D^2-Tree: A New Overlay with Deterministic Bounds

Gerth Stølting Brodal[1], Spyros Sioutas[2], Kostas Tsichlas[3], and Christos Zaroliagis[4]

[1] MADALGO (Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation), Aarhus University
gerth@madalgo.au.dk
[2] Ionian University, Department of Informatics
sioutas@ionio.gr
[3] Aristotle University of Thessaloniki, Department of Informatics
tsichlas@csd.auth.gr
[4] CTI and Dept. of Computer Engineering & Informatics, University of Patras
zaro@ceid.upatras.gr

Abstract. We present a new overlay, called the *Deterministic Decentralized tree* (D^2-tree). The D^2-tree compares favourably to other overlays for the following reasons: (a) it provides matching and better complexities, which are deterministic for the supported operations; (b) the management of nodes (peers) and elements are completely decoupled from each other; and (c) an efficient deterministic load-balancing mechanism is presented for the uniform distribution of elements into nodes, while at the same time probabilistic optimal bounds are provided for the congestion of operations at the nodes.

1 Introduction

Decentralized systems and in particular Peer-to-Peer (P2P) networks have become very popular of late and are widely used for sharing resources and store very large data sets. Data are stored at the nodes (or peers) and the most crucial operations are data search (identify the node that stores the requested information) and updates (insertions/deletions of data). Searching and updating is typically done by building a logical *overlay network* that facilitates the assignment and indexing of data at the nodes. Sometimes, we distinguish between the overlay structure per se and the indexing scheme used to access the data.

Following the typical modeling, a decentralized communication network is represented by a graph. Its nodes correspond to the network nodes, while its edges correspond to communication links. We assume constant size messages between nodes through links and asynchronous communication. It is assumed that the network provides an upper bound on the time needed for a node to send a message and receive an acknowledgment. The complexity of an operation is measured in terms of the number of messages issued during its execution. Throughout the paper, when we refer to cost we shall mean number of messages (internal computations at nodes are considered insignificant). The *overlay* is another graph defined over the communication network. The nodes of the overlay correspond to nodes of the original network, while its edges (links) may not correspond to existing communication links, but to communication paths.

With respect to its *structure*, the overlay supports the operations *Join* (of a new node v; v communicates with an existing node u in order to be inserted into the overlay), and *Departure*(of an existing node u; u leaves the overlay announcing its intent to other nodes of the overlay). The overlay is used to implement an *indexing scheme* for the stored data. Such a scheme supports the operations *search* for an element, *insert* a new element, *delete* an existing element, and *range query* for elements in a specific range.

In terms of efficiency, an overlay network should address the following issues:

– *Fast queries and updates:* updates and queries must be executed in a minimal number of communication rounds and using a minimal number of messages.
– *Ordered data:* keeping the data in order facilitates the implementation of various enumeration queries when compared to a simple dictionary that can only answer membership queries, including those arising in DNA databases, location-based services, and prefix searches for file names or data titles. Indeed, the ever-wider use of P2P infrastructures has found applications that require support for range queries (e.g., [6]).
– *Size of nodes (peers):* the size of a node is the routing information (links and related data) maintained by this node and it is not related to the number of data elements stored in it. Keeping the size of a node small allows for more efficient update operations, but in general reduces the efficiency of access operations while aggravating fault tolerance.
– *Fault Tolerance:* the structure should be able to discover and heal failures at nodes or links.
– *Congestion:* it refers to the distribution of the load of search (access) operations per node, aiming at distributing this load equally across all nodes. The congestion is an *expected* quantity defined as the maximum, among all nodes, of the fraction of the expected number of accesses of a node due to a random sequence of operations on the structure.
– *Load Balancing:* it refers to the distribution of data elements on the nodes. The goal of load balancing is to distribute equally the n elements stored in the N nodes of the network (typically $N \ll n$). That is, ideally each node should carry approximately k elements, where $\lfloor n/N \rfloor \leq k \leq \lfloor n/N \rfloor + 1$.

There has been considerable recent work in devising effective distributed search and update techniques. Existing structured P2P systems can be classified into two broad categories: distributed hash table (DHT)-based systems and tree-based systems. Examples of the former, which constitute the majority, include Chord [11], Pastry [14], Symphony [12], and Tapestry [17]. DHT-based systems support exact match queries well and use (successfully) probabilistic methods to distribute the workload among nodes equally. DHT-based systems work with little synchrony and high *churn* (the collective effect created by independent burstly arrivals and departures of nodes), a fundamental characteristic of the

Internet. Since hashing destroys the ordering on keys, DHT-based systems typically do not possess the functionality to support straightforwardly range queries, or more complex queries based on data ordering (e.g., nearest-neighbor and string prefix queries). The most recent effort towards range queries is reported in [16].

Tree-based systems are based on hierarchical structures. They support range queries more naturally and efficiently as well as a wider range of operations, since they maintain the ordering of data. On the other hand, they lack the simplicity of DHT-based systems, and they do not always guarantee data locality and load balancing in the whole system. Important examples of such systems include Skip Graphs (SG) [4,7], NoN SG [13], SkipNet (SN), Deterministic SN [9], Bucket SG [3], Family Trees [15], Skip Webs [1], BATON [10], Rainbow Skip Graphs (RSG) [8], and Strong RSG [8].

In this work, we focus on tree-based overlay networks that support directly range and more complex queries. Let N be the number of nodes present in the network and let n denote the size of data ($N \ll n$). Let M be the size of each node, $Q(n, N)$ be the cost of a single query, $U(n, N)$ be the cost of an update, $C(n, N)$ be the congestion per node (measuring the load) incurred by search operations, and let $L(n, N)$ be the cost for load balancing the overlay w.r.t. element updates. With respect to congestion, each node issues one operation, while the destination node of the operation is assumed to be selected uniformly at random among all nodes of the network. Congestion depends on the distribution of elements into nodes as well as on the topology of the overlay. It provides hints as to how well the structure avoids the existence of *hotspots* (i.e., nodes which are accessed multiple times during a sequence of operations – the root of a tree is usually a hotspot in decentralized tree structures).

A comparison of the aforementioned tree-based overlays is given in Table 1. We would like to emphasize that w.r.t. load balancing, there are solutions in the literature either as part of the overlay (e.g., [10]) or as a separate technique (e.g. [3,7]). These solutions are either heuristics, or provide expected bounds under

Table 1. A comparison between previous methods and the D^2-tree. By $\widehat{O}$ we represent expected bounds, by $\widetilde{O}$ we represent amortized bounds, and by $\overline{O}$ expected amortized bounds. All other bounds are worst-case. Typically, $N \ll n$.

Methods	N	M	$Q(n,N)$	$U(n,N)$	$C(n,N)$	$L(n,N)$
SG [4,7]	$\leq n$	$O(\log N)$	$\widehat{O}(\log N)$ w.h.p.	$\widehat{O}(\log N)$ w.h.p.	$\widehat{O}(\frac{\log N}{N})$	$\widetilde{O}(\log N)$
NoN SG [13]	n	$O(\log^2 n)$	$\widehat{O}(\frac{\log n}{\log\log n})$	$\widehat{O}(\log^2 n)$	$\widehat{O}(\frac{\log^2 n}{n})$	–
Determ. SN [9]	n	$O(\log n)$	$O(\log n)$	$O(\log^2 n)$	$O(\frac{n^{0,32}}{n})$	–
BATON [10]	$\leq n$	$O(\log N)$	$O(\log N)$	$O(\log N)$	–	$\overline{O}(\log n)$
Family Trees [15]	n	$O(1)$	$\widehat{O}(\log n)$	$\widehat{O}(\log n)$	$\widehat{O}(\frac{\log n}{n})$	–
Bucket SG [3]	$\leq n$	$O(\frac{n}{N} + \log N)$	$\widehat{O}(\log N)$	$\widehat{O}(\log N)$	$\widehat{O}(\frac{1}{N} + \frac{\log N}{n})$	No Bounds
Skip Webs [1]	n	$O(\log n)$	$\widehat{O}(\frac{\log n}{\log\log n})$	$\widehat{O}(\frac{\log n}{\log\log n})$	$\widehat{O}(\frac{\log n}{n})$	–
Rainbow SG [8]	n	$O(1)$	$\widehat{O}(\log n)$ w.h.p.	$\overline{O}(\log n)$ w.h.p.	$\widehat{O}(\frac{\log n}{n})$	–
Strong RSG [8]	n	$O(1)$	$O(\log n)$	$\widetilde{O}(\log n)$	$\widehat{O}(\frac{n^{\epsilon}}{n})$	–
D^2-tree	$\leq n$	$O(1)$	$O(\log N)$	$\widetilde{O}(\log N)$	$\widehat{O}(\frac{\log N}{N})$	$\widetilde{O}(\log N)$

certain assumptions, or amortized bounds but at the expense of increasing the size per node (see [5] for a detailed discussion).

Our Contribution. In this paper we present a new tree-based overlay, called the *Deterministic Decentralized tree* or *D^2-tree.* The D^2-tree (see also Table 1) uses $O(1)$ space per node, achieves a deterministic $O(\log N)$ query bound and a deterministic (amortized) $O(\log N)$ update bound for elements as well as for node joins and departures, achieves *optimal* congestion, and exhibits a deterministic (amortized) $O(\log N)$ bound for load-balancing. Moreover, it supports ordered data queries optimally, and tolerates node failures.

The D^2-tree is an overlay consisting of two levels. The upper level is a perfect binary tree, while the lower level consists of buckets (sets of nodes), where each bucket is structured as a doubly linked list. Each bucket contains $O(\log N)$ nodes. Since N changes, the size of buckets is dynamically maintained by the overlay.

In the D^2-tree, we separate the index from the overlay structure using the load-balancing mechanism. The number of elements per node is dynamic w.r.t. node joins and departures and it is controlled by the load-balancing mechanism. Moreover, the number of nodes of the perfect binary tree is not connected by any means to the number of elements stored in the structure. The overlay structure supports the operations of node join and node departure, while at the same time it tackles failures of nodes whenever these are discovered.

Our load-balancing technique distributes almost equally the elements among nodes by making use of weights. Weights are used to define a metric of load-balance, which shows how uneven is the load between nodes. When the load is uneven, then a data migration process is initiated to equally distribute elements.

Our load-balancing technique is quite general and can be applied to any hierarchical decentralized overlay (e.g., BATON, Skip Graphs) with the following specifications: (i) The overlay structure must be a tree with height $O(\log N)$ with each node having $O(1)$ children. (ii) Nodes at level i having the same father have approximately (within constant factors) the same weight, which is $\Omega(i^4)$. (iii) Updates are performed at the leaves. Alternatively, if each node has access to a leaf in $O(1)$ messages then this is enough, since the update is simply forwarded to this leaf.

We discuss the load balancing technique in Section 2, and present the D^2-tree in Section 3. We conclude in Section 4. Due to space constraints, some details and proofs are deferred to the full version [5].

2 Deterministic Load Balancing

The load-balancing mechanism distributes almost equally the elements among nodes by making use of weights, which are used to define a metric showing how uneven is the load between nodes. When the load is uneven, then a data migration process is initiated to equally distribute elements.

A few definitions are in place. Assume that the overlay structure is a tree $\mathcal{T}$. Based on $\mathcal{T}$ ancestor-descendant relationships are defined. There is a node that has no ancestor (the *root*) and there are nodes with no descendants (the *leaves*).

All nodes which are not leaves are called *internal.* The subgraph induced by the descendants of node v (including v) in $\mathcal{T}$ is the *subtree* of v. The *weight* $w(v)$ of node v is equal to the number of elements stored in its subtree. The term weight will also be used to express other similar quantities at some parts of the paper, in which case we explicitly say so. The number of elements residing in node v is represented by $e(v)$. The *height* of node v is the length of the longest path from v to one of its leaves. The *depth* (or *level*) of node v is the length of the path from v to the root. Two nodes are called *brothers* when they have the same father and they are consecutive in his child list.

We describe the load-balancing mechanism in two steps. First, we provide a mechanism that allows for efficient and local update of weight information in a tree when elements are added or removed at the leaves. This is necessary to avoid hotspots. Then, we describe the load-balancing scheme in a tree overlay.

2.1 A Technique for Amortized Constant Weight Updating

We provide a technique that lazily updates the weights on the nodes of a tree. When an element is added or removed to/from a leaf u in $\mathcal{T}$ the weights on the path from u to the root must be updated. If the height of $\mathcal{T}$ is H, then the cost of the weight updating is $O(H)$. Assume that node v lies at height h and its children are $v_1, v_2, \ldots, v_s$ at height $h-1$. We relax the weight of a node and its recomputation. We define the *virtual weight* $b(v)$ of v as the weight stored in node v. In particular, for node v the following invariants are maintained

Invariant 1. $b(v) > e(v) + (1-\epsilon_h)\left(\sum_{i=1}^{s} b(v_i)\right)$

Invariant 2. $b(v) < e(v) + (1+\epsilon'_h)\left(\sum_{i=1}^{s} b(v_i)\right)$

where ϵ_h and ϵ'_h are appropriate constants. These invariants imply that the weight information is approximate, at most by a multiplicative constant.

Assume that an update takes place at leaf u. Apparently, only the weight of its ancestors need to be updated by ± 1 and no other node is affected. We traverse the path from u to the root until we find a node z for which Invariants 1 and 2 hold. Let v be its child for which either Invariant 1 or 2 does not hold on this path. We recompute all weights on the path from u to v. In particular, for each node z on this path, we update its weight information by taking the sum of the weights written in its children plus the number of elements that z carries.

The following lemma states how frequently the weight information in each node changes. Its proof follows from the fact that the update of node v is a result of the violation of either of Invariants 1 or 2 and by taking into account that $\frac{1}{2}\cdot w(v) < b(v) < 2\cdot w(v)$, if we choose $\epsilon_h = \epsilon'_h = \frac{1}{h^2}$ [5].

Lemma 1. *The minimum number of updates in the subtree of v, causing a weight update at v, is $\Theta(\epsilon_h w(v))$.*

The above lemma states that if we make $\epsilon_h w(v)$ update operations then the maximum number of weight changes at node v is 1, implying that the amortized cost per update operation at height h is $\frac{1}{\epsilon_h b(v)}$. Since a node on the path at height i has (by assumption) virtual weight $\Omega(i^4)$, it is not hard to see that the weight updating mechanism is efficient in an amortized sense.

Theorem 1. *The amortized cost of the weight update algorithm is $O(1)$.*

2.2 Updates and Load Balancing

We now investigate how load balancing is realized on the balanced tree structure $\mathcal{T}$. For clarity of exposition, we assume that $\mathcal{T}$ is a binary tree. The following discussion can be easily generalized for trees with $O(1)$ maximum degree, simply by looking between brother nodes.

First, bear in mind that this mechanism does not tamper with the structure of $\mathcal{T}$. An update operation (either insertion or deletion of an element) is initiated at node v. Node v issues a search for the involved element and the appropriate node u is returned. Then, the update request is forwarded from v to u. Node u executes the update operation and signals v for the status of the update. The load balancing mechanism redistributes the elements among nodes when the load between nodes is not distributed equally enough.

Assume that node v at height h has child p and its right brother q at height $h-1$. Let $|v|$ denote the number of nodes of the subtree of v (including v) in the overlay structure. The *density* $d(v) = \frac{w(v)}{|v|}$ of node v represents the mean number of elements per node in the subtree of v. The *criticality* $c(p,q) = \frac{d(p)}{d(q)}$ of two brother nodes p and q represents their difference in densities. The following invariant guarantees that there will not be large differences between densities.

Invariant 3. *For two brothers p and q, it holds that $\frac{1}{c} \leq c(p,q) \leq c$, $1 < c \leq 2$.*

For example, choosing $c = 2$ we get that the density of any node can be at most twice or half of that of its brother. In the more general case where the number of children of node v is $O(1)$, we get that no child of v has more density than a constant factor w.r.t. the other children of v.

When an update takes place at leaf u, weights are updated by using the mechanism described in Section 2.1. In this way, we guarantee that no hotspot exists w.r.t. weight updating as implied by Lemma 1. Then, starting from u, the highest ancestor w is located that is unbalanced w.r.t. his brother z, meaning that Invariant 3 is violated. Finally, the elements in the subtree of their father v are redistributed uniformly so that the density of the brothers becomes equal; this procedure is henceforth called *redistribution* of node v. Assume that the redistribution phase has a cost of $O(f(w(v)))$, for some increasing function $f : \mathbb{N} \rightarrow \mathbb{N}$. The following theorem provides amortized bounds for the redistribution.

Theorem 2. *The load balancing has an amortized cost of $O\left(H\frac{f(n)}{n}\right)$.*

3 The D^2-Tree

In this section we design and analyze the D^2*-tree* overlay. We first describe the overlay structure, then move to the description of the index, and finally discuss efficiency issues regarding congestion and fault-tolerance.

3.1 The D^2-Tree Structure

The D^2-tree is a binary tree, where each node maintains an additional set of links to other nodes apart from the standard links which form the tree. Each node v in the tree maintains the following links:

1. Links to its father (if there is one) and its children.
2. Links to its adjacent nodes based on an inorder traversal of the tree.
3. Links to nodes at the same level as v. These links facilitate an exponential search on the nodes of the same level. Assume that node v lies at level ℓ. In a binary tree, the maximum number of nodes at level ℓ is equal to 2^ℓ. Node v maintains at most 2ℓ links: ℓ links to nodes to the right and ℓ links to nodes to the left. The links are distributed in exponential steps, that is the first link points to a node (if there is one) 2^0 positions to the left (right), the second 2^1 positions to the left (right) and the i-th link 2^{i-1} positions to the left (right). These links constitute the *routing table* of v.

The next lemma captures some important properties of the routing tables w.r.t. their construction. It follows immediately from the aforementioned link structure and the fixed distances between successive links in the routing tables.

Lemma 2. *(i) If a node v contains a link to node u in its routing table, then the parent of v also contains a link to the parent of u, unless u and v have the same father. (ii) If a node v contains a link to node u in its routing table, then the left (right) sibling of v also contains a link to the left (right) sibling of u, unless there are no such nodes. (iii) Every non-leaf node has two adjacent nodes in the inorder traversal, which are leaves.*

A Weight-Balanced Overlay. The overlay consists of two levels. The upper level of the overlay is a Perfect Binary Tree (PBT). The lower level of the overlay are the leaves of this tree, which are sets of nodes called *buckets* containing $O(\log N)$ nodes. Each bucket is structured as a doubly linked list. Each node of the bucket points to the node which is a leaf of the PBT and is called the *representative* of the bucket. Additionally, it maintains its routing table w.r.t. the nodes of all buckets.

When a node z makes a join request to v, then this node is forwarded to its adjacent leaf u w.r.t. the inorder traversal. Then, node z is added to the doubly linked list representing the bucket of u by manipulating a constant number of links. The routing table of z is updated by using Lemma 2(ii). When a node v leaves the network, then it is replaced by its right adjacent node u (if there is no right adjacent node then we choose the left one) which in turn is replaced by its first node z in its bucket. Link and data information are copied from v to u and from u to z. When a node v is discovered to be unreachable, its adjacent node u is first located. This is accomplished by traversing the path to the rightmost or leftmost leaf starting from the left or right child respectively. Node u fills the gap of v and the first child z in the bucket of u fills the gap left by u. The contents of u are not moved to another node except from the navigation data (routing tables and other links) which are moved to node z that take its place. Node u has its routing tables recomputed.

The join and departure of nodes may cause the size of the buckets to be uneven, which in the long run renders the structure unbalanced (imagine a bucket holding almost all nodes). To control the size of the buckets we employ a weight-based approach. Each node v of the PBT maintains its weight $|v|$, which is equal to the number of nodes in the buckets of its subtree. The size control is accomplished by using the method introduced in Section 2.1, in order to avoid the existence of hotspots.

The *node criticality* nc_v of a node v at level ℓ with left and right children w and z at level $\ell + 1$, respectively, is defined as $nc_v = \frac{|w|}{|v|}$. The following invariant bounds the criticality of nodes.

Invariant 4. *The node criticality of all nodes is in the range* $\left[\frac{1}{4}, \frac{3}{4}\right]$.

Invariant 4 implies that the number of nodes in buckets in the left subtree of a node v is at least $1/3$ and at most threefold the corresponding number of its right subtree (this definition can be easily generalized when v has a $O(1)$ number of children). When an update takes place at bucket x, then we locate the highest ancestor v of x whose node criticality is out of bounds, w.r.t. Invariant 4, and we redistribute the nodes in its subtree. The redistribution phase is described in [5]. The redistribution guarantees that if there are z nodes in total in the y buckets of the subtree of v, then after the redistribution each bucket maintains either $\lfloor z/y \rfloor$ or $\lfloor z/y \rfloor + 1$ nodes. However, the following discussion still holds (with minor changes) even if the redistribution phase guarantees that the minimum and maximum size of the buckets is within constant factors. The cost for the redistribution we propose for node v is $f(|v|) = O(|v|)$.

We guarantee that each bucket contains $O(\log N)$ nodes when subject to joins or departures of nodes by employing two operations on the PBT, the *contraction* and the *extension*. When a redistribution takes place at the root of the PBT, we also check whether any of these two operations can be applied to the PBT. The extension operation adds one more level of nodes at the PBT from existing nodes in the buckets, thus increasing its height by one. The contraction operation removes one level of nodes from the PBT and puts them into the buckets, thus decreasing its height by one. In order to decide whether the PBT needs extension or contraction we compare the size of the buckets B after the redistribution with the height of the PBT. Note that after redistribution, the sizes of all buckets may differ by at most 1. If the size is larger by at least 1 then an extension takes place. If the size of the bucket is smaller than the height of the PBT by at least 1 then a contraction takes place. The height of the PBT can be deduced by the size of the routing table in the nodes of the last level of the PBT. These two operations involve a reconstruction of the overlay which rarely happens as shown in the following lemma.

Lemma 3. *If a redistribution operation is performed at a node with weight* s*, then this node will be redistributed again after* $\Omega(s)$ *joins or departures have been performed in its subtree.*

Lemma 3 states that the expensive operations of extension and contraction take place when the number of nodes has at least doubled or halved. Assuming that the redistribution of v has $O(f(|v|))$ cost, it follows by Lemma 3 that the amortized cost for join/departure of a node v at height h is $O\left(\frac{f(|v|)}{|v|}\right)$. Since the PBT has height H, we establish the following.

Lemma 4. *The amortized cost of join/departure of a node v is $O\left(H\frac{f(N)}{N}\right)$.*

$O(1)$ Space per Node. The routing tables require $O(\log N)$ space for each node. To make the space consumption constant, one could apply on the overlay the schemes described in [8,15]. However, on the one hand the complexities will not be deterministic while on the other hand even in the case of the strong rainbow graphs [8] with deterministic bounds our congestion for searching is much better than theirs. To achieve constant space we distribute the routing tables to many nodes doing the same also for nodes in the buckets. A set of nodes with constant degree is grouped together and a routing table is distributed on all these nodes, such that each node uses constant space. Thus, a node can recreate approximately its routing table by accessing nodes inside the same group. We call each such group a *hypernode.*

A hypernode at level ℓ consists of at most ℓ nodes, numbered from left to right $1, 2, \ldots$. This number is the *rank* of the node within the hypernode. A node v with rank i maintains two links to the nodes that are approximately 2^i positions to the right and to the left. In particular, node v either points to a node z in the same hypernode whose distance is 2^i or to a node z' whose rank is i and lies in a different hypernode than that of v which contains a node whose distance is 2^i from v. The concatenation of all such links constitutes the routing table for the hypernode. Additionally, each node with rank i maintains two links to nodes with ranks $i-1$ and $i+1$, if there are such nodes. Finally, each node with rank i in the hypernode maintains a link to the node with the largest rank. The following lemma translates Lemma 2(ii) in the setting of hypernodes.

Lemma 5. *If node v contains a link to node u, then the left (right) sibling of v also contains a link to the left (right) sibling of u, unless $\nexists$ such nodes.*

Using Lemma 5 we can update the links of a node v by simply looking at the links of its siblings u and w and update the links of v by pointing to the adjacent nodes of the nodes pointed to by u and w. Hypernodes are static in the overlay and only in the case of contraction we destroy the hypernodes of the last level while in the case of extension we create new hypernodes for the new level. A faulty node inside a hypernode will not disconnect it since by accessing the parents we can find its siblings and reconstruct the missing routing information.

3.2 The Index Structure of the D^2-Tree

The overlay provides the infrastructure for the index to efficiently support various operations. The overlay is used as a node-oriented tree. The range of all values stored in the overlay is partitioned into subranges each one of which is assigned to a node of the overlay. An internal node v with range $[x_v, x'_v]$ may have a left child u and a right child w with ranges $[x_u, x'_u]$ and $[x_w, x'_w]$ respectively such that $x_u < x'_u < x_v < x'_v < x_w < x'_w$. Thus, if an element $x \in [x_v, x'_v]$ then it must be stored at node v. Ranges are dynamic in the sense that they depend on the values maintained by the node.

Search and Range Queries. The search for an element α in the overlay may be initiated from any node v at level ℓ. Let z be the node with range of values containing α. Assume $O(\log N)$ space per node and assume that w.l.o.g $x'_v < \alpha$. Then, by using the routing tables we search at level ℓ for a node u with right sibling w (if there is such sibling) such that $x'_u < \alpha$ and $x_w > \alpha$ unless α is in the range of u and the search terminates. This step has $O(\ell)$ cost, since we simulate a binary search. If the search continues, then node z will either be an ancestor of u or in the subtree rooted at u. If u is a leaf, then we move upwards (or in its corresponding bucket) until we find node z in $O(\log N)$ steps. If u is an internal node, by following the respective link we move to the left adjacent node y of u which is certainly a leaf (inorder traversal). If $x'_y > \alpha$ then an ordinary top down search from node u will suffice to find z in $O(\log N)$ steps (or in its bucket). Otherwise, node z is certainly an ancestor of u and thus we can move upwards from u until we find it in $O(\log N)$ steps. The case with $O(1)$ space per node, along with the proof of the following lemma, are given in [5].

Lemma 6. *The search for an element α in a D^2-tree of N nodes is carried out in $O(\log N)$ steps.*

A range query $[a, b]$ reports all elements x such that $x \in [a, b]$. A range query $[a, b]$ initiated at node v, invokes a search operation for element a. Node u that contains a returns to v all elements in this range. If all elements of u are reported then the range query is forwarded to the right adjacent node (inorder traversal) and continues until an element larger than b is reached for the first time.

Updates and Load Balancing. Assume that an update operation is initiated at node v involving element α. By invoking a search operation we locate node u with range containing element α. Finally, the update operation is performed on u. The main issue is how to balance the load to all nodes of the overlay as much equally as possible. To do that we employ the machinery developed in Section 2. Details can be found in [5].

The cost for the redistribution of a node v is $O(|v| \log N)$ for the case of $O(\log N)$ space per node or $O(|v|)$ for the case of $O(1)$ space per node. This is because, during the transfer of elements the routing tables must be reconstructed. The following lemma states that the load balancing is efficient in an amortized sense when the structure is subject to insertions and deletions of elements. It is a direct implication of Theorem 2 and the space used by the nodes.

Lemma 7. *The load rebalancing operation of the index has an amortized cost of $O(\log N)$.*

One final comment is that the redistribution of elements may be affected by the redistribution of nodes in the weight-balanced overlay. In order to avoid such a phenomenon, the redistribution of nodes in the subtree of node v in the overlay is preceded by a redistribution of elements.

3.3 Other Efficiency Issues and the Main Result

We are now ready to tackle the congestion and the fault-tolerance of the D^2-tree overlay, and to present the main results of this work.

Congestion. We assume that a sequence of searches $s_1, s_2, \ldots, s_N$ is initiated from each of the N nodes of the overlay. Assume that s_i is looking for an element residing in a node z_i (target node for s_i). The target nodes $z_1, z_2, \ldots, z_N$ are chosen independently and uniformly at random from all nodes of the overlay. There are two phases in the search. The first is the horizontal search, which makes use of the routing tables, and the second is the vertical search on a path from a node either towards the root or towards a leaf. The following theorem, whose proof can be found in [5], establishes the congestion bound.

Theorem 3. *The congestion due to the search operations is $O(\frac{\log N}{N})$ expected in a D^2-tree with N nodes, where each node uses $O(1)$ space.*

Fault Tolerance. If a node v discovers (during the execution of an operation) that node u is unreachable, then it contacts a sibling of u through the routing tables of the siblings of v (by making use of Lemma 2(ii)). This sibling of u is able by Lemma 2(ii) (or Lemma 5) to reconstruct all links of node u and a node departure for u is initiated, which resolves this failure. A more extensive discussion can be found in [5].

Main Result. We are now ready to establish the main results of this work stated in the Introduction and in Table 1. In particular, space usage is $O(1)$ by construction. The search cost follows from Lemma 6, which also dominates the cost for updating a data element. Node join and departures are $O(\log N)$ amortized by Lemma 4 and the fact that $f(n) = O(N)$. The congestion bound comes from Theorem 3. Finally, the load-balancing bound comes from Lemma 7.

4 Conclusions and Discussion

The load-balancing scheme can be applied straightforwardly to BATON [10]. BATON is a balanced tree-like overlay that satisfies the specifications set in the Introduction. The same goes also for Skip Graphs [4] with the exception that the specifications hold probabilistically and thus the bounds are also probabilistic. Additionally, it provides a mechanism to control the bucket size of [3].

We provide a technique that lazily updates the weights on the nodes of a tree. This technique is interesting by itself and can be straightforwardly applied to weighted balanced trees [2] in the Pointer Machine model of computation for single processor internal memory machines. In this manner, the update of balancing information is supported in $O(1)$ amortized time.

References

1. Arge, L., Eppstein, D., Goodrich, M.T.: Skip-Webs: Efficient Distributed Data Structures for Multidimensional Data Sets. In: Proc. of the 24th PODC, pp. 69–76 (2005)
2. Arge, L., Vitter, J.: Optimal External Memory Interval Management. SIAM Journal on Computing 32(6), 1488–1508 (2003)
3. Aspnes, J., Kirsch, J., Krishnamurthy, A.: Load-balancing and Locality in Range-Queriable Data Structures. In: Proc. of the 23rd PODC, pp. 115–124 (2004)
4. Aspnes, J., Shah, G.: Skip Graphs. In: Proc. of the 14th SODA, pp. 384–393 (2003)
5. Brodal, G.S., Sioutas, S., Tsichlas, K., Zaroliagis, C.: D^2-Tree: A New Overlay with Deterministic Bounds (September 2010), `http://arxiv.org/abs/1009.3134`
6. Li, D., Cao, J., Lu, X., Chan, K.C.C.: Efficient Range Query Processing in Peer-to-Peer Systems. IEEE Transactions on Knowledge and Data Engineering 21(1), 78–91 (2009)
7. Gasenan, P., Bawa, M., Garcia-Molina, H.: Online Balancing of range-Partitioned Data with Applications to Peer-to-Peer Systems. In: Proc. of the 13th VLDB, pp. 444–455 (2004)
8. Goodrich, M.T., Nelson, M.J., Sun, J.Z.: The Rainbow Skip Graph: A Fault-Tolerant Constant-Degree Distributed Data Structure. In: Proc. of the 17th SODA, pp. 384–393 (2006)
9. Harvey, N., Munro, J.I.: Deterministic SkipNet. In: Proc. of the 22nd PODC, pp. 152–153 (2003)
10. Jagadish, H.V., Ooi, B.C., Vu, Q.H.: BATON: a Balanced Tree Structure for Peer-to-Peer Networks. In: Proc. of the 31st VLDB, pp. 661–672 (2005)
11. Karger, D., Kaashoek, F., Stoica, I., Morris, R., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: Proc. of the SIGCOMM, pp. 149–160 (2001)
12. Manku, G.S., Bawa, M., Raghavan, P.: Symphony: Distributed hashing in a small world. In: 4th USENIX Symp. on Internet Technologies and Systems (2003)
13. Manku, G.S., Naor, M., Wieder, U.: Know thy Neighbor's Neighbor: the Power of Lookahead in Randomized P2P Networks. In: Proc. of the 36th STOC, pp. 54–63 (2004)
14. Rowstron, A., Druschel, P.: Pastry: A Scalable, Decentralized Object Location, and routing for large-scale peer-to-peer systems. In: Liu, H. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
15. Zatloukal, K.C., Harvey, N.J.A.: Family trees: An Ordered Dictionary with Optimal Congestion, Locality, Degree and Search Time. In: Proc. of the 15th SODA, pp. 301–310 (2004)
16. Zhang, Y., Liu, L., Li, D., Liu, F., Lu, X.: DHT-Based Range Query Processing for Web Service Discovery. In: Proc. of the 2009 IEEE ICWS, pp. 477–484 (2009)
17. Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D., Kubiatowicz, J.D.: Tapestry: A Resilient Global-scale Overlay for Service Deployment. IEEE Journal on Selected Areas in Communications 22(1), 41–53 (2004)