D^3-Tree: A Dynamic Deterministic Decentralized Structure

Spyros Sioutas[3], Efrosini Sourla[1], Kostas Tsichlas[4], and Christos Zaroliagis[1,2]

[1] Dept of Computer Eng. & Informatics, Univ. of Patras, 26504 Patras, Greece
{sourla,zaro}@ceid.upatras.gr
[2] Computer Technology Institute & Press "Diophantus", N. Kazantzaki Str., Patras University Campus, 26504 Patras, Greece
[3] Dept of Informatics, Ionian University, 49100 Corfu, Greece
sioutas@ionio.gr
[4] Dept of Informatics, Aristotle Univ. of Thessaloniki, 54124 Thessaloniki, Greece
tsichlas@csd.auth.gr

Abstract. We present D^3-Tree, a dynamic deterministic structure for data management in decentralized networks, by engineering and further extending an existing decentralized structure. D^3-Tree achieves $O(\log N)$ worst-case search cost (N is the number of nodes in the network), $O(\log N)$ amortized load-balancing cost, and it is highly fault-tolerant. A particular strength of D^3-Tree is that it achieves $O(\log N)$ amortized search cost under massive node failures. We conduct an extensive experimental study verifying that D^3-Tree outperforms other well-known structures and that it achieves a significant success rate in element queries in case of massive node failures.

1 Introduction

Decentralized systems are ubiquitous and are encountered in various forms and structures. They are widely used for sharing resources and store very large data sets, using systems of small computers instead of large costly servers. Typical examples include cloud computing environments, peer-to-peer systems and the internet. In decentralized systems, data are stored at the network nodes and the most crucial operations are data search and data updates. A decentralized system is typically represented by a graph, a logical *overlay network*, where its N nodes correspond to the network nodes, while its edges may not correspond to existing communication links, but to communication paths. The complexity (cost) of an operation is measured in terms of the number of messages issued during its execution (internal computations at nodes are considered insignificant).

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). Moreover, the overlay implements an *indexing scheme* for the stored data, supporting the operations *Insert* (a new element), *Delete* (an existing element), *Search* (for an element), and *Range Query* (for elements in a specific range).

© Springer-Verlag Berlin Heidelberg 2015
N. Bansal and I. Finocchi (Eds.): ESA 2015, LNCS 9294, pp. 989–1000, 2015.
DOI: 10.1007/978-3-662-48350-3_82

Related Work. Considerable work has been done recently in order to build efficient decentralized systems with effective distributed search and update operations. In general, decentralized networks 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, CAN, Pastry, Symphony, Tapestry (see [7] for an overview), Pagoda [1], SHELL [9] and P-Ring [3]. In general, DHT-based systems support exact match queries well and use (successfully) probabilistic methods to distribute the workload among nodes equally. 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-neighbour and string prefix queries). Some efforts towards addressing range queries have been made in [4,8], getting however approximate answers and also making exact searching highly inefficient. Pagoda [1] achieves constant node degree but has polylogarithmic complexity for the majority of operations. SHELL [9] maintains large routing tables of $O(\log^2 N)$ space complexity, but achieves constant amortized cost for the majority of operations. Both are complicated hybrid structures and their practicality (especially concerning fault tolerant operations) is questionable. The most recent effort towards range queries is the P-Ring [3], a fully distributed and fault-tolerant system that supports both exact match and range queries, achieving $O(\log_d N + k)$ range search performance (d is the *order*[1] of the ring and k is the answer size). It also provides load-balancing by maintaining a load imbalance factor of at most $2 + \epsilon$ of a stable system, for any given constant $\epsilon > 0$, and has a stabilization process for fixing inconsistencies caused by node failures and updates, achieving an $O(d \cdot \log_d N)$ performance for load-balancing.

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 Family Trees [7], BATON [6], BATON* [5] and Skip List-based schemes like Skip Graphs (SG), NoN SG, SkipNet (SN), Bucket SG, Skip Webs, Rainbow Skip Graphs (RSG) and Strong RSG [7] that use randomized techniques to create and maintain the hierarchical structure. We should emphasize that w.r.t. load-balancing, the solutions provided in the literature are either heuristics, or provide expected bounds under certain assumptions, or amortized bounds but at the expense of increasing the memory size per node. In particular, in BATON [6], a decentralized overlay is provided with load-balancing based on data migration. However, their $O(\log N)$ amortized bound is valid only subject to a probabilistic assumption about the number of nodes taking part in the data migration process, and thus it is in fact an amortized expected bound. Moreover, its successor BATON*[5], exploits the advantages of higher *fanout* (number of children per node), to achieve reduced search cost of $O(\log_m N)$, where m is the fanout.

[1] Maximum fanout of the hierarchical structure on top of the ring. At the lowest level of the hierarchy, each node maintains a list of its first d successors in the ring.

However, the higher fanout leads to a higher update and load-balancing cost of $O(m \cdot \log_m N)$. Recently, a deterministic decentralized tree structure, called D^2-Tree [2], was presented that overcomes many of the aforementioned weaknesses of tree-based systems. In particular, D^2-Tree achieves $O(\log N)$ searching cost, amortized $O(\log N)$ update cost both for element updates and for node joins and departures, and deterministic amortized $O(\log N)$ bound for load-balancing. Its practicality, however, has not been tested so far.

Regarding fault tolerance, P-Ring [3] is considered highly fault-tolerant, using the Chord's Fault Tolerant Algorithms [11]. BATON [6] maintains vertical and horizontal routing information not only for efficient search, but to offer a large number of alternative paths between two nodes. In BATON* [5], fault tolerance is greatly improved due to higher fanout. D^2-Tree can tolerate the failure of a few nodes, but cannot afford a massive number of $O(N)$ node failures.

Our Contribution. In this work, we focus on hierarchical tree-based decentralized systems and introduce D^3-Tree (cf. Section 2), a dynamic deterministic decentralized structure. D^3-Tree is an extension of D^2-Tree [2] that adopts all of its strengths and extends it in two respects: it introduces an enhanced fault-tolerant mechanism and it is able to answer efficiently search queries when massive node failures occur. D^3-Tree achieves the same deterministic (worst-case or amortized) bounds as D^2-Tree for search, update and load-balancing operations, and answers search queries in $O(\log N)$ amortized cost under massive node failures. A comparison of D^3-Tree with state-of-the-art decentralized structures is given in Table 1. Note that all previous structures provided only empirical evidence of their capability to deal with massive node failures; no previous structure provided a theoretical guarantee for searching in such a case.

Our second contribution is an implementation of the D^3-Tree and a subsequent comparative experimental evaluation (cf. Section 3) with its main competitors BATON, BATON*, and P-Ring. Our experimental study verified the theoretical results (as well as those of the D^2-Tree) and showed that D^3-Tree outperforms other state-of-the-art hierarchical tree-based structures. Our experiments demonstrated

Table 1. Comparison of BATON, BATON*, P-Ring, D^2-Tree, and D^3-Tree.

Structures	Search	Search with massive failures Theor.	Exp.	Node Updates (updating rout. tables)	Element Updates (load balancing)
BATON	$O(\log N)$	—	Yes	$\overline{O}(\log N)$	$\overline{O}(\log N)$
BATON*	$O(\log_m N)$	—	Yes	$\overline{O}(m \cdot \log_m N)$	$\overline{O}(m \cdot \log_m N)$
P-Ring	$O(\log_d N)$	—	Yes	$\widetilde{O}(d \cdot \log_d N)$	$\widetilde{O}(d \cdot \log_d N)$
D^2-Tree	$O(\log N)$	—	No	$\widetilde{O}(\log N)$	$\widetilde{O}(\log N)$
D^3-Tree	$O(\log N)$	$\widetilde{O}(\log N)$	Yes	$\widetilde{O}(\log N)$	$\widetilde{O}(\log N)$

N: number of nodes; m: fanout; d: order of the ring; $\widetilde{O}$: amortized bound; $\overline{O}$: expected amortized bound; Theor: theoretical bound; Exp: empirical evidence.

that D^3-Tree has a significantly small redistribution rate (structure redistributions after node joins or departutes), while element load-balancing is rarely necessary. We also investigated the structure's fault tolerance in case of massive node failures and show that it achieves a significant success rate in element queries. Omitted details can be found in [10].

2 The D^3-Tree

In this section, we present D^3-Tree. A key feature is the weight-based mechanism (adopted from [2]), used for node redistribution after node updates and data load-balancing after element updates. The main idea is the almost equal distribution of elements among nodes, using *weights*, a metric showing how uneven is the load among nodes. The mechanism lazily updates the weight information on nodes, so load-balancing is performed only when it is absolutely necessary.

The new features of D^3-Tree are its enhanced fault-tolerant and search mechanisms, in case of node failures. The enhanced search operation is successful even when a considerable number of nodes fails. D^3-Tree is highly fault tolerant, since it supports a procedure of *node withdrawal* when a node is found unreachable, regardless of its position (internal node, leaf, bucket node). The success of these two operations is due to a small number of additional links a node maintains, through which it can reconstruct the routing table of a failed node.

2.1 The Structure

Let N be the number of nodes present in the network and let n denote the size of data elements residing in the nodes ($N \ll n$). The structure consists of two levels. The upper level is a Perfect Binary Tree (PBT) of height $O(\log N)$. The leaves of this tree are *representatives* of the *buckets* that constitute the lower level of the D^3-Tree. Each bucket is a set of $O(\log N)$ nodes which are structured as a doubly linked list.

Each node v of the D^3-Tree maintains an additional set of links (described below) to other nodes apart from the standard links which form the tree. The first four sets are inherited from the D^2-Tree, while the fifth set is a new one that contributes in establishing a better fault-tolerance mechanism.

1. Links to its father and its children.
2. Links to its adjacent nodes based on an in-order traversal of the tree.
3. Links to nodes at the same level as v. The links are distributed in exponential steps; 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 and require $O(\log N)$ space per node.
4. Links to leftmost and rightmost leaf of its subtree. These links accelerate the search process and contribute to the structure's fault tolerance when a considerable number of nodes fail.

5. For leaf nodes only, links to the buckets of the nodes in their routing tables. The first link points to a bucket 2^0 positions 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 require $O(\log N)$ space per node and keep the structure fault tolerant, since each bucket has multiple links to the PBT.

The next lemma [2] captures some important properties of the routing tables.

Lemma 1. *(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 in-order traversal, which are leaves.*

Regarding the index structure of the D^3-Tree, the range of all values stored in it is partitioned into sub-ranges 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 $x_u < x'_u < x_v < x'_v < x_w < x'_w$. Ranges are dynamic in the sense that they depend on the values maintained by the node.

2.2 Node Joins and Departures

When a node z makes a join request to v, v forwards the request to an adjacent leaf u. If v is a PBT node, the request is forwarded to the left adjacent node, w.r.t. the in-order traversal, which is definitely a leaf (unless v is a leaf itself). In case v is a bucket node, the request is forwarded to the bucket representative, which is leaf. Then, node z is added to the doubly linked list of the bucket represented by u. In node joins, we make the simplification that the new node is clear of elements and we place it after the most loaded node of the bucket. Thus, the load is shared and the new node stores half of the elements of the most loaded node.

When a node v leaves the network, it is replaced by an existing node, so as to preserve the in-order adjacency. All navigation data are copied from the departing node v to the replacement node, along with the elements of v. If v is an internal PBT node, then it is replaced by its right adjacent node, which is a leaf and which in turn is replaced by the first node z in its bucket. If v is a leaf, then it is directly replaced by z. Then v is free to depart.

After a node join or departure, the modified weight-based mechanism [2] is activated and updates the sizes by ± 1 on the path from the leaf u to the root. Afterwards, the mechanism traverses the path from u to the root, in order to find the first unbalanced node (if such a node exists) and performs a *redistribution* in its subtree. The redistribution guarantees that if there are x nodes in total in the y buckets of the subtree of v, then after the redistribution each bucket maintains either $\lfloor x/y \rfloor$ or $\lfloor x/y \rfloor + 1$ nodes. The redistribution cost is $O(\log N)$ [2], which is indeed verified by our experiments.

The redistribution of nodes in the subtree of v starts from the rightmost bucket b and it is performed in an in-order fashion so that elements in the nodes are not affected. The transfer of nodes is accomplished by maintaining a link, called *dest*, to the bucket representative b' in which nodes should be put or taken from. In case b has q extra nodes, the nodes are removed from b and are added to b'. Finally, bucket b informs b' to take over and the same procedure applies again with b' as the source bucket. The case where q nodes must be transferred to bucket b from bucket b' is completely symmetric.

Throughout joins and departures of nodes, the size of buckets can increase undesirably or can decrease so much that some buckets may become empty. The structure guarantees that each bucket contains $O(\log N)$ nodes, throughout joins or departures of nodes, by employing two operations on the PBT, the *contraction* and the *extension*.

2.3 Single and Range Queries

The search for an element a may be initiated from any node v at level l. If v is a bucket node, then if its range contains a the search terminates, otherwise the search is forwarded to the bucket representative, which is a binary node. If v is a PBT node, then let z be the node with range of values containing a, $a \in [x_z, x'_z]$ and assume w.l.o.g. that $x'_v < a$. The case where $x_v > a$ is completely symmetric. First, we perform a horizontal binary search at the level l of v using the routing tables, searching for a node u with right sibling w (if there is such sibling) such that $x'_u < a$ and $x_w > a$.

Having located nodes u and w, the horizontal search is terminated and a vertical search is initiated. Node z will either be the common ancestor of u and w, or it will be in the right subtree rooted at u, or in the left subtree rooted at w. Node u contacts the rightmost leaf y of its subtree. If $x_y > a$ then an ordinary top down search from node u will suffice to find z. Otherwise, node z is in the bucket of y, or in its right in-order adjacent (this is also the common ancestor of u and w), or in the subtree of w.

When z is located, if a is found in z then the search was successful, otherwise a is not stored in the structure. The search for an element a is carried out in $O(\log N)$ steps [2], and it is indeed verified by our experiments.

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

2.4 Element Insertions and Deletions

Assume that an update operation (insertion/deletion) is initiated at node v involving element a. By invoking a search operation, node u with range containing element a is located and the update operation is performed on u.

In order to apply the weight-based mechanism for load balancing, the element should be inserted in a bucket node (similar to node joins) or in a leaf. If u is an internal node of the PBT, then element a is inserted in u and then the first element of u (note that elements into nodes are sorted) is removed from u and it is inserted into node q, which is the last node of the bucket of the left adjacent of u, in order to preserve the sequence of elements in the in-order traversal. This way, the insertion has been shifted to a bucket node. The case of element deletion is similar.

After an element update in leaf u or in its bucket, the weight-based mechanism is activated and updates the weights by ± 1 on the path from leaf u to the root. Afterwards, the mechanism traverses the path from leaf u to the root, in order to find the first node (if such a node exists) which is unbalanced and performs a load-balancing in its subtree.

The load-balancing mechanism guarantees that if there are $w(v)$ elements in total in the subtree of v of size $|v|$ (total number of nodes in the subtree of v including v), then after load-balancing each node stores either $\left\lfloor \frac{w(v)}{|v|} \right\rfloor$ or $\left\lfloor \frac{w(v)}{|v|} \right\rfloor + 1$ elements. The load-balancing cost is $O(\log N)$ [2], which is indeed verified by our experiments. The load-balancing mechanism is similar to the redistribution mechanism described above, so its description is omitted.

2.5 Fault Tolerance

Searches and updates in the D^3-Tree do not tend to favour any node, and in particular nodes near the root. However, a single node can be easily disconnected from the overlay, when all nodes with which it is connected fail. This means that 4 failures (two adjacent nodes and two children) are enough to disconnect the root. The most easily disconnected nodes are those which are near the root, since their routing tables are small in size.

When a node w discovers that v is unreachable, the network initiates a *node withdrawal* procedure by reconstructing the routing tables of v, in order for v to be removed smoothly, as if v was departing. If v belongs to a bucket, it is removed from the structure and the links of its adjacent nodes are updated. In case v is an internal binary node, its right adjacent node u is first located, making use of Lemma 1, in order to replace v.

If v is a leaf, then it should be replaced by the first node u in its bucket. In the D^2-Tree, if a leaf was found unreachable, contacting its bucket would be infeasible, since the only link between v and its bucket would have been lost. This weakness was eliminated in the D^3-Tree, by maintaining multiple links towards each bucket, distributed in exponential steps (in the same way as the horizontal adjacency links). This way, when w is unable to contact v, it contacts directly the first node of its bucket u and u replaces v. Regardless of node's v position in the structure, the elements stored in v are lost.

2.6 Single Queries with Node Failures

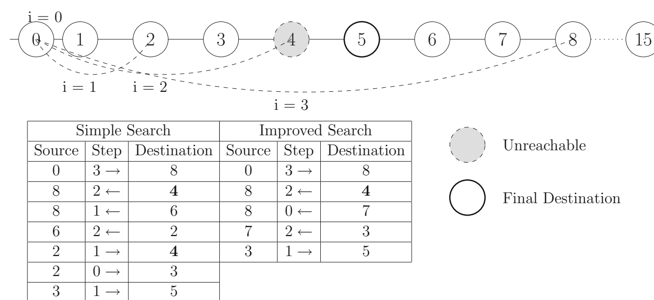
In a network with node failures, an unsuccessful search for element a refers to the cases where either z (the node with range of values containing a, i.e., $a \in [x_z, x'_z]$) is unreachable, or there is a path to z but the search algorithm can not follow it to locate z due to failures of intermediate nodes. The D^2-Tree provides a preliminary fault-tolerant mechanism that succeeds only in the case of a few node failures. That mechanism cannot deal with massive node failures (also known as churn), i.e., its search algorithm may fail to locate a. In the following, we present the key features of our D^3-Tree efficient search algorithm in case of massive node failures.

The search procedure is similar to the simple search described in Section 2.3. One difference in horizontal search lies in the fact that if the most distant right adjacent of v is unreachable, v keeps contacting its right adjacent nodes by decreasing the step by 1, until it finds node q which is reachable.

In case $x'_q < a$ the search continues to the right using the most distant right adjacent of q, otherwise the search continues to the left and q contacts its most distant left adjacent p which is in the right of v. If p is unreachable, q doesn't decrease the travelling step by 1, but contacts directly its nearest left adjacent (at step $= 0$) and asks it to search to the left. This improvement reduces the number of messages that are meant to fail, because of the exponential positions of nodes in routing tables and the nature of binary horizontal search. For example, in Fig. 1, the search starts from v_0 and v_8 contacts v_7, since v_4 has failed. No node contacts v_4 from then onwars and the number of messages is reduced by 2.

A vertical search to locate z is always initiated between two siblings u and w, which are either both active, or one of them is unreachable, as shown in Fig. 2 where the left sibling u is active and w, the right one, is unreachable. In both cases, first we search into the subtree of the active sibling, then we contact the common ancestor and then, if the other sibling is unreachable, the active sibling tries to contact its corresponding child (right child for left sibling and left child for right sibling). When the child is found the search is forwarded to its subtree.

In general, when node u wants to contact the left (right) child of unreachable node w, the contact is accomplished through the routing table of its own left



Simple Search			Improved Search		
Source	Step	Destination	Source	Step	Destination
0	3 →	8	0	3 →	8
8	2 ←	4	8	2 ←	4
8	1 ←	6	8	0 ←	7
6	2 ←	2	7	2 ←	3
2	1 →	4	3	1 →	5
2	0 →	3			
3	1 →	5			

Fig. 1. Example of binary horizontal search with node failures

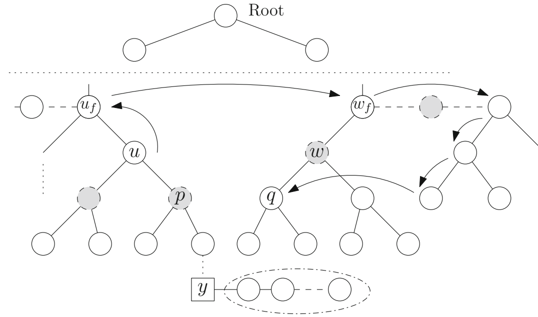


Fig. 2. Example of vertical search between u and unreachable w

(right) child. If its child is unreachable (Fig. 2), then u contacts its father u_f and u_f contacts the father of w, w_f, using Lemma 1(i). Then w_f, using Lemma 1(ii) twice in succession, contacts its grandchild through its left and right adjacents and their grandchildren.

In case initial node v is a bucket node, then if its range contains a the search terminates, otherwise the search is forwarded to the bucket representative. If the bucket representative has failed, the bucket contacts its other representatives right or left, until it finds a representative that is reachable. The procedure continues as described above for the case of a binary node.

The following lemma gives the amortized upper bound for the search cost in case of massive failures of $O(N)$ nodes.

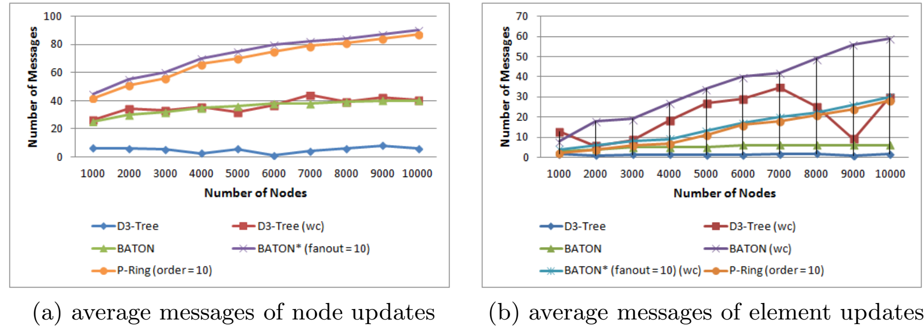
Lemma 2. *The amortized search cost in case of massive node failures is* $O(\log N)$.

3 Experimental Study

We have built a simulator[2] with a user friendly interface and a graphical representation of the structure, to evaluate the performance of D^3-Tree. To evaluate the cost of operations, we ran experiments with different number of nodes N from 1,000 to 10,000, in order to be directly compared to BATON, BATON* and P-Ring. BATON* is a state-of-the-art decentralized architecture and P-Ring outperforms DHT-based structures in range queries and achieves a slightly better load-balancing performance compared to BATON*. For a structure of N nodes, 1000 x N elements where inserted. We used the number of passing messages to measure the performance of the system.

Cost of Node Joins/Departures: To measure the network performance for the operation of node updates, in a network of N initial nodes, we performed $2N$ node updates. In a preliminary set of experiments with mixed operations (joins/departures), we observed that redistributions rarely occurred, thus leading in negligible node update costs. Hence, we decided to perform only one type

[2] Our simulator is a standalone desktop application, developed in Visual Studio 2010, available in `https://github.com/sourlaef/d3-tree-sim`



(a) average messages of node updates (b) average messages of element updates

Fig. 3. Node and Element Update operations

of updates, $2N$ joins, that are expected to cause several redistributions. Fig. 3a shows average case (nodes where joins occur are chosen randomly) and worst case (joins occur only in the leftmost leaf), and in both cases the curves represent the average amortized redistribution cost.

We observed that even for the worst case scenario, the D^3-Tree node update and redistribution mechanism achieves a better amortized redistribution cost, compared to that of BATON, BATON* and P-Ring. In the average case, during node joins, redistribution is rarely necessary (about 3% of join operations lead to redistributions). However, in the worst case, during node joins, a great number of nodes are accumulated into the bucket of the leftmost leaf, leaving the other buckets unchanged. This naturally leads to more frequent and costly redistributions (about 9% of join operations lead to redistributions).

Cost of Element Insertions/Deletions: To measure the network performance for the operation of element updates, in a network of N nodes and n elements, we performed n element updates. In a preliminary set of experiments with mixed operations (insertions/deletions), we observed that load-balancing operations rarely occurred, thus leading in negligible node update costs. Hence, we decided to perform only one type of updates, n insertions. Fig. 3b shows average case (element insertions occur at nodes chosen randomly) and worst case (element insertions occur only in the leftmost leaf), and in both cases the curves represent the average amortized load-balancing cost.

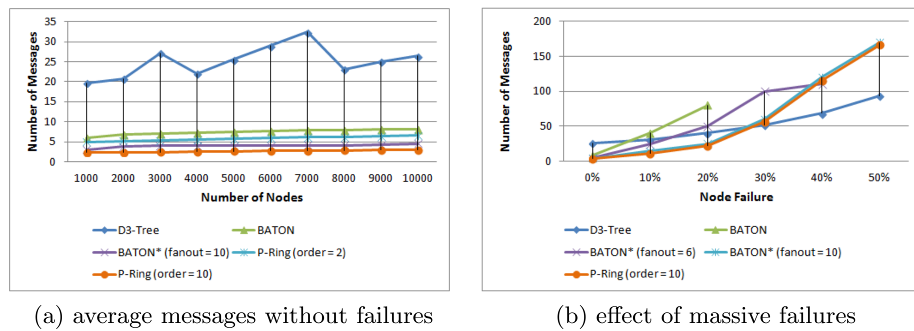
Conducting experiments, we observed that in the average case, the D^3-Tree outperforms BATON, BATON* and P-Ring. However, in D^3-Tree's worst case, the load-balancing performance is degraded compared to BATON* of $fanout$ = 10 and P-Ring. In the average case, during element insertions, load-balancing is rarely necessary (about 15% of insertions lead to load-balancing operations). However, in worst case, a great number of element insertions take place into the bucket of the leftmost leaf, leaving the other nodes unaffected, thus rendering the subtree imbalanced very often. This leads to more frequent and costly operations of load-balancing (about 50% of insertions evoke load-balancing).

Cost of Element Search with/without Node Failures. To measure the network performance for the operation of single queries, we conducted experiments in

which for each N, we performed $2M$ (M is the number of binary nodes) searches. The search cost is depicted in Fig. 4a. An interesting observation here was that although the cost of search in D^3-Tree doesn't exceed $2 \cdot \log N$, it is higher that the cost of BATON, BATON* and P-Ring. This is due to the fact that when the target node is a *Bucket* node, the search algorithm, after locating the correct leaf, performs a serial search into its bucket to locate it.

To measure the network performance for the operation of element search with node failures, we conducted experiments for different percentages of node failures: 10%, 20%, 30%, 50% and 75%. For each N and node failure percentage, we performed $2M$ searches divided into 4 groups, each of $M/2$ searches. In order to get a better estimation of the search cost, we forced a different set of nodes to fail in each group. Fig. 4b depicts the increase in search cost when massive node failures take place in D^3-Tree, BATON, different fanouts of BATON* and P-Ring. We observe that D^3-Tree maintains low search cost, compared to the other structures, even for a failure percentage $\geq 30\%$.

Describing the effect of the enhanced search mechanism of D^3-Tree in case of massive failures in more detail, we must note that when the node failure percentage is small (10% to 15%), the majority of single queries that fail are the ones whose elements belong to failed nodes. When the number of failed nodes increases, single queries are not always successful, since the search mechanism fails to find a path to the target node although the node is reachable. However, even for the significant node failure percentage of 30%, our search algorithm is 85% successful, confirming thus our claim about the structure's fault tolerance.



(a) average messages without failures (b) effect of massive failures

Fig. 4. Single Queries without/with node failures

4 Conclusions

We presented D^3-Tree, a dynamic distributed deterministic structure, that turns out to be very efficient in practice and outperforms other state-of-the-art structures. Our experimental study showed (among others) that the $O(\log N)$ amortized bound for load balancing (the most costly operation) is achieved even for the worst case scenario. Moreover, investigating the structure's fault tolerance, we showed that D^3-Tree is highly fault tolerant, since even for a substantial

amount of 30% node failures it achieves a significant success rate of 85% in element search, without increasing the search cost considerably.

Acknowledgments. This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) – Research Funding Programs Thales & Heracletus II, Investing in knowledge society through the European Social Fund.

References

1. Bhargava, A., Kothapalli, K., Riley, C., Scheideler, C., Thober, M.: Pagoda: A dynamic overlay network for routing, data management, and multicasting. In: ACM SPAA 2004, pp. 170–179 (2004)
2. Brodal, G., Sioutas, S., Tsichlas, K., Zaroliagis, C.: D^2-tree: A new overlay with deterministic bounds. Algorithmica 72(3), 860–883 (2015)
3. Crainiceanu, A., Linga, P., Machanavajjhala, A., Gehrke, J., Shanmugasundaram, J.: Load balancing and range queries in P2P systems using P-Ring. ACM Trans. Internet Technol. 10(4), Art.16, 1–16 (2011)
4. Gupta, A., Agrawal, D., Abbadi, A.E.: Approximate range selection queries in peer-to-peer systems. In: Proc. 1st Biennial Conference on Innovative Data Systems Research – CIDR (2003)
5. Jagadish, H.V., Ooi, B.C., Tan, K., Vu, Q.H., Zhang, R.: Speeding up search in P2P networks with a multi-way tree structure. ACM SIGMOD 2006, 1–12 (2006)
6. Jagadish, H.V., Ooi, B.C., Vu, Q.H.: Baton: a balanced tree structure for peer-to-peer networks. In: VLDB 2005, pp. 661–672 (2005)
7. Ozsu, M.T., Valduriez, P.: Principles of Distributed Database Systems. Springer (2011)
8. Sahin, O., Gupta, A., Agrawal, D., Abbadi, A.E.: A peer-to-peer framework for caching range queries. In: ICDE 2004, pp. 165–176 (2004)
9. Scheideler, C., Schmid, S.: A distributed and oblivious heap. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part II. LNCS, vol. 5556, pp. 571–582. Springer, Heidelberg (2009)
10. Sourla, E., Sioutas, S., Tsichlas, K., Zaroliagis, C.: D^3-tree: A dynamic distributed deterministic load–balancer for decentralized tree structures. Tech. Rep. ArXiv:1503.07905, ACM CoRR (March 2015)
11. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput. Commun. Rev. 31(4), 149–160 (2001)