining a P-Grid and the issues arising from attempting to maintain local storage balance and global uniform replication simultaneously. Assume 6 peers that may store up to two data items and initially hold 6 different data items are given as follows:
$[P_1, \epsilon, \{A_1, B_1\}]$ $[P_2, \epsilon, \{B_1, C_1\}]$ $[P_3, \epsilon, \{A_2, B_2\}]$ $[P_4, \epsilon, \{B_2, C_2\}]$ $[P_5, \epsilon, \{C_1, A_1\}]$ $[P_6, \epsilon, \{C_2, A_2\}]$
The states of peers are represented as triples, consisting of the peer identifier, the current path and the data

keys the peer currently holds. We assume that the first two bits of the data items' keys are $pre_{key}(A_i) = 00$, $pre_{key}(B_i) = 10$, $pre_{key}(C_i) = 11, i = 1, 2$. Initially all paths are empty (ϵ), i.e., no P-Grid has been built yet. We describe now a scenario of possible interactions among peers when constructing a P-Grid.

P_1 initiates a P-Grid network N_1. P_2 joins the network by contacting P_1. We assume that whenever at least 1 data item pertaining to a subspace is available a peer attempts to specialize to that subspace.Thus P_1 and P_2 can split the search space and their resulting states are:

$N_1 : [P_1, 0, \{A_1\}], [P_2, 1, \{B_1, C_1\}]$

Independently P_3 starts a P-Grid network N_2 and P_4 joins this network resulting in the states

$N_2 : [P_3, 0, \{A_2\}], [P_4, 1, \{B_2, C_2\}]$

Next P_5 joins the first network N_1 by contacting P_2. Since P_2 is already specialized on path 1, P_5 decides to extend to path 0. The resulting state is then

$N_1 : [P_1, 0, \{A_1\}], [P_2, 1, \{B_1, C_1\}], [P_5, 0, \{A_1\}]$

Then P_6 enters the network N_1 by contacting P_5. Since P_5 has specialized on path 0 P_6 decides to adopt path 1 and P_5 stores the data of P_6 pertaining to path 0.

$N_1 : [P_1, 0, \{A_1\}], [P_2, 1, \{B_1, C_1\}], [P_5, 0, \{A_1, A_2\}], [P_6, 1, \{C_2\}]$

Now P_3 contacts P_1. As a result the two networks N_1 and N_2 merge into a common network N and become a single P-Grid. This shows that P-Grids do not require to start from a single origin, as assumed by standard DHT approaches, but can dynamically merge, similarly to Gnutella networks. Since both P_3 and P_1 have path 0 and still have extra storage space they can replicate their data to increase data availability. This results in a state

$N : [P_1, 0, \{A_1, A_2\}], [P_2, 1, \{B_1, C_1\}], [P_3, 0, \{A_1, A_2\}], [P_4, 1, \{B_2, C_2\}], [P_5, 0, \{A_1, A_2\}], [P_6, 1, \{C_2\}]$

In order to explore the network P_2 contacts P_4. Network exploration serves the purpose of network maintenance and can be compared to the ping/pong protocol used in Gnutella. Since P_2 and P_4 both store data pertaining to path 1 they can now further refine the search space by specializing their paths and exchange their data according to the new paths.

$N : [P_1, 0, \{A_1, A_2\}], [P_2, 10, \{B_1, B_2\}], [P_3, 0, \{A_1, A_2\}], [P_4, 11, \{C_1, C_2\}], [P_5, 0, \{A_1, A_2\}], [P_6, 1, \{C_2\}]$

Apparently all peers except P_6 have now specialized to the maximum possible degree. So what will happen to P_6? It will eventually contact first P_2 and decide to specialize to the opposite path 11 and later encounter P_4 and obtain the missing data item pertaining to path 11. This results in the final state

$N : [P_1, 0, \{A_1, A_2\}], [P_2, 10, \{B_1, B_2\}], [P_3, 0, \{A_1, A_2\}], [P_4, 11, \{C_1, C_2\}], [P_5, 0, \{A_1, A_2\}], [P_6, 11, \{C_1, C_2\}]$

The resulting P-Grid is now not only complete, but also prefix-free. The storage load for all peers is perfectly balanced, as a result of the local decisions made to exchange and replicate data and specialize paths. Globally, however, the replication factors are not balanced. There exist three peers that support path 0, two that support path 11, and only one supports path 10. Consequently data items pertaining to path 0 are more often replicated and thus higher available. This imbalance resulted from the specific sequence of interactions performed. Other sequences would have led to other, possibly more balanced replication.

Since, however, no global coordination can be assumed we cannot exclude such "undesired" sequences of events. In the following we will introduce distributed algorithms requiring no central coordination that will both reduce global imbalance of replication factors and at the same time maintain local storage balance while P-Grids are constructed. Moreover, in case such imbalances occur as a result of the construction or changing data distributions they will re-balance the structure a-posteriori. In our example such re-balancing could be achieved by having one of the peers supporting path 0 to decide to become a peer supporting path 10. The difficulty is to decide on such changes of the P-Grid structure based on locally available information only.

5 Load balancing

5.1 Notation and Data Structures

Before we introduce the algorithms for P-Grid construction and maintenance with load balancing properties we have to give an overview of the data structures each peer maintains. We have already introduced in Section 2 the following data associated with a peer, that represents the P-Grid structure:

– The path associated with a peer: $path(p)$.

– The routing table associated with a peer: $ref(p, l)$. We will assume that a routing table contains at each level at most $refmax$ entries.

In addition we need to denote the set of data keys that is associated with a peer: $data(p)$ denotes all data stored at p. The constant m_{store} is related to the number of data keys a peer is willing to store. We assume that all peers have the same storage capacity and define overload as storing more then $2m_{store}$ data keys and underload as storing less than m_{store}. We will use the functions given in Table 1 in the following discussions.

In order to build the P2P network, peers need to interact with other peers. Like many other systems, P-Grid relies on random peer interactions. In the P-Grid implementation as well as for simulation purposes, these random interactions are facilitated by list of *fidget peers*. It is assumed that each peer p knows some (random) other peers in the network. This is the initial fidget list $fidget(p)$ for peer p. When p interacts with other peers, each peer merges their mutual fidget lists and then retains at the most $fidgetmax$ (a constant) entries from the merged list. This process ensures that over time the fidget lists are random, thereby providing peer p with knowledge about other random peers in the network, which it may contact.

While interacting peers learn about the network by encountering other peers. They keep this information in a statistics. $stat(p, k)$ corresponds to the statistics gathered by peer p from its random interactions with other peers, corresponding to level k of its path. $stat(p, k)$ has three attributes.

1. $count$: Number of times the statistics has been updated.
2. $samePath$: Statistics on encountered peers p' with $commonPath(p, p') = k$.

$length(path)$	length of string $path$
$Bit^k_{p_i}$	kth bit of $path(p_i)$
$\overline{Bit}$	$1 - Bit$
$pathPrefix(p, k)$	the first k bits of $path(p)$
$path + Bit$	string $path$ extended by bit Bit
$DataIn(u, path)$	data in temporary variable u corresponding to keys with prefix $path$
$CommonPath(p_i, p_j)$	maximal common prefix of paths for peers p_i and p_j
$RandomSelect(Set, i_{max})$	returns i_{max} entries randomly and uniformly chosen from Set; if i_{max} is larger than size of Set, the whole Set is returned
$random(a, b)$	returns a real number chosen uniformly within the interval $[a, b]$
$size()$	returns the size (number of elements) of the argument
$Reset(stat)$	initializes peers statistics $stat$

Table 1: Table of Functions

3. $compPath$: Statistics on encountered peers with $commonPath(p, p') = k-1$ and $length(path(p')) > k - 1$, i.e., the peers have a complimentary bit at level k.

5.2 Storage load balancing

$Exchange$ Algorithm 1 below describes the interactions that peers have to perform in order to construct and maintain a P-Grid. For purposes of presentation we describe all interactions of peers as a global algorithm. From this global algorithm we derived for the implementation corresponding local algorithms that are executed at the peers and communication protocols for information exchange.

As elaborated earlier P-Grid is a distributed binary search tree. In order to build such a tree without global coordination, we depend on random peer interactions, in which peers decide whether to modify locally the distributed tree data structure by changing their paths. The random interactions are initiated by using the *fidget lists* maintained by peers. We now discuss step by step the various concepts implemented by the $Exchange$ algorithm.

5.2.1 Path extension and retraction

The decisions taken by peers during their interaction are guided by the peers' desire to optimally utilize their local storage by avoiding both underload and overload situations. In order to achieve that, when peers meet, they need to do either of the following:

Become mutual replicas: If two peers meet and have the same path, and the combined data the peers store is within the limit that they are willing to store ($2m_{store}$), then they both decide to replicate all the

Algorithm 1 $Exchange(p_i, p_j, r, l_{current})$ w. l. o. g. $length(path(p_i)) \leq length(path(p_j))$

```
 1: Define $lc = length(commonPath(p_i, p_j))$ ;
 2: $UpdateStat(p_i, p_j, l_{current})$; $UpdateStat(p_j, p_i, l_{current})$;
    {Update statistics at each interacting peer according to the Algorithm 5.}
 3: $Refresh(p_i, p_j, lc)$;
    {P-Grid maintenance activities: Refreshes the routing tables and fidget lists at each peer, as described in Algorithm 4.}
 4: Define $l_i = length(path(p_i)) - lc$; Define $l_j = length(path(p_j)) - lc$;
    Define $u = data(p_i) \cup data(p_j)$;
    Define $d_{i2j} = DataIn(data(p_i), pathPrefix(p_j, lc+1))$;
    Define $d_{j2i} = DataIn(data(p_j), pathPrefix(p_i, lc+1))$;
    {$d_{i2j}$ is the data presently stored at $p_i$ which belongs to $p_j$'s current path. Similar definition for $d_{j2i}$.}
    {**Path retraction with peers having equal length paths.**}
 5: **if** $(l_i = 1)\ AND\ (l_j = 1)\ AND\ (size(u) \leq m_{store})$ **then**
 6:   $path(p_i) = pathPrefix(p_i, lc)$; $path(p_j) = pathPrefix(p_j, lc)$;
      $data(p_i) = u$; $refs(p_i, lc+1) = \{\}$; $Reset(stat(p_i, *))$;
      $data(p_j) = u$; $refs(p_j, lc+1) = \{\}$; $Reset(stat(p_j, *))$;
      {Data is replicated, references of now non-existing level are removed and statistics is reset.}
      {**Peers have incompatible paths**}
 7: **else if** $l_i > 0$ AND $l_j > 0$ **then**
 8:   $data(p_i) = data(p_i) \cup d_{j2i} - d_{i2j}$;
      $data(p_j) = data(p_j) \cup d_{i2j} - d_{j2i}$; {Retrieve data belonging to own path from the other peer.}
 9:   $RandEx(p_i, p_j, r, lc)$;
      {Initiate an exchange with a random entry from $p_j$'s routing table at level $lc+1$ as described in Algorithm 2.}
      {**Peers have the same paths, path extension possible**}
10: **else if** $(l_i = 0)$ AND $(l_j = 0)$ **then**
11:   Define $ExtensionBit = RandomSelect(\{0,1\}, 1)$;
      Define $s_0 = size(DataIn(u, (path(p_i) + \underline{ExtensionBit})))$;
      Define $s_1 = size(DataIn(u, (path(p_i) + \overline{ExtensionBit})))$; {extend paths by additional bits}
      {**Path extension by split**}
12:   **if** $(s_0 + s_1 > 2m_{store})$ AND $random(0,1) < p_{split}$ **then**
13:     $path(p_i) = path(p_i) + ExtensionBit$; $path(p_j) = path(p_j) + \overline{ExtensionBit}$;
        $data(p_i) = DataIn(u, path(p_i))$;$refs(p_i, lc+1) = \{p_j\}$; $Reset(stat(p_i, *))$;
        $data(p_j) = DataIn(u, path(p_j))$;$refs(p_j, lc+1) = \{p_i\}$; $Reset(stat(p_j, *))$;
        {Data is exchanged, new level of routing table is added and statistics is reset. }
14:   **end if**
15:   $ExchangeDecProb(p_i, p_j, s_0, s_1, l_i, l_j, r)$;
      {New $Exchange$ is initiated with decreasing probability, as described in Algorithm 3}
      {**Paths are in prefix relationship, exchange or retraction is possible**}
16: **else if** $(l_i = 0)$ AND $(l_j > 0)$ **then**
17:   Define $ExtensionBit = \overline{Bit_{p_j}^{lc+1}}$; {Complementary to the $lc+1$st bit of $p_j$'s path.}
18:   Define $s_0 = size(DataIn(u, (path(p_i) + \underline{ExtensionBit})))$;
      Define $s_1 = size(DataIn(u, (path(p_i) + \overline{ExtensionBit})))$;
      {**Path retraction of $p_j$ to that of $p_i$ if there is not enough data**}
19:   **if** $size(u) \leq m_{store}$ **then**
20:     $path(p_j) = path(p_i)$; $data(p_i) = u$; $data(p_j) = u$; {Data is replicated.}
21:     **for** $lc < d \leq length(path(p_j))$ **do**
22:       $refs(p_j, d) = \{\}$; {References of now non-existing levels are removed.}
23:     **end for**
24:     $Reset(stat(p_i, *))$; $Reset(stat(p_j, *))$;
25:   **end if**
      {**Path extension to complimentary bit at level $lc+1$ if too much data**}
26:   **if** $s_0 > m_{store}$ **then**
27:     $path(p_i) = path(p_i) + ExtensionBit$;
28:     $data(p_i) = data(p_i) \cup d_{j2i} - d_{i2j}$;$refs(p_i, lc+1) = \{p_j\}$;$Reset(stat(p_i, *))$;
29:     $data(p_j) = data(p_j) \cup d_{i2j} - d_{j2i}$;$refs(p_j, lc+1) = refs(p_j, lc+1) \cup \{p_i\}$; $Reset(stat(p_j, *))$;
        {Adding mutual entries for routing tables at level $lc+1$.}
30:
31:   **end if**
32:   $ExchangeDecProb(p_i, p_j, s_0, s_1, l_i, l_j, r)$;
33: **end if**
```

data and thus increase data availability (line 6 in Algorithm 3).

Extend paths: If two peers with the same path meet, and they together store more than $2m_{store}$ data items, then they may decide to extend their path by complimentary bits and then distributing the data according to the new paths (line 12 in Algorithm 1). Similarly, if peers meet where the path of one peer is a prefix of the other peer's path, the peer with the shorter path may decide to extend its path by a complimentary bit, if more than m_{store} data items for the extended path exist (line 27 in Algorithm 1).

Retract paths: If two peers with exactly the last bit of their path different meet, and they together store less than m_{store} (such a situation may arise, possibly from data deletion), then they may decide to retract simultaneously their path (line 5 in Algorithm 1). Similarly, if one peer's path is prefix of another peer's path, the peer with the longer path might retract by one bit, if it finds less than m_{store} data (line 20 in Algorithm 1).

5.2.2 Initiation of further exchanges

The most frequent case when two peers meet randomly, however, is that they have incompatible paths, i.e., only a common prefix, and therefore cannot perform any of the possible actions for restructuring the P-Grid (line 7 in Algorithm 1). In that case, the peers initiate immediately a further exchange between the peer with the shorter path and a peer selected from the reference table of the other peer participating in the current exchange (Algorithm 2 $RandEx(p_i, p_j, r, lc)$). This corresponds to the process of searching a peer with a compatible path, and as a result every exchange eventually results in an exchange among peers with compatible paths in which case one of the actions for restructuring can take place. This strategy is essential for a fast convergence of any P-Grid restructuring process.

In order to identify a peer for a further exchange, a peer p_i randomly selects a reference (excluding p_i itself) from p_j's routing table for level $lc + 1$. It is possible that this reference has in the meanwhile changed its path (see Algorithm Algorithm 6 in Section 6). So while searching a random reference for initiating a child $Exchange$, such stale routing references are also eliminated from p_j's routing table, thereby performing further maintenance of the routing tables (line 8 in Algorithm 2).

A second situation in which further exchanges are initiated is when peers after having changed their paths still remain in an underload situation (line 16 and line 33 in Algorithm 1). It has shown to be beneficial for a more uniform replication of data to increase the probability of exchanges occurring for these peers, which is done in Algorithm 3 $ExchangeDecProb(p_i, p_j, s_0, s_1, l_i, l_j, r)$. If the memory currently in use at peers is less than what they want to devote (m_{store}) on average, then $Exchange$ is initiated with a decreasing probability the larger the storage load is for a peer. The peer with shorter path is given preference over the one with longer path, so that on average all peers get equal opportunity to extend their paths. In the symmetric case (when $l_i = 0$ and $l_j = 0$), data is replicated by both peers if possible.

Algorithm 2 $RandEx(p_i, p_j, r, lc)$

```
1: Define $reffound = False$;
2: if $r < recmax$ then
3:   while $NOT(reffound)$ do
4:     $p_{ref} = RandomSelect(refs(p_j, lc+1) \setminus \{p_i\}, 1)$;
5:     if $length(commonPath(p_i, p_{ref})) = lc + 1$ then
6:       $reffound = True$;
7:     else
8:       $refs(p_j, lc+1) = refs(p_j, lc+1) \setminus \{p_{ref}\}$;
         {$p_{ref}$ has totally changed its path according to Algorithm 6, thus we remove invalid routing information at $p_j$}
9:     end if
10:  end while
11:  $Exchange(p_i, p_{ref}, r+1, lc+1)$;
     {Reference has been found with common prefix of $lc + 1$, so initiate another (child process) $Exchange$.}
12: end if
```

Algorithm 3 $ExchangeDecProb(p_i, p_j, s_0, s_1, l_i, l_j, r)$

```
1: if $s_0 \leq m_{store}$ AND $r < recmax$ AND $random(0, m_{store}) > s_0$ then
2:   $Exchange(p_i, RandomSelect(fidget(p_i), 1), r+1, 0)$;
3: else if $s_1 \leq m_{store}$ AND $r < recmax$ AND $random(0, m_{store}) > s_1$ then
4:   $Exchange(p_j, RandomSelect(fidget(p_j), 1), r+1, 0)$;
5: else if $(s_0 + s_1 < 2m_{store})$ AND $l_i = 0$ AND $l_j = 0$ then
6:   $data(p_i) = u$; $data(p_j) = u$;
     {In symmetric case, replicate data if possible}
7: end if
```

The recursion for $Exchange$ can be restricted from going on indefinitely by limiting the process to a maximum of $recmax$. Further, at the most one child process is initiated by $Exchange$, so there is no exponential explosion in invocations.

5.2.3 Convergence and Complexity

Ideally, if data distribution is static, once the P-Grid stabilizes, such that there are no more path extensions or retractions, fresh $Exchange$ iterations may be stopped or made to gradually die down. However, in a realistic system, there is possibly no stable data distribution due to data updates and deletions, and hence the process has to be continuous. Thus the P-Grid continuously would adapt itself to conform with the latest situation.

This continuous adaptation seems to be expensive, since at any instant of time, typically each peer is expected to interact with two peers on average (one that it contacts, and one that has contacted it). Given the dynamics of P2P systems operations for network maintenance (e.g., ping/pong messages in Gnutella networks) have to be done regularly. Thus the effort of continuously executing $Exchange$ operations is comparable to the network maintenance in any other existing P2P system.

An exact cost evaluation and comparison of P-Grid construction and maintenance is beyond the scope of this paper as it would have to include many different factors. For illustration, we just mention two points to that respect:

- Let us assume a P-Grid is forming from scratch (similarly as illustrated in the motivating example) based on a uniform data distribution and the resulting P-Grid would be based on a balanced tree of depth $O(\log(n))$. Therefore a minimum of $O(n\log(n)$ path extensions performed as part of an $Exchange$ operation would have to be performed. Since each initiation of an $Exchange$ would require $O(\log(n))$ steps on average in order to locate a peer with compatible path, $O(n\log(n)^2)$ would be an expected message cost (at least a lower bound) for stabilizing the P-Grid. Simulations [1, 3] confirm this analysis.

- As illustrated in the motivating example the construction process may result in non-uniform replication of data. From theoretical considerations we could derive that a possible counter-measure to non-uniform replication during P-Grid construction is to favor path extensions in the case where the path of one peer is a prefix of the other peer's path as opposed to those where peers meet having equal paths. Therefore we introduced p_{split}, the probability to perform a split in the case of equal paths and choose $p_{split} < 1$ in line 12 of Algorithm 1. This is also confirmed by simulations (see Section 8.1). However, such a strategy slows down the rate of changes to the P-Grid and therefore more messages will be consumed for arriving at a stable state. Therefore we have a trade-off in cost between proactively supporting uniform replication vs. reactively establish uniform replication as will be described in Section 6. Another strategy to favor uniform replication of data is to aggressively initiate additional exchanges for underloaded peers, as done by the function $ExchangeDecProb$.

5.2.4 Association of peers with identifiers

Another important aspect of the algorithm is the dynamic and unbiased association of peers with paths. The exact bit at each level that an individual peer becomes responsible for is decided randomly, so that there is no bias or correlation between peer identifier (e.g., the IP address) and peer path, unlike many other systems (e.g., Chord [24] or Pastry [23]) which associate peer identifier with data. In doing so, essentially we achieve a separation of concerns between peer identifier and stored data. This provides greater resilience against denial of service attacks, among other advantages. It also helps in replication load-balancing, since peers may migrate to any arbitrary (overloaded) path, unlike in systems, where peers do not have such flexibility. As will be elaborated in Section 10 on related work, we are the first to address the issue of replication load balancing in structured peer-to-peer systems, and it is done in a completely decentralized manner, as described in Section 6.

5.2.5 Maintenance

While executing $Exchange$, further activities pertaining to P-Grid maintenance are performed (line 2 and line 3 in Algorithm 1). When peers meet according to the $Exchange$ algorithm, they perform the $Refresh$ operation which is meant for maintenance of the P-Grid network by updating peers' fidget lists and routing tables. This includes making routing table entries random (done in Algorithm 4 $Refresh(p_i, p_j, lc)$), as required for search efficiency, since we assume random routing table entries for the proof in Section 3 and to disseminate routing entries of peers that have recently changed their paths. The fidget lists are also randomized in order to facilitate random peer interactions. During the interactions the peers also maintain the statistics needed for uniform replication by executing the $UpdateStat$ operation (Algorithm 5), which will be introduced in Section 6.

Algorithm 4 $Refresh(p_i, p_j, lc)$

```
1: $commonFidget = fidget(p_i) \cup fidget(p_j)$;
2: $fidget(p_i) = RandomSelect(commonFidget, fidgetmax)$;
   {Generate new fidget list for each peer by randomly choosing at the most $fidgetmax$ entries.}
3: $fidget(p_j) = RandomSelect(commonFidget, fidgetmax)$;
4: if $lc > 0$ then
5:    for $d = 1, lc$ do
6:       $commonRef = refs(p_i, d) \cup refs(p_j, d)$;
7:       $refs(p_i, d) = RandomSelect(commonRef, refmax)$; $refs(p_j, d) = RandomSelect(commonRef, refmax)$;
         {At the most $refmax$ routing references are randomly and independently chosen from $commonref$ by each of $p_i$ and $p_j$
         in order to ensure that the network is uniformly connected, and routing references are random, as required in the proof of
         section 3.}
8:    end for
9: end if
10: if $length(path(p_i)) > lc$ AND $length(path(p_j)) > lc$ then
11:    $refs(p_i, lc+1) = refs(p_i, lc+1) \cup \{p_j\}$; $refs(p_j, lc+1) = refs(p_j, lc+1) \cup \{p_i\}$;
       {Peers add mutual entries in their routing tables for level $lc + 1$.}
12: end if
```

5.2.6 Properties of the resulting P-Grid

There is another important characteristic of the $Exchange$ algorithm. While in a dynamic environment it is possible that the P-Grid tree is temporarily incomplete and not prefix-free, the algorithm aims at ensuring eventual completeness of the tree as well as making it eventually prefix-free. We provide some intuitive arguments why this is the case.

Prefix-free tree: If there exists peers p_i and p_j such that $Path(p_i)$ is a prefix of $Path(p_j)$ and assuming random peer interactions, these peers will eventually meet, and such a meeting will lead to path retraction for p_j or extension of the path to a complimentary bit for p_i, thus making the tree prefix-free.

Complete tree: During the path extension process, the tree is always complete. It is only during the retraction process that it is possible that one side of the tree has some peers remaining, while the other branch ceases to exist. However in such a case, eventually the remaining branch will retract as well, or find another peer that will become responsible for the complimentary bit, similar to the above case.

6 Uniform replication

As described in Section 5.2, the storage resources devoted to the P-Grid infrastructure can be locally adapted by peers in a proactive manner, primarily during the P-Grid construction process, and also during its maintenance. The number of peers responsible for a given key space should be approximately the same so that data is uniformly replicated, which amounts to global coordination problem. In the absence of global coordination or information, we try to settle for a compromise, where the imbalance in replication factors is decreased (measured in terms of the variance of replication factors for all paths), instead of achieving perfect replication load balancing.

We use a reactive randomized distributed algorithm which tries to achieve globally uniform replication based on locally available information. We provide the intuitive foundations of the algorithm in Section 6.1, followed by the heuristic algorithm in Section 6.3. We then evaluate the algorithm for a realistic situation using simulations, and describe the results in Section 8.

6.1 Think local, act global

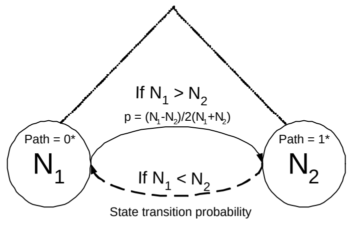
Consider the case of a P-Grid with two leaves, as shown in Figure 3(a). Let N_1 and N_2 be the actual number of replicas belonging to the paths 0 and 1. Without loss of generality, consider $N_1 > N_2$. Then, under the assumption that each peer has the ideal knowledge, $\frac{N_1-N_2}{2}$ of the peers belonging to the path 0 need to change path to 1, thus achieving replication load balancing, such that each P-Grid leaf has $\frac{N_1+N_2}{2}$ replicas. Since each of the peers has to make an autonomous decision as to whether to change path or continue to stay at the same path, we propose a randomized decision, such that peers decide to change path with probability $p_{t:0\to1} = max(\frac{N_1-N_2}{2N_1}, 0)$. The function ensures that no $0 \to 1$ transition occurs if $N_2 > N_1$. Similarly, we have $p_{t:1\to0} = max(\frac{N_2-N_1}{2N_2}, 0)$. Now, if we define $Prob_0 = \frac{N_1}{N_1+N_2}$ as the probability that peers belong to the path 0, and similarly for the path 1 define $Prob_1 = \frac{N_2}{N_1+N_2}(= 1 - Prob_0)$, then we can rewrite the migration probabilities as $p_{t:0\to1} = max(\frac{1}{2}(1-\frac{Prob_1}{Prob_0}), 0)$ and $p_{t:1\to0} = max(\frac{1}{2}(1-\frac{Prob_0}{Prob_1}), 0)$. For the case of a 2-leaved P-Grid, it is easy to see that defining the transition probabilities like this leads to equal replication factor for each path (leaf).

Of course peers cannot obtain the exact probabilities, but can estimate them by sampling randomly the network. For the situation of a two leaf P-Grid, analytical results show, that using a small sample size of 10 randomly selected samples for calculating the decision probabilities as described and repeating sampling followed by balancing for 5 cycles reduces any imbalance to less than 0.25% of the network size, independent of the network size and the initial conditions.

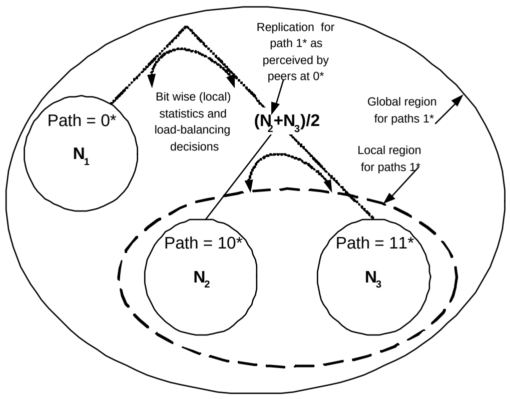
Now consider the case of a P-Grid with three leaves, as shown in Figure 3(b), with N_1, N_2 and N_3 replicas for the paths starting with 0, 10 and 11 respectively. This simplistic extension of the example captures the essential choices that have to be made by individual peers even in a realistic P-Grid, and provides an insight for the design of the replication balancing strategy encoded in Algorithm 6.

In an unbalanced tree, simply counting peers for the two halves at any level is not sufficient. This is because, even if the actual replication factor is equal for all leaves (which is desirable), the count will provide a biased information, with greater count for the half of the tree with more leaves. Such a count in itself is useless. On the other hand, knowledge of the whole subtree is not practical. Fortunately, such knowledge is not necessary either. For example, in the P-Grid with three leaves, peers belonging to path 0 will meet peers belonging to the paths 10 and 11. Essentially, they need to know that there are on an average $\frac{N_2+N_3}{2}$ peers at each leaf of the other sub-tree, but do not need to understand the shape of the sub-tree or the distribution of replication factor.

Thus, while collecting the statistical information, any peer p_i meeting another peer p_j counts the num-



(a)


Replication for path 1* as perceived by peers at 0*
Bit wise (local) statistics and load-balancing decisions
(N2+N3)/2
Global region for paths 1*
Local region for paths 1*
Path = 0*
N1
Path = 10*
N2
Path = 11*
N3

(b)

Figure 3: (a) P-Grid with two leaves, (b) P-Grid with three leaves

ber of peers encountered at each level l for either subtree (same path/complementary path), normalized through dividing it by $2^{length(path(p_j))-length(CommonPath(p_i,p_j))}$. Peers thus try to see things from a local perspective, without bothering about the larger picture, and yet, the decision is such that eventually global load balance will emerge. We expect that if two sub-halves of the tree can be mutually replication load balanced, then the sub-trees in each of the halves will recursively achieve replication load balance as well. Thus peers think about load balancing locally, but the algorithm leads to global load balancing over time.

Peers belonging to the path 10 (or 11) have to count the number of peers belonging to the path 0 and 11 (or 10), and have to decide whether to balance load, and if so, at which of the levels. For example, if $N_1 > N_2 > N_3$ then peers belonging to paths $1*$ obviously do not need to change path to $0*$. However some peers belonging to path 10 may decide to change path to 11 ($10 \rightarrow 11$), thus achieving a local load balancing. Simultaneously, migrations $0 \rightarrow 1*$ take care of the overall balancing.

If $N_1 < N_2 < N_3$ then peers belonging to 11 can prioritize migrations $11 \rightarrow 0$ over $11 \rightarrow 10$. Since peers in 10 perceive themselves to be overloaded locally (second bit), they will not participate in relatively grander scale (first bit). This is another facet of thinking locally, before participating in actions of possibly global consequence.

Note that N_i changes over time, and thus the statistics have to be refreshed and built from scratch regularly. We can thus consider the algorithm to have two phases, one that gathers statistics, and the other one that makes the probabilistic decision to change path. For example, it is possible that after such migrations, there are more peers at path 10 than at 11. But the algorithm should eventually converge achieving a global balance of replication factors.

For the sake of completeness, consider the other possible scenario where $N_2 < N_1 < N_3$. Then peers belonging to 11 can prioritize migrations $11 \rightarrow 10$ over $11 \rightarrow 0$. Other combinations essentially belong to one of the above mentioned three cases, with reversal of role for paths 10 and 11.

These ideas give the intuitive basis for the path changing strategy of Algorithm 6 and statistics collection of Algorithm 5.

Based on this idea, we defined the transition probabilities and numerically solved the changing population at each of the three leaves for various initial combinations of N_1, N_2, N_3. We give the results in Table 2. In this model in each time step t each peer determines the transition probability based upon a complete statistics of the current peer population and then performs its decision based on these probabilities. Note that in practice the replication factor will always be an integer value. The fractional values result from the numerical evaluation.

t	$N_1(t)$	$N_2(t)$	$N_3(t)$	$N_1(t)$	$N_2(t)$	$N_3(t)$	$N_1(t)$	$N_2(t)$	$N_3(t)$
0	15	20	30	20	30	15	30	20	15
1	21	24	20	21.25	21.25	22.5	23.75	20.625	20.625
2	21.5	21.5	22				22.2	21.4	21.4

Table 2: Number of replicas over time

Thus we see that the strategy is robust, irrespective of the initial replication distribution. Keeping in mind the insight obtained from the above examples of P-Grid with 2 and 3 leaves, next we propose our heuristic algorithm for balancing replication load in an arbitrary P-Grid.

6.2 Statistics maintenance: $UpdateStat(p_i, p_j, l_{current})$

For the approach to work in practice, and scale for realistic P2P systems, it is impossible to gather the exact statistical information. Thus the statistics is gathered out of small (constant) number of interactions creating a sample $minchange$. We argue that the noise due to this imperfect knowledge is compensated by the randomization inherent to our reactive algorithm. Moreover, for a practical system, perfect balancing of replication factor is neither feasible, nor required, and instead reduction in the variance is good enough for all practical purposes. Later we show by simulations that our heuristic algorithm meets the practical requirements within an agreeable latency by using statistical information.

In order to prevent higher levels of the tree (smaller l) to build up their statistics much faster than the lower ones, the $Exchange$ algorithm (Algorithm 1) passes the parameter $l_{current}$ while invoking $UpdateStat(p_i, p_j, l_{current})$. Thus, statistics is updated for levels larger than or equal to $l_{current}$, up to $lc_{i,j}$. This is because $l_{current}$ is set to $lc + 1$ during the recursive calls of $Exchange(p_i, p_j, r, l_{current})$. Thus each peer updates its statistics at each level uniformly during the course of one $Exchange$.

As mentioned above, the statistics being collected by p_i need to account for the depth of the encounter peer, so the increase in count is not one unit, but rather is normalized by $1/2^{length(path(p_j))-length(commonPath(p_i,p_j))}$.

Algorithm 5 $UpdateStat(p_i, p_j, l_{current})$

```
1:  $lc = lc_{p_i,p_j}$; $l = l_{current}$
2:  while (($l \leq lc$) AND ($length(path(p_i)) > lc$) AND ($length(path(p_j)) > lc$)) do
3:    $stat(p_i, l+1).count++$;
4:    if $lc > l$ then
5:      $stat(p_i, l+1).samePath = stat(p_i, l+1).samePath + 1/2^{length(path(p_j))-l-1}$;
6:    else
7:      $stat(p_i, l+1).compPath = stat(p_i, l+1).compPath + 1/2^{length(path(p_j))-l-1}$;
8:    end if
9:    $l++$;
10: end while
```

6.3 Balancing replication load: $ChangePath(p_i, prob_c)$

Every peer determines the replication factors for its own path-prefix and that of the complementary path for each level, by collecting at least $minchange$ samples for each level.

A change of path by a peer is only needed if it does not lead to a reduction of the number of replicas in a subspace that is already underpopulated. A subspace is defined as the keys sharing the same prefix. Therefore the peer first determines whether the subspace of the corresponding prefix is underpopulated for each level, starting from the deepest level, i.e., its own path length. If not, it may proceed up in the tree (smaller l). In that way all possible levels at which path changes may occur are determined. While doing so, the probabilities with which a change should be performed are determined from the statistics. This information is stored as a 2-tuple $(l, transition\_probability)$ in the data-structure $ProbTrans$. Ideally the transition probability for level l should be

$$Max((stat(p_i, l).samePath - stat(p_i, l).compPath), 0)/(2stat(p_i, l).samePath).$$

However, since our statistical information does not necessarily reflect the actual replication factors, it may lead to false transitions, thereby increasing the imbalance, at least for a short term. Such false decisions lead to oscillatory behavior, which not only slows down the convergence process, but also is very expensive in terms of network maintenance, for example, frequent and unnecessary repairs of P-Grid routing tables.

In order to reduce such oscillatory behavior, we add some heuristic design choices, such that the path change probability is decreased by a constant to terminate the process when the fractional difference of imbalance is of bl (chosen to be 0.1 after simulating with various values). This probability is further attenuated by a overall factor of $prob_c$. Experiments show that $prob_c = 0.25$ gives an acceptable rate of load balancing without either getting into oscillations or making the load balancing process too slow. Thus the transition probability is defined as

$$prob_c Max((stat(p_i, l).samePath - stat(p_i, l).compPath - bl, 0)/(2stat(p_i, l).samePath)).$$

While this slows down the overall process, we avoid the unnecessary oscillations, which apart from being very expensive in terms of maintenance, would stress the network even further. P-Grid is resilient to inconsistent information in the routing tables, and has been shown to gracefully degrade [14]. However, from the experiences during implementation (Section 9) it is still desirable to make conscious design choices to reduce such inconsistencies, so that maintenance overhead is reduced, which in turn leads to better system performance.

Having determined the probability for changing path, we sort $ProbTrans$ in descending order ($SortedProbTrans$), thus defining the priority of path changes at each level.

We use the notation $SortedProbTrans_j^k$ for the j^{th} field of the k^{th} entry of $SortedProbTrans$ where $j = \{1, 2\}$. Thus $SortedProbTrans_1^k$ gives the path level corresponding to the k prioritized level, and $SortedProbTrans_2^k$ gives the probability of path change at that level $SortedProbTrans_1^k$.

The peer then sequentially decides to change its path with the given transition probability, until it either changes its path, or it exhausts the whole $SortedProbTrans$. The second case corresponds to the fact that the peer chose to continue with its present path.

Since such path changes lead to complete changes of the replication factors, it renders the statistical information useless. Thus it is imperative to refresh statistics every once in a while. We do this when a peer either changes path itself, or when it has accumulated $2minchange$ samples for each level of its path.

The last piece of the jigsaw puzzle for replication load balancing is the action of changing path itself. For this, the peer needs to find another peer from the complimentary path, and then clone all its content: data, routing table, but not the statistical information, since it is in any case useless.

Algorithm 6 $ChangePath(p_i, prob_c)$

```
1: Define $l = length(path(p_i))$; Define $stop = False$; Define $bl = 0.1$
2: Define $ProbTrans =$ ; {Set of 2-tuple. First entry is the depth and second entry the probability of transition at that depth.}
3: while $l > 0$ AND NOT $stop$ do
4:    if $stat(p_i, l).samePath \geq stat(p_i, l).compPath$ then
5:       if $stat(p_i, l).count > minchange$ then
6:          $ProbTrans = ProbTrans \cup$
7:          $\{[l, prob_c Max((stat(p_i, l).samePath - stat(p_i, l).compPath - bl, 0)/(2stat(p_i, l).samePath))]\}$;
8:       end if
9:    else
10:      $stop = True$;
11:   end if
12:   $l - -$;
13: end while
14: $SortedProbTrans = ProbTrans$ sorted in descending order of 2nd field;
    {We use the notation $SortedProbTrans_j^k$ for $j^{th}$ field of the $k^{th}$ entry of $SortedProbTrans$ where $j = \{1, 2\}$, the tentative
    transition probabilities.}
15: Define $k = 1$; Define $change = False$;
16: while NOT $change$ AND $p \leq size(SortedProbTrans)$ do
17:    Define $prob = random(0, 1)$;
18:    if $prob < SortedProbTrans_2^k$ then
19:       $NewPeer = SearchPath(p_i, SortedProbTrans_1^k)$; {Function described in Algorithm 7.}
20:       $p_i$ becomes $NewPeer$'s clone. {Abandons all present information, and copies every thing from $NewPeer$.}
21:       $Reset(stat(p_i, *))$;
22:       $change = True$;
23:    end if
24:    $k + +$;
25: end while
26: if NOT $change$ AND $(stat(p_i, l).count > 2minchange) \forall l \in \{1, ..., Length(Path(p_i))\}$ then
27:    $Reset(stat(p_i, *))$; {Reset statistics of $p_i$ if more than $2minchange$ statistics available for all levels.}
28: end if
```

6.4 Find appropriate path to migrate: $SearchPath(p_i, l)$

In the first part of this section we provided intuitive reasoning to determine the probability with which peers need to change path. The $SearchPath$ algorithm begins traversing the path of the other subtree by choosing a reference ref from peer p_i's level l routing table (the level at which the decision to change path has been made), and then traversing ref's path at each lower level (larger l) or choosing to find a peer belonging to the complimentary path at the level, each with equal probability of 0.5, by recursively using $SearchPath$, until a P-Grid leaf is reached, that is, the current level equals the latest ref's path length.

This ensures that at each level both bits are chosen with equal probability. While this strategy works perfectly if the subtree is balanced, in unbalanced subtrees, it has a bias for the shorter paths, causing over-replication of shorter paths. This, however, is not a serious problem, since the subtree will in any case perform load balancing in the future. Our simulations are based on this approach.

While choosing a reference, it is possible that p_i discovers that some of the entries in its routing table are invalid (they might have changed path), and these entries are purged.

This is a lazy approach of removing invalid entries from routing tables. One may argue that routing tables will thus get depleted, however, that is not true, since entries in the routing tables are also potentially added during execution of $Refresh$ (Algorithm 4) and $Exchange$ (Algorithm 1). Simulations have shown that the inconsistent references due to path changes never pose a problem.

Apart from that, routing tables may be proactively updated by recursively querying a self-contained P-Grid directory [14] in order to account for peers that may no longer be contacted, either because they are offline or changed their physical address. This issue of routing table maintenance has been exhaustively analyzed in [14], and is beyond the scope of this work.

Algorithm 7 $SearchPath(p_i, l)$

```
1:  Define $reffound = False$;
2:  while NOT($reffound$) AND $Length(refs(p_i, l) > 0)$ do
3:      $ref = RandomSelect(refs(p_i, l), 1)$; {Select uniformly a random reference from the routing table for level $l$}
4:      if $lc_{p_i,ref} = l - 1$ then
5:          $reffound = True$; {found a proper routing reference}
6:      else
7:          $refs(p_i, l) = refs(p_i, l) \backslash \{ref\}$; {Invalid reference removed from routing table.}
8:      end if
9:  end while
10: if $reffound$ then
11:     Define $l0 = l$;
12:     while $random(0, 1) > 0.5$ AND $l0 < Length(Path(ref))$ do
13:         $l0 + +$;
14:     end while
15:     if $l0 \geq Length(Path(ref))$ then
16:         Define $NewPeer = ref$;
17:     else
18:         Define $NewPeer = SearchPath(ref, l0)$;
19:     end if
20: else
21:     Define $NewPeer = p_i$;{If further search is not possible, return itself}
22: end if
23: Return($NewPeer$);
```

7 Basic P-Grid functions

We now describe the basic operations for maintaining and using a P-Grid based on the introduced algorithms.

Node Join/Leave: When a peer p_i joins the network, it needs to know some existing member p_k. Using a small set of peers as the points of first contact may add a bias to the network topology in any structured P2P system, thereby causing inadvertent creation of a backbone. To avoid such bias p_i contacts a random peer in P-Grid. How such peer can be found using the fidget peers is described in Section 9. If p_i has data of its own, then it needs to perform the $Exchange$ algorithm, so the data initially held by p_i is distributed to responsible peers of the P-Grid. If p_i has no data of its own, it can become a replica of the random peer it had found.

If a node wants to leave the network, either temporarily (going offline) or permanently, it may do so autonomously without informing other peers. Such behavior will not jeopardize P-Grid's functionality unless it is a consorted effort of many peers, which is difficult, since the randomized construction/evolution process reduces the prospects of collusion. Replication in P-Grid takes care of availability of data, and maintenance of multiple references for routing tables at each level guarantees availability of routes with very high probability.

A node rejoining the network retains its previously established path. In order to do so, it searches its replica in P-Grid by querying for its path at any arbitrary peer. This query is then routed to a random replica, from which the peer may pull updates it had missed while it was offline. Actually it pulls from multiple replicas to form a probabilistic quorum, as has been elaborated in [13].

A fundamental difference among standard DHT approaches, such as Chord and Pastry, and real-world P2P systems, such as Gnutella and Freenet, is the ability of the latter to evolve multiple networks independently, that may join later into a common network, or eventually split again. This possibility is also provided by P-Grid, where different networks may dynamically join, as soon as its members get in touch by executing an $Exchange$ operation.

Search: The basic query routing strategy of P-Grid was already described in Section 2.

Insertion: In order to insert a new data item d into P-Grid, the hashed key k_d corresponding to the data is generated and inserted by forwarding the relevant index information to one of the replicas r_{k_d} responsible for path k_d. r_{k_d} may then initiate an update in the replica subnetwork as described below for data updates.

Update: Besides Gnutella (implicit update semantics with multiple versions), Freenet, and P-Grid no other P2P system provides data updates. P-Grid's approach [13] is the most sophisticated one among these three systems. Updates in P-Grid are performed by locating a responsible peer first as described above for data insertion and then spreading this information among its replicas. For this P-Grid uses a hybrid push/pull algorithm [13]. The push phase uses a low overhead gossiping (epidemic/rumor spreading) algorithm to convey the update to online replicas with high probability and low latency. Any replica coming online, or replica that has not received updates for a given latency conducts a pull to obtain the missed updates (actually we use a probabilistic quorum with effects similar to anti-entropy used in distributed databases). In [13] we have shown that such an approach will scale to large number of replicas (even beyond 100-1000), and can work in highly unreliable environments, for example, even if only 10% of the replicas are online. Not surprisingly, for the probabilistic guarantees to hold in such unreliable environments a moderately high replication factor (e.g., 100) is needed.

Delete: No P2P system addresses this explicitly at the moment. However, deletion can be seen as a special case of data update and similar mechanisms can be applied.

Network maintenance: Peers may change their IP addresses, for example, because they moved or got a temporary address assigned via DHCP. While this is not a problem in unstructured P2P systems, it is critical in structured P2P systems since the routing tables will be rendered useless in this case. In [14] we demonstrate that P-Grid can be used as a self-contained and self-healing directory to address this issue. The basic idea of the approach is that each peer inserts a (replicated) mapping of a unique identifier to its current location (IP address) and updates this mapping every time it changes location or becomes online again. The routing process thus exploits the unique identifiers instead of the changing IP addresses. The algorithms used for that are efficient (proven analytically and by simulations) and have self-healing capabilities. Also they guarantee some security against distributed denial of service attacks and impersonation. The directory can additionally be used as a decentralized public key infrastructure [12] similar to PGP but overcoming some of PGP's problems. Thus P-Grid's routing is successful with a very high probability even if many peers are offline or change location.

8 Simulation results

This section presents the most important experiments we performed to justify our claims and illustrates the behavior of the system if the algorithms described above are applied. The simulations aimed at verifying the abstract properties of the algorithms presented in the previous sections. Therefore they do not consider the modelling of a physical runtime environment, with different network topologies, communication latencies or heterogeneity of processing and storage capacity of nodes. All simulations are implemented in Mathematica 4.2.

8.1 Optimizing Replication Factor Balance during Storage Balancing

This simulation concerns the construction of a P-Grid from scratch and how to maintain local and global load balance simultaneously. Even though local and global load balancing can be in conflict with each other, as shown in Section 4, this trade-off can be alleviated by choosing a proper splitting strategy while extending paths as discussed in Section 5: When peers with the same path interact they have the possibility to immediately extend their paths by splitting the key space. A better approach is to reduce the splitting probability and to perform path extensions preferably between peers whose paths are of different lengths and are in a prefix relation. This naturally slows down the construction of the indexing structure, but results in a better replication distribution and thus reduces the subsequent efforts for re-balancing.

To demonstrate the effectiveness of this strategy we have conducted the following experiment: We start with 256 peers whose paths are empty and store 50 data items each, selected from a data set that is Zipf-distributed. More precisely, in the data set the probability for a data key to occur increases with the size of the key (i.e., the distribution function is correlated with the ordering of the keys). For such a distribution we can expect the most unbalanced tree structures and therefore they are most likely to be difficult cases to address. Then the peers perform exchanges till the indexing structure stabilizes. No re-balancing using Algorithm 6 is performed. The results of this experiment are shown in Table 3.

p_{split}	Exchanges	Replication Mean[a]	Replication Standard Dev.[b]	Replication Max.[c]
0.05	40,000	3.32	1.82	10
0.1	35,000	3.20	1.99	9
0.5	20,000	3.55	3.39	21
1.0	20,000	3.28	3.94	23

[a]mean of the replication factors of the different paths
[b]standard deviation of the replication factors of the different paths
[c]maximum replication factor of a path

Table 3: Influence of splitting probability on replication and path variance

The different number of steps in the table is due to the fact that if the splitting probability is higher, it takes less steps for the data structure to converge to a stable state. The increasing standard deviation of replication factors with increasing split probability clearly illustrates the advantage of our strategy. Also the maximum replication factor (an outlier) is substantially reduced.

8.2 Optimized Parameter Setting for Statistical Sampling

In this experiment we explore the parameter values for the probability of changing peer paths $prob_c$ and the sample size s needed for obtaining a valid statistical estimate when globally rebalancing replication using Algorithm 6 ($ChangePath$). $prob_c$ can also be seen as a damping factor here. Generally we can expect that higher values of $prob_c$ lead to more oscillatory behavior, as do lower values of s. On the other hand low values of $prob_c$ and high values of s substantially slow down convergence. So a good compromise needs to be found to provide optimal behavior.

The experimental setup is as follows: We generate a P-Grid randomly. The number of different paths in this P-Grid is 20 with replication factors for paths chosen randomly and uniformly from $[10, 30]$. Then statistical information for balancing is gathered by initiating peer random interactions (as in Algorithm 1 $Exchange$, though no restructuring operations are performed) and after each update of statistics both peers perform Algorithm 6 $ChangePath$. This is performed for $100*N$ rounds, where N is the number of peers, for each combination $(prob_c, s) \in \{0.125, 0.25, 0.5, 1\} \times \{5, 10, 20, 40\}$. We use the absolute and relative decrease of the variance of replication factors for the different paths as a measure for the quality of the parameters. From this initial experiment we could determine that values of $prob_c$ of 0.25 and 0.5 perform significantly better than the others, with a slight advantage for 0.5, and that a value of $s = 10$ provided the best results.

In a second phase we performed a more thorough evaluation of a reduced search space taken from the parameter combinations $\{0.25\} \times \{5, 10, 20\}$ and $\{0.125, 0.25, 0.5\} \times \{10\}$, by running 10 repeated experiments for each combination. The relative decreases of variance can be taken from Table 4:

Experiment	$prob_c$	s	Variance decrease (rel)	Standard deviation[a]
1	0.125	10	0.402992	0.173949
2	0.25	10	0.410525	0.169879
3	0.5	10	0.413688	0.112806
4	0.25	5	0.51176	0.147114
5	0.25	10	0.339744	0.124764
6	0.5	20	0.521321	0.135214

[a]Standard deviation of variance decrease over 10 experiments

Table 4: Decrease of variance of replication factors for different parameters $prob_c$ and s

Though the differences are not very substantial (which also shows that the method is fairly robust with respect to choice of parameters) we will use $prob_c = 0.25$ and $s = 10$ as obtained in experiment 5 since these parameters render a slightly better result than the others. The value of $s = 10$ is also in line with the analytical results that we have mentioned in Section 6.1.

8.3 Scalability of Replication Factor Balancing with Respect to Tree Size

In this experiment we explore the relationship between the number of different paths in a P-Grid and the rate of convergence of the global replication balancing algorithm. Since the expected depth of the tree structure grows logarithmically in the number of paths and the effort of balancing is expected to grow linearly in the tree depth, we expected to observe a logarithmic dependency between the number of different paths and the rate of convergence when performing balancing .

The experimental setup is as follows: First we randomly generate P-Grid structures with a number of paths in the interval $[p - 2, p + 2]$, for $p = \{10, 20, 40, 80\}$ (we use an interval to generate a P-Grid with the properties desired for the experiment more easily). The generated P-Grids are typically not balanced. Then we generate $r \in [10, 30]$ replicas for each path uniformly randomly and run the balancing procedure as described in Section 8.2 (without restructuring the P-Grid) $50 * N$ times, i.e. on average each peer is expected to participate 100 times in an exchange operation for updating statistics and deciding whether to change its path. The decision parameters used are the ones determined as being optimal in Section 8.2 ($prob_c = 0.25$ and $s = 10$). For each value of p the experiment is repeated 5 times. At the end of the experiment the reduction of the variance of the replica frequencies (replicas per path) is determined as compared to the initial variance. This value we consider as a measure for the rate of convergence.

Figure 4 shows the mean values and standard deviation (as error bars) of the experiment results, i.e., the mean values of the reduction of variances of replica frequencies and their associated standard deviations, for the experiments $k = 1, \ldots, 4$, with $p = 10 * 2^k$. As the x-axis is a logarithmic scale the result clearly demonstrates that the expected logarithmic dependency between the number of paths and the rate of convergence exists.

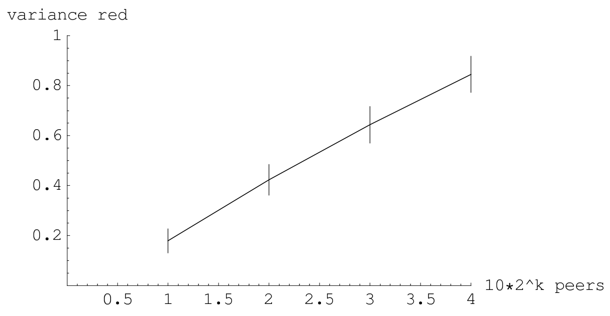


Figure 4: Variance of replication factors after experiment

8.4 Scalability of Replication Factor Balancing with Respect to Replication Factor

In this experiment we explore the relationship between the number of replicas per path and the rate of convergence of the global replication balancing algorithm. We can expect that the minimal variance obtained after the balancing has converged increases linearly in the number of replicas, and that the rate of convergence is independent of the number of replicas per path.

The experimental setup is as follows: We generate a P-Grid randomly. The number of different paths of this P-Grid is 20. Then we generate in each experiment k an increasing number of replicas by choosing uniformly randomly the number of replicas from the interval $[kn, 3kn]$. Then the balancing procedure is performed for $100 * N$ rounds as described in Section 8.2. The decision parameters are $prob_c = 0.125$ and $s = 5$. For each value of k the experiment is repeated 5 times.

Figure 5 shows the resulting variance of replication factors after the experiment. The figure shows the mean of the variance after 5 experiments for each value of $k = 1, \ldots, 7$. We included the linear regression function for the mean values. The error bars show the standard deviation of the variance values. The result supports our assumption of a linear dependency of the variance in the stable state and the number of replicas, though the error is fairly large for higher values of k.

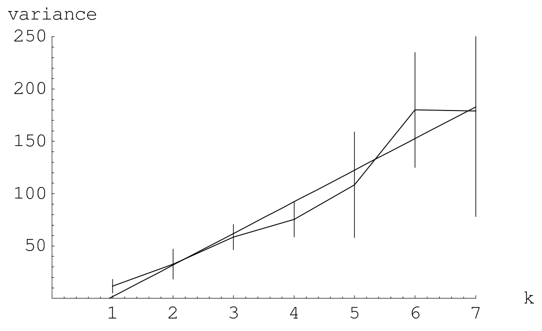


Figure 5: Variance of replication factors after experiment

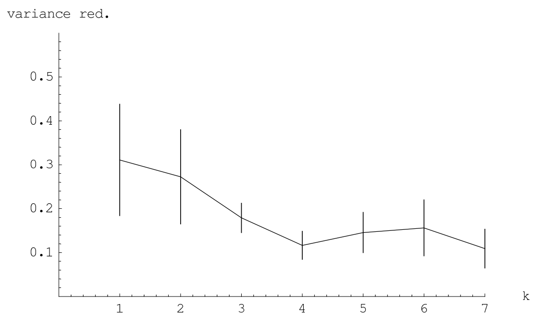


Figure 6: Ratio between initial and final variance

Figure 6 shows the ratio between the initial variance of replication factors and the variance at the end of the experiment. Actually the convergence rate appears to slightly improve for higher replication factors. This might result from the possibility of a more fine-grained adaptation with higher replication factors.

8.5 Scalability of Combined Storage Load Balancing and Replication Factor Balancing

In this experiment we explore the combined behavior of local storage load balancing by extending and retracting paths while at the same time performing global replication factor balancing by changing paths based on the statistics gathered by the peers. This is the ultimate goal of the load balancing method we propose. We have two main questions in mind when performing this experiment:

1. How adaptive is the method? Can it adapt to changing load situations in general?
2. Does the method scale with the number of peers when all balancing operations are performed simultaneously. We have seen that each of the methods for local and global load balancing scales individually, but we want to verify that scalability is maintained also if all mechanisms are integrated.

To perform the experiment we used the following setup: We generate a synthetic P-Grid which is not necessarily balanced. The replication factor is randomly selected as before. Then we generate a data set that is Zipf-distributed, such that the distribution function is correlated with the ordering of the keys as in Experiment 8.1. The data is then distributed over the different peers such that each peer holds exactly those keys that pertain to its current path. Thus the data load of the peers varies considerably, and some peers hold temporarily much more data items than their accepted maximal storage load would be. Then the algorithms for path adaptation and path changes are performed by initiating the execution of Algorithm 1 among randomly selected pairs of peers. After each exchange that two peers perform, they also test whether they should perform a change of their path by executing Algorithm 6 individually. The experiment shows that both of the questions posed above can be answered positively. The method is highly adaptive, even for such an extreme case, and it scales in the size of the peer population.

In the following we give the detailed results. We generated P-Grids with $p = 10, 20, 40, 80$ paths and chose replication factors for each path from $[10, 30]$ (thus for $p = 80$ the peer population was of size about 1600). The value m_{store} was chosen as 50 and the dataset consisted of approximately 3000 data keys.

To demonstrate the scalability we executed the same number of rounds per peer for each setting. We chose an average of 382 exchanges each peer performed, which was sufficient to reach a fairly stable state of the process. After this we took the measures shown in Table 5.

Number peers	Number paths		Replication variance[a]		Data variance[b]	
	initial	final	initial	final	initial	final
219	10	43	55.47	3.92	180,338	175
461	20	47	46.30	10.77	64,104	156
831	40	50	40.69	45.42	109,656	488
1568	80	62	35.80	48.14	3,837	364

[a]Variance of the replication factors for the different paths
[b]Variance of the number of data items stored per peer

Table 5: Results of the combined balancing method

The figures show that the re-balancing was successful. More precisely we have to look at the variances of data load and replication factors in order to measure the quality of load balancing. A number of observations can be derived from these results:

1. In all cases the data variance dropped significantly.
2. The final variance of replicas per path is increasing with the number of paths. Two factors explain this increase. First, and more importantly, the number of replicas per peer is higher (different to the initial situation) since the number of paths is reduced as a result of the adaptation of the tree structure. As we have already seen this leads to a higher expected variance. Secondly, a slight increase is due to the higher number of paths and thus the higher number of steps it takes to converge to a stable state.
3. The initial data variance varies heavily. It depends on the degree to which the randomly chosen P-Grid and the data distribution already matched. From the case $p = 40$ we might deduce that this

has also a substantial impact on the convergence speed since more restructuring has to take place. Actually, after doubling the number of interactions, in that case, the replication variance drops to 20.93, which is an expected value.

4. The number of interactions is a substantial fraction of the total size of the peer population, even in the case $p = 80$. We attribute this to a high constant factor involved in performing the load balancing since all mechanism are designed to scale logarithmically in the number of different paths. Further experiments are required to confirm this, but the current simulation environment does not support substantially larger experiments. One has to consider that a single experiment simulating 300k interactions takes approximately 1 full day of simulation time.

8.6 Search Efficiency

In the following experiment we verify whether the result from Theorem 2 also holds for the more general P-Grids occurring in practice. In particular P-Grids that are not prefix-free, where peers maintain multiple references, and replicas occur.

In this experiment we constructed P-Grids for peer populations of $p = 20, 40, 80, 160$ using the exchange algorithm $Exchange$. The data distribution was Zipf-like with probability of occurrence monotonically increasing with the data key. Thus the resulting P-Grids are heavily unbalanced. For example, in the case $p = 160$ the path lengths vary between 4 and 8 (not considering peers having prefixes of other peers' paths). Due to replication the number of different paths of the P-Grids were lower than the size of the peer population, namely $8, 26, 55, 124$.

We performed 1000 searches and computed mean and standard deviation of the number of messages used for the searches (all searches terminated successfully). We give the results in Figure 7.

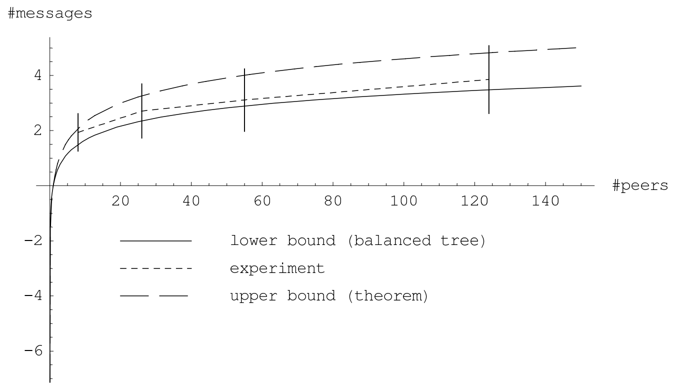


Figure 7: Experiment 4

We also included into the figure the curves for the function $\frac{\log_2(x)}{2}$, which provides a lower bound as it is the expected search cost in a fully balanced tree, and the function $\log(x)$ which is the upper bound that we determined in Theorem 2 for a subclass of all P-Grids which are highly unbalanced. The simulation shows that the effective search cost is between these two bounds as we expected.

9 Implementation

We are implementing the algorithms presented in this paper in Java resulting in our P-Grid software. A first implementation, based on earlier (somewhat simplified) versions of the algorithms [1, 6] presented in this paper has already been completed. In this section we will provide a brief overview of this implementation and some interesting experiences we made when making abstract algorithms, as introduced in this paper, suitable within a practical implementation. The system implementation aspects and the observations we made can be expected to remain valid when we will upgrade the system to the more recent versions of the algorithms we have introduced in this paper.

Up-front our main experience gained from the implementation: Large-scale systems such as P-Grid require thorough theoretical foundations (analytical evaluations and simulations) but to the same extent need to be implemented and tested in practice. In the course of the implementation many situations occurred in which the implementation of theoretically sound and efficient algorithms for constructing an overlay network required solutions to various practical issues, such as variable network delays, thread scheduling, and memory consumption.

A key issue for the implementation of P-Grid construction and maintenance was the design of protocols for exchanging administrative information and data according to the distributed algorithms described in this paper. The first version of the protocol was designed as a simple request-reply protocol: One peer sends a request to another peer (e.g. to perform the $Exchange$ algorithm) and includes the necessary information (path, routing tables, data keys) into the request. The receiver in turn returns its information. After this exchange the peers are completely aware of each others state and can independently perform the processing pertaining to their own state. This protocol had two problems: (1) The amount of data transferred was excessively high causing high network bandwidth consumption and processing cost and (2) inconsistent state changes could occur if one peer failed while processing resulting in corrupted routing tables and data.

Problem number (1) was addressed by extending the protocol with an initial invite message allowing the receiving peer to decide which data actually will be required in the processing. For example, data not pertaining to the path of the inviting peer could in many situations be ignored and needs not to be exchanged. This reduced the message traffic dramatically in particular after the P-Grid stabilized.

To provide a portable system we decided to use XML as the presentation language of the protocol's messages which would allow other implementors to provide peer software of their own that would be compatible. We experienced that the sizes of the messages in the original protocol could become quite large due to the massive overhead involved by XML. Additionally the XML parser was a major source of memory consumption when the DOM tree was built up in memory which again harmed the performance of the system. We solved these problems by implementing a tiny XML parser tailored towards our requirements and by using a zip-compressed protocol which further decreased the bandwidth consumption to a small fraction of the sized in the original implementation.

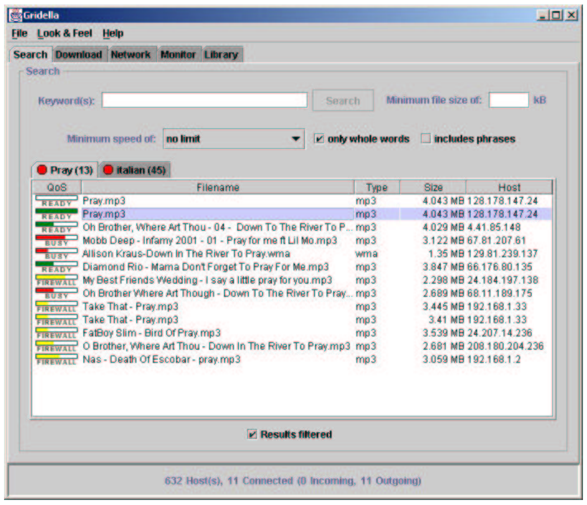
Problem number (2) was caused by failing peers as any real-world P2P system has to deal with limited availability of peers and frequent peer failures and was more difficult to tackle. If a peer fails during the exchange process this may leave the system in an inconsistent state, for example, if the peers had agreed to extend the path but one peer fails before doing so. In other words our problem was to guarantee consistent state changes of the communicating peers. The problem was solved by the above mentioned invite message and a timeout-mechanism: If the interaction does not complete in time it is considered to have failed and its effects are undone.

Most descriptions of P2P systems make the assumption that the users know some peer to serve as the point of first contact to join the P2P system. However, this assumption does not hold in practice and out-of-band mechanisms have to be provided to achieve this. For example, in Gnutella the lists of Gnutella hosts available from various web servers. Our first approach of hard-coding some peers which are guaranteed to be online clearly failed because this approach introduced a bias into the construction of the P-Grid. Thus we used the following strategy which has proven successful: when a peer initially contacts the network it selects from routing tables of peers already being online random entries and performs fixed length random walks. The end point of the random walk is chosen as the initial entry point for the P-grid construction process. We have simulated and implemented this strategy and shown that it is efficient and provides a sufficient degree of randomization. For later contacts the peer collects a large peer base to randomly choose from through this strategy and maintains a list of fidget peers. A second purpose of the fidget peer list is that we use it as a means to increase fault tolerance and provide P-Grid with additional self-healing capabilities. In extreme cases “holes” in the routing table may occur due to network separations or extreme peer failures. In such cases the fidget peers are used for random walk searches to satisfy search requests and repair the routing tables despite the bad network situation. This approach was simulated and works well in the implemented system.

Typical DHT overlay networks exclusively support exact match queries which is insufficient as soon as keys bear additional semantics. Since P-Grid does not require uniform hashing of keys to achieve load balance, it allows to support prefix respectively range queries directly. For applications using substring search on the indexed data keys, as typically done in file sharing applications, this possibility satisfies a minimal requirement. In addition, we implemented a full substring search capability by extracting and indexing all suffixes from the data keys. This was also a good test for demonstrating that our P-Grid implementation performs well under larger dataloads.

Not unexpected a major effort in the implementation went into optimizations of the software itself. After we had solved the problems described above we still were faced with severe inefficiencies due to Java as the implementation platform. Among others we experienced extreme memory consumption, memory leaks, inappropriate garbage collection behavior, and threading problems such as blocking requests though we had used sophisticated and well-tested libraries such as Doug Lea's excellent library for multi-threading. It took us a considerable effort to profile the system and track down the causes for the problems we experienced and finally we ended up with re-implementing major parts of the system to eliminate inefficiencies. In fact we had to break our textbook-clean design in some areas to get around some of the problems. The resulting implementation is stable now and consumes rather low memory, CPU, and bandwidth resources.

To provide an impression of the software we show two major functionalities of the graphical user interface below. The basic layout idea of the GUI was taken from the Limewire Gnutella client (http://www.limewire.com/ but recoded. Figure 8 shows the search interface where the user can enter queries to P-Grid and receives the resulting hits (both in the P-Grid and the Gnutella network because the software can connect to both networks in parallel to simplify migration).



Minimum speed of: no limit
[✓] only whole words
☐ includes phrases

● Pray (13) ● italian (45)

QoS	Filename	Type	Size	Host
READY	Pray.mp3	mp3	4.043 MB	128.178.147.24
READY	Pray.mp3	mp3	4.043 MB	128.178.147.24
READY	Oh Brother, Where Art Thou - 04 - Down To The River To P...	mp3	4.029 MB	4.41.85.148
BUSY	Mobb Deep - Infamy 2001 - 01 - Pray for me ft Lil Mo.mp3	mp3	3.122 MB	67.81.207.61
BUSY	Allison Kraus-Down In The River To Pray.wma	wma	1.35 MB	129.81.239.137
READY	Diamond Rio - Mama Don't Forget To Pray For Me.mp3	mp3	3.847 MB	66.176.80.135
FIREWALL	My Best Friends Wedding - I say a little pray for you.mp3	mp3	2.298 MB	24.184.197.138
BUSY	Oh Brother Where Art Though - Down To The River To Pray...	mp3	2.689 MB	68.11.189.175
FIREWALL	Take That - Pray.mp3	mp3	3.445 MB	192.168.1.33
FIREWALL	Take That - Pray.mp3	mp3	3.41 MB	192.168.1.33
FIREWALL	FatBoy Slim - Bird Of Pray.mp3	mp3	3.539 MB	24.207.14.236
FIREWALL	O Brother, Where Art Thou - Down In The River To Pray.mp3	mp3	2.681 MB	208.180.204.236
FIREWALL	Nas - Death Of Escobar - pray.mp3	mp3	3.059 MB	192.168.1.2

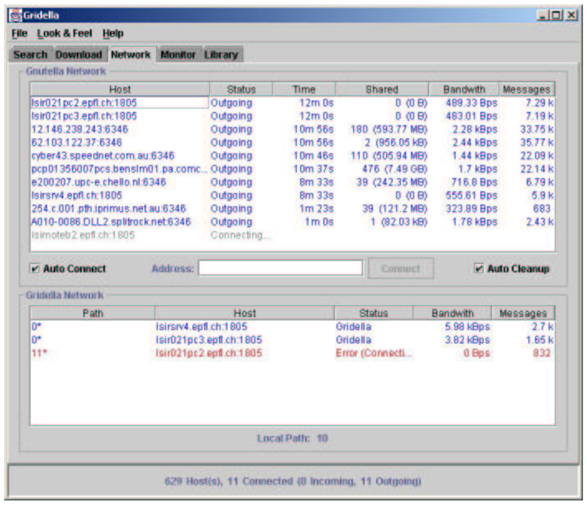
[✓] Results filtered

632 Host(s), 11 Connected (0 Incoming, 11 Outgoing)

Figure 8: Search GUI

To optimize the search results the user can provide qualitative requirements such as minimum speed of the answering peer or whether the query should be a sub-string search or only apply to full words. The search results in turn provide quality-of-service (QoS) feedback (e.g., load of the responding peer) and can be filtered to allow the user to select a fast peer for download. Downloads can be initiated via a one-click operation from this window. The status information at the bottom of the window provides the user with a rough state of the currently connected network.

A more detailed network state is provide in the network tab of the GUI which is shown in Figure 9. Here the user sees the local path of the peer and its routing table, i.e., the parts of the overall tree it knows about. For each of the peers in the routing table some status information is provided. Additionally the screenshot shows the state of Gnutella (top) network the peer is currently participating in. In fact the software can include arbitrary P2P systems besides P-Grid under a single GUI and Gnutella was included as a proof-of-concept.



Gnutella Network

Host	Status	Time	Shared	Bandwith	Messages
lsir021pc2.epfl.ch:1805	Outgoing	12m 0s	0 (0 B)	489.33 Bps	7.29 k
lsir021pc3.epfl.ch:1805	Outgoing	12m 0s	0 (0 B)	483.01 Bps	7.19 k
12.146.238.243:6346	Outgoing	10m 56s	180 (593.77 MB)	2.28 kBps	33.75 k
62.103.122.37:6346	Outgoing	10m 56s	2 (956.05 kB)	2.44 kBps	35.77 k
cyber43.speednet.com.au:6346	Outgoing	10m 46s	110 (505.94 MB)	1.44 kBps	22.09 k
pcp01356007pcs.benslm01.pa.comc...	Outgoing	10m 37s	476 (7.49 GB)	1.7 kBps	22.14 k
e200207.upc-e.chello.nl:6346	Outgoing	8m 33s	39 (242.35 MB)	716.8 Bps	6.79 k
lsirsrv4.epfl.ch:1805	Outgoing	8m 33s	0 (0 B)	555.61 Bps	5.9 k
254.c.001.pth.iprimus.net.au:6346	Outgoing	1m 23s	39 (121.2 MB)	323.89 Bps	683
A010-0086.DLL2.splitrock.net:6346	Outgoing	1m 0s	1 (82.03 kB)	1.78 kBps	2.43 k
lsirnoteb2.epfl.ch:1805	Connecting...				

[✓] Auto Connect Address: [✓] Auto Cleanup

Gridella Network

Path	Host	Status	Bandwith	Messages
0*	lsirsrv4.epfl.ch:1805	Gridella	5.98 kBps	2.7 k
0*	lsir021pc3.epfl.ch:1805	Gridella	3.82 kBps	1.65 k
11*	lsir021pc2.epfl.ch:1805	Error (Connecti...	0 Bps	832

Local Path: 10

629 Host(s), 11 Connected (0 Incoming, 11 Outgoing)

Figure 9: Network status GUI

10 Related Work

A detailed overview and classification of current approaches for P2P systems (structured, unstructured, hierarchical) is given in [4]. In the following we focus on the more specific issues of load balancing and replication in P2P systems.

For data replication in P2P systems we can distinguish five different methods that are employed depending on the mechanism to initiate the replication (partially according to the classification from [19]):

Owner replication: A data object is replicated to the peer that has successfully located it through a query. This form of replication occurs naturally in P2P file sharing systems such as Gnutella (unstructured), Napster (hierarchical), and Kazaa (super-peers) since peers implicitly make available to other users the data that they have found and downloaded (though this feature can be turned off by the user).

Path replication: A data object is replicated along the search path that is traversed as part of a search. This form of replication is used in Freenet which routes results back to the requester along the search path in order to achieve a data clustering effect for accelerating future searches. This strategy would also be applicable to unstructured P2P networks in order to replicate data more aggressively.

Random replication: A data object is replicated as part of a randomized process. For unstructured networks it has been shown that random replication, initiated by searches and implemented by selecting random nodes visited during the search process, is superior to owner and path replication [19].

Controlled replication: Here data objects are actively replicated a pre-defined number of times when they are inserted into the network. This approach is used in strongly coupled P2P networks such as Chord [24], CAN [22], and Pastry [23]. We can distinguish two principal approaches: Either a fixed number of structured networks is constructed in parallel or multiple peers are associated with the same or overlapping parts of the data key space. This approach does not adapt replication to the changing environment with variable resource availability.

Adaptive replication: Here the replication process aims at uniformly exploiting the storage resources available at peers while also trying to achieve uniform distribution of the replicas of a data object, i.e., for each data object approximately the same number of replicas exist. This is the approach used in P-Grid.

Replication of index information is applied in structured and hierarchical P2P networks. For the super-peer approach it has been shown that having multiple replicated super-peers maintaining the same index information increases system performance [25]. Structured P2P networks typically maintain multiple entries for the same routing path to have alternative routing paths at hand in case a referenced node fails.

With respect to load balancing in structured P2P systems (DHT based systems) only a few recent works have been reported.

Systems that apply uniform hashing, i.e. do not exploit the structure of the key space in the search, have to deal with only moderate load balancing problems, as the imbalance will be of order $O(\log(n))$ [21]. For non-uniformly distributed data keys we proposed in [6] an order preserving hashing function based on data sampling to improve load balance, but this approach is only applicable with fairly stable load distributions.

A load balancing strategy for DHTs based on Chord is proposed in [8]. In order to provide load-balancing, multiple (a constant number of) hash functions are used instead of only one, and multiple peers, each close to one such key generated is chosen. Then, among these multiple possible peers, the one with least load stores the data item, while the others store a pointer to this peer. Since each data item creates different sets of hashed keys, essentially, redirections need to be maintained for each data item. This means that the scheme does not scale in the number of data items. As a further adaptation, if a particular peer is overloaded, it transfers the data to one of the redirecting peers, and adds a redirecting pointer itself, thus increasing the number of redirections. The introduction of these mechanisms (redirections) imply that the Chord's original search algorithm no longer works in itself. Essentially, the approach ends up maintaining multiple Chord overlays in the same physical peers, and these Chord networks are interconnected in an unpredictable manner, running risk of loosing its efficiency.

In [27], an extension of CAN, namely e/CAN is proposed which uses shortcuts (so-called expressways) for forwarding queries to non-neighboring zones. The resulting access structure is essentially a binary search tree which resembles the P-Grid's underlying access structure [1]. Compared to CAN, in e/CAN queries are faster and the individual peer load for forwarding queries has been shown to be more balanced, as it is the typical behavior we have shown for P-Grid.

In [21] a load balancing scheme for Chord is introduced that is based on the notion of virtual servers. Each physical node can support multiple virtual servers. For overloaded nodes several strategies for moving virtual servers to underloaded nodes are discussed. Nodes are responsible to split the data space to keep the load of each virtual server bounded. The splitting strategy is similar to the splitting used in our storage load balancing strategy, however, this work does not consider the effects on replication nor on search efficiency. The authors mention that there might be negative effects on search efficiency with their approach.

Substantial work on distributed data access structures has also performed in the area of distributed databases on scalable data access structures, such as [16, 17]. This work is apparently relevant, but the existing approaches apply to a different physical and application environment. Databases are distributed over a moderate number of fairly stable database servers and workstation clusters. Thus reliability is assumed to be high and replication is used only very selectively [18] for dealing with exceptional errors. Central servers for realizing certain coordination functions in the network are considered as acceptable and execution guarantees are mostly deterministic rather than probabilistic. Distributed search trees [15] are constructed by a full partitioning, not using the principle of scalable replication of routing information at the higher tree levels, as originally published in [20] (with exceptions [26]). Nevertheless, we believe that at the current stage the potential of applying principles developed in this area to P2P systems is not yet fully exploited.

11 Conclusions

A key success factor for P2P systems is their application of the principle of resource sharing, a key characteristics is their lack of central control. In this paper we proposed a complete solution approach for enabling fair and efficient resource sharing for structured P2P networks. The approach includes the distributed data structures and algorithms for supporting efficient search, self-organizing load balancing mechanisms, their evaluation using simulation and a practical implementation. We both enable fair distribution of workload by local load balancing as well as optimized usage of available resources by global load balancing. The infrastructure introduced in this paper is intended as a basis for higher-level services in P2P systems, in particular for enabling trusted interactions. Examples of such decentralized services we are currently investigating are identity management, reputation management and document ranking.

References

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

[2] Karl Aberer. Efficient Search in Unbalanced, Randomized Peer-To-Peer Search Trees. Technical Report IC/2002/79, Ecole Polytechnique Fédérale de Lausanne (EPFL), 2002.

[3] Karl Aberer. Scalable Data Access in P2P Systems Using Unbalanced Search Trees. In *Proceedings of the 4th Workshop on Distributed Data and Structures (WDAS'2002)*, 2002.

[4] Karl Aberer and Manfred Hauswirth. *Practical Handbook of Internet Computing*, chapter Peer-to-Peer Systems. CRC Press, 2003. To be published.

[5] Karl Aberer, Manfred Hauswirth, and Magdalena Punceva. Self-organized construction of distributed access structures: A comparative evaluation of P-Grid and Freenet. In *Proceedings of the 5th Workshop on Distributed Data and Structures (WDAS'2003)*, 2003.

[6] Karl Aberer, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. Improving Data Access in P2P Systems. *IEEE Internet Computing*, 6(1), Jan./Feb. 2002.

[7] E. Adar and B.A. Huberman. Free Riding on Gnutella. *First Monday*, 5(10), 2000. http://firstmonday.org/issues/issue5_10/adar/index.html.

[8] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[9] Ian Clarke, Theodore W. Hong, Scott G. Miller, Oskar Sandberg, and Brandon Wile. Protecting Free Expression Online with Freenet. *IEEE Internet Computing*, 6(1):40–49, Jan./Feb. 2002.

[10] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymit y and Unobservability*, number 2009 in LNCS, 2001.

[11] A. Datta, M. Hauswirth, and K. Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *Proceedings of ICDCS 2003*, 2003.

[12] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Beyond "web of trust": Enabling P2P E-commerce. In *IEEE Conference on Electronic Commerce (CEC'03)*, 2003.

[13] Anwitaman Datta, Manfred Hauswirth, and Karl Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003)*, 2003.

[14] Manfred Hauswirth, Anwitaman Datta, and Karl Aberer. Handling Identity in Peer-to-Peer Systems. In *6th International Workshop on Mobility in Databases and Distributed Systems, in conjunction with the 14th International Conference on Database and Expert Systems Applications (DEXA'2003)*, 2003.

[15] B. Kröll and P. Widmayer. Distributing a Search Tree Among a Growing Number of Processors. In *ACM SIGMOD Conference*, pages 265–276, 1994.

[16] W. Litwin, M. Neimat, and D. A. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *VLDB*, pages 342–353, 1994.

[17] W. Litwin, M. Neimat, and D. A. Schneider. LH* – A Scalable, Distributed Data Structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.

[18] W. Litwin and T. Schwarz. LH*RS: A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes. In *SIGMOD Conference*, pages 237–248, 2000.

[19] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *International Conference on Supercomputing*, 2002.

[20] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing Nearby Copies of Replicated Objects in a Distribute d Environment. In *9th Annual Symposium on Parallel Algorithms and Architectures*, 1997.

[21] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load Balancing in Structured P2P Systems. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *ACM SIGCOMM*, 2001.

[23] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

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

[25] Beverly Yang and Hector Garcia-Molina. Improving Search in Peer-to-Peer Networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, 2002.

[26] H. Yokota, Y. Kanemasa, and J. Miyazaki. Fat-Btree: An Update-Conscious Parallel Directory Structure. In *International Conference on Data Engineering*, pages 448–457, 1999.

[27] Zheng Zhang, Shu-Ming Shi, and Jing Zhu. Self-balanced P2P Expressway: When Marxism meets Confucian. Technical Report MSR-TR-2002-72, Microsoft Research, July 2003.