Article

A Dynamic Distributed Deterministic Load-Balancer for Decentralized Hierarchical Infrastructures

Spyros Sioutas 1, Efrosini Sourla 1, Kostas Tsichlas 1, Gerasimos Vonitsanos 1,* and Christos Zaroliagis 1,2

1 Department of Computer Engineering and Informatics, University of Patras, 26504 Patras, Greece; sioutas@ceid.upatras.gr (S.S.); sourla@ceid.upatras.gr (E.S.); ktsichlas@ceid.upatras.gr (K.T.); zaro@ceid.upatras.gr (C.Z.)

2 Computer Technology Institute & Press “Diophantus”, 26504 Patras, Greece

* Correspondence: mvonitsanos@ceid.upatras.gr

Abstract: In this work, we propose $D^3$-Tree, a dynamic distributed deterministic structure for data management in decentralized networks, by engineering and extending an existing decentralized structure. Conducting an extensive experimental study, we verify that the implemented structure outperforms other well-known hierarchical tree-based structures since it provides better complexities regarding load-balancing operations. More specifically, the structure achieves an $O(\log N)$ amortized bound ($N$ is the number of nodes present in the network), using an efficient deterministic load-balancing mechanism, which is general enough to be applied to other hierarchical tree-based structures. Moreover, our structure achieves $O(\log N)$ worst-case search performance. Last but not least, we investigate the structure’s fault tolerance, which hasn’t been sufficiently tackled in previous work, both theoretically and through rigorous experimentation. We prove that $D^3$-Tree is highly fault-tolerant and achieves $O(\log N)$ amortized search cost under massive node failures, accompanied by a significant success rate. Afterwards, by incorporating this novel balancing scheme into the ART (Autonomous Range Tree) structure, we go one step further to achieve sub-logarithmic complexity and propose the ART$^+$ structure. ART$^+$ achieves an $O(\log^2 \log N)$ communication cost for query and update operations ($b$ is a double-exponentially power of 2 and $N$ is the total number of nodes). Moreover, ART$^+$ is a fully dynamic and fault-tolerant structure, which supports the join/leave node operations in $O(\log \log N)$ expected WHP (with high probability) number of hops and performs load-balancing in $O(\log \log N)$ amortized cost.

Keywords: decentralized systems; distributed data structures; P2P; load balancing; fault tolerance

1. Introduction

Decentralized systems have generated interest worldwide among the computer networking community. Although they have existed for many years, they have become prevalent nowadays and are promoted as the future of Internet networking. They are widely used for sharing resources and storing extensive data sets, using small computers instead of large, costly servers. Typical examples include cloud computing environments, node-to-node (P2P) systems, and the Internet.

According to [1], in a decentralized network, individual nodes act as both suppliers and consumers of resources, without the need for centralized coordination by servers, in contrast to the centralized client-server model, where client nodes request access to resources provided by central servers.

In decentralized systems, data are stored at the network nodes, and the most crucial operations are data search and data updates. A decentralized network is represented by a graph, a logical overlay network, where its nodes correspond to the network nodes. In contrast, its edges may not correspond to existing communication links but to communication paths.
We assume constant size messages between nodes through links and asynchronous communication. Furthermore, it is considered that the network provides an upper bound on the time needed for a node to send a message and receive an acknowledgement. In this way, the network provides a mechanism to identify communication problems, which may refer to communication links or nodes that are down. The complexity (cost) of an operation is measured by 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).

Range query processing in decentralized network environments is a complex problem to solve efficiently and scalably. In cloud infrastructures, a most significant and apparent requirement is the monitoring of thousands of computer nodes, which often requires support for range queries: consider range queries issued to identify under-utilized nodes to assign them more tasks, or to identify overloaded nodes to avoid bottlenecks in the cloud.

For example, we wish to execute range queries such as:

```sql
SELECT NodeID
FROM CloudNodes
WHERE Low < utilization < High
```

or both single and range queries such as:

```sql
SELECT NodeID
FROM CloudNodes
WHERE Low < utilization < High AND os = UNIX
```

Moreover, in cloud infrastructures that support social network services like Facebook, stored user profiles are distributed in several nodes and we wish to retrieve user activity information, executing range queries such as:

```sql
SELECT COUNT(userID)
FROM CloudNodes
WHERE 3/1/2015 < time < 3/31/2015 AND userID = 123456
AND NodeID IN Facebook
```

An acceptable solution for processing range queries in such large-scale decentralized environments must scale in terms of the number of nodes and the number of data items stored.

For very large volume data (trillions of data items at millions of nodes), the classic logarithmic complexity offered by solutions in literature is still too expensive for single and range queries.

Further, all available solutions incur large overheads concerning other critical operations, such as join/leave of nodes and insertion/deletion of items.

Our aim in this work is to provide a solution that is comprehensive and outperforms related work for all primary operations, such as search of elements, join/departure of nodes, insertion/deletion of elements, and load-balancing and to the required routing state that must be maintained to support these operations. In particular, our ultimate goal is to achieve a sub-logarithmic complexity for all the above processes.
2. Related Work

Extensive work has been done to support search and update techniques in distributed data essential operations in decentralized networks. In general, decentralized 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, CAN, Pastry, Symphony, Tapestry [2], and P-Ring [3]. DHT-based systems generally 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 [4,5], getting however approximate answers and also making exact searching highly inefficient.

Pagoda [6] achieves constant node degree but has polylogarithmic complexity for the majority of operations. SHELL [7] 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.

P-Ring [3] is 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 (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.) 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 $O(d \cdot \log N)$ performance for load-balancing.

FMapper [8] is a novel distributed read mapper employing asynchronous I/O and parallel computations between different compute elements of the core groups and uses dynamic task scheduling and asynchronous data transfers. However, this highly scalable structure supports optimal scheduling and doesn’t investigate either range query, since it’s a hashed-based system, nor the load balancing performance of the whole cluster.

Tree-based systems are based on hierarchical structures. As a result, they support range queries more naturally and efficiently and a broader 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 [2], BATON [9], BATON* [10], SPIS [11] and Skip List-based schemes like Skip Graphs (SG), NoN SG, SkipNet (SN), Deterministic SN, Bucket SG, Skip Webs, Rainbow Skip Graphs (RSG) and Strong RSG [2] 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 [9], a decentralized overlay is provided with load-balancing based on data migration. However, their $O(\log N)$ amortized bound ($N$ is the number of nodes in the network) is valid only subject to a probabilistic assumption about the number of nodes taking part in the data migration process. Thus it is an amortized expected bound. Moreover, its successor BATON*, 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. However, the higher fanout leads to larger update and load-balancing cost of $O(m \cdot \log_m N)$.

SPIS [11] is the decentralized version of Interpolation Search Tree (IST), the most recent version of which has been presented recently in [12], and its concurrent version in [13]. It exploits the Apache Spark engine, the state-of-the-art solution for big data management and analytics. In particular, it is a Spark-based indexing scheme that supports range queries in such large-scale decentralized environments and scales well w.r.t. the number of nodes.
and the data items stored. Towards this goal, there have been solutions in the last few years, which, however, turn out to be inadequate at the envisaged scale since the classic linear or even the logarithmic complexity (for point queries) is still too expensive. In contrast, range query processing is even more demanding. Therefore, SPIS goes one step further and supports range query processing with sub-logarithmic complexity. Also, the weak point of this structure is the load balancing and fault tolerance after massive failures. For more details, see [14].

In the last decade, a deterministic decentralized tree structure, called $D^2$-Tree [15], was presented that overcomes many of the weaknesses mentioned above 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.

As far as network’s fault tolerance is concerned, P-Ring [3] is considered highly fault-tolerant, using Chord’s Fault-Tolerant Algorithms [16]. BATON [9] maintains vertical and horizontal routing information not only for efficient search but to offer a large number of alternative paths between two nodes. In its successor BATON* [10], fault tolerance is greatly improved due to higher fanout. When $\text{fanout} = 2$, approximately 25% of nodes must fail before the structure becomes partitioned while increasing the fanout up to 10 leads to increasing fault tolerance (60% of failed nodes partition the structure). $D^2$-Tree can tolerate the failure of a few nodes, but cannot afford a massive number of $O(N)$ node failures.

3. Our Contribution

In this work, we focus on hierarchical tree-based overlay networks that directly support range and more complex queries. We introduce a promising deterministic decentralized structure for distributed data, called $D^3$-Tree. $D^3$-Tree is an extension of $D^2$-Tree [15] that adopts all of its strengths and extends it in two respects: it introduces an enhanced fault-tolerant mechanism, and it can answer efficiently search queries when massive node failures occur. $D^3$-Tree achieves the same deterministic $O(\log N)$ worst-case bound as $D^2$-Tree for the search operations and $O(\log N)$ amortized bounds for update and load-balancing operations, using an efficient deterministic weight-based load-balancing mechanism, which is general enough to be applied to other hierarchical tree-based structures. Moreover, $D^3$-Tree answers search queries in $O(\log N)$ amortized cost under massive node failures.

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 7) 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. Furthermore, our experiments demonstrated that $D^3$-Tree has a significantly small redistribution rate (structure redistributions after node joins/departures), while element load-balancing is rarely necessary.

We also investigated the structure’s fault tolerance in case of massive node failures and showed that it achieves a significant success rate in element queries.

Afterwards, by incorporating this novel balancing scheme into ART (Autonomous Range Tree), the state-of-the-art hierarchical distributed structure presented in [17], we go one step further to achieve sub-logarithmic complexity and propose the ART+ structure.

The outer level of ART+ is built by grouping clusters of nodes, whose communication cost of query and update operations is $O(\log^2 N \log N)$ hops, where the base $b$ is a double-exponentially power of two and $N$ is the total number of nodes. Moreover, ART+ is a fully dynamic and fault-tolerant structure, which supports the join/leave node operations in
O(log log N) expected w.h.p number of hops and performs load-balancing in O(log log N) amortized cost.

Each cluster node of ART+ is organized as a D^3-Tree. A comparison of the state-of-the-art decentralized structures with our contribution is given in Table 1.

As noted earlier, preliminary parts of this work were presented in [18,19]. In this work, we further contribute, providing additional technical details regarding the key operations of D^3-Tree; node and element updates, load-balancing, search queries without/with massive node failures.

We also provide the proof for Lemma 2 about the amortized search cost in case of massive node failures. Moreover, we include a thorough presentation of the redistribution, load-balancing and extension-contraction operations, omitted in our previously published work.

We have also included 11 algorithms in pseudo-code, to clarify the algorithmic process of the most significant operations of our structure. The evaluation of D^3-Tree is further extended, including additional experiments for (a) the operations of node and element updates, taking into account different values of specific invariants of D^3-Tree (described in Section 4), and (b) the detailed behaviour of D^3-Tree in case of massive node failures.

The corresponding section also includes a brief presentation of the graphical interface of the D^3-Tree simulator.

The rest of this paper is organized as follows: The weight-based mechanism used in D^3-Tree is briefly described in Section 4. Then, Section 5 presents our proposed structure by describing the theoretical background and discussing the enhancements and implementation aspects. In Section 6 we present the ART+ structure which exploits the performance of D^3-Tree. Section 7 hosts the performance evaluation for both structures. The paper concludes in Section 8.

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

<table>
<thead>
<tr>
<th>Structures</th>
<th>Search</th>
<th>Search with Massive Failures</th>
<th>Node/Element Updates</th>
<th>Max. Size of Routing Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-Ring</td>
<td>O(log_k N)</td>
<td>—</td>
<td>Yes</td>
<td>O(log N)</td>
</tr>
<tr>
<td>BATON</td>
<td>O(log N)</td>
<td>—</td>
<td>Yes</td>
<td>O(log N)</td>
</tr>
<tr>
<td>BATON+</td>
<td>O(log_m N)</td>
<td>—</td>
<td>Yes</td>
<td>O(m · log_m N)</td>
</tr>
<tr>
<td>D^2-Tree</td>
<td>O(log N)</td>
<td>—</td>
<td>No</td>
<td>O(log N)</td>
</tr>
<tr>
<td>D^3-Tree</td>
<td>O(log N)</td>
<td>O(\tilde{O}(log N))</td>
<td>Yes</td>
<td>O(log N)</td>
</tr>
<tr>
<td>ART</td>
<td>O(\tilde{O}(log N))</td>
<td>—</td>
<td>Yes</td>
<td>O(m · log_m log N)</td>
</tr>
<tr>
<td>ART+</td>
<td>O(\tilde{O}(log^2 N))</td>
<td>—</td>
<td>Yes</td>
<td>O(N^{1/4} / log^2 N)</td>
</tr>
</tbody>
</table>

N: number of nodes, N^c: number of cluster nodes, d: order of ring, m: fanout, c > 0, b: double-exponentially power of 2, \tilde{O}: expected bound, \tilde{O}: amortized bound, O expected amortized bound.

4. A Weight-Based Load-Balancer

In this section, we describe a weight-based load-balancing mechanism that is an efficient solution for element updates in hierarchical tree-based structures. All definitions used in this section and throughout this paper, have been compiled in Table 2. The main idea of this mechanism is the almost equal distribution of elements among nodes by making use of weights, a metric that shows how uneven is the load among nodes. When the load is uneven, a data migration process is initiated to distribute the elements equally. The method has two steps; (a) first, it provides an efficient and local update of weight information in a tree when elements are added or removed at the leaves, using virtual weights, and (b) it provides an efficient load-balancing mechanism which is activated when necessary.
Algorithms 2022, 15, 96

Table 2. Symbols and Definitions.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w(v)$:</td>
<td>weight of $v$</td>
</tr>
<tr>
<td>$e(v)$:</td>
<td>number of elements residing in a node $v$</td>
</tr>
<tr>
<td>$</td>
<td>v</td>
</tr>
<tr>
<td>$d(v)$:</td>
<td>density of $v$</td>
</tr>
<tr>
<td>$c(p, q)$:</td>
<td>criticality</td>
</tr>
<tr>
<td>$nc_v$:</td>
<td>node criticality</td>
</tr>
</tbody>
</table>

Invariants 1 and 2. Moreover, a new invariant is defined, involving node criticality.

More specifically, when an element is added/removed to/from a leaf $u$ in a tree structure $T$, the weights on the path from $u$ to the root must be updated. This is a costly operation, when it is performed on every element update. Instead of updating weights every time, a new metric is defined, virtual weight $W(v)$. Assume that node $v$ lies at height $h$ and its children are $v_1, v_2, \ldots, v_k$ at height $h - 1$. $W(v)$ of $v$ is defined as the weight stored in node $v$. In particular, for node $v$ the following invariants are maintained:

**Invariant 1.** $W(v) > e(v) + (1 - \epsilon_h)(\sum_{i=1}^k W(v_i))$.

**Invariant 2.** $W(v) < e(v) + (1 + \epsilon'_h)(\sum_{i=1}^k W(v_i))$.

where $\epsilon_h = \epsilon'_h = \frac{1}{h^2}$. The constants $\epsilon_h$ and $\epsilon'_h$ are chosen such that for all nodes the virtual weight will be within a constant factor $c > 1$ of the real weight, i.e., $\frac{1}{c} \cdot w(v) < W(v) < c \cdot w(v)$.

When an update takes place at leaf $u$, the mechanism traverses the path from $u$ to the root updating the weights by $\pm 1$, until node $z$ is found 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. All weights on the path from $u$ to $v$ are recomputed; for each node $z$ on this path, its weight information is updated by taking the sum of the weights of its children plus the number of elements that $z$ carries.

Another Invariant which is maintained and is crucial for the load-balancing mechanism, involves the criticality $c(p, q) = \frac{d(p)}{d(q)}$ of two brother nodes $p$ and $q$ (representing their difference in densities). The 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.

When an update takes place at leaf $u$, weights are updated as described above. Then, the load-balancing mechanism redistributes the elements among leaves when the load between leaves is not distributed equally enough. In particular, starting from $u$, the highest ancestor $w$ is located that is unbalanced w.r.t. its 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 $v$.

The weight-based mechanism is slightly modified to be applied also in node updates. Virtual size $S(v)$ is defined and the same mechanism is applied using invariants similar to Invariants 1 and 2. Moreover, a new invariant is defined, involving node criticality.
Invariant 4. The node criticality of all nodes is in the range $[\frac{1}{4}, \frac{3}{4}]$.

Invariant 4 implies that the number of nodes in the left subtree of a node $v$ is at least half and at most twice the corresponding number of its right subtree.

The weight-based mechanism described above (and its slight modification) achieves an $O(1)$ amortized cost for weight update and $O(\log N)$ amortized cost for redistribution. It is general enough to be applied to other hierarchical tree-based structures and was proposed and thoroughly described in [15].

5. The $D^3$-Tree

In this section, we present our proposed structure, $D^3$-Tree, which introduces many enhancements over the solutions in literature and its predecessor, $D^2$-Tree [15].

In general, a $D^3$-Tree structure with $N$ nodes and $n$ data elements residing on them achieves: (i) $O(\log N)$ space per node; (ii) deterministic $O(\log N)$ searching cost; (iii) deterministic amortized $O(\log N)$ update cost both for element updates and for node joins and departures; (iv) deterministic amortized $O(\log N)$ bound for load-balancing; (v) deterministic amortized $O(\log N)$ search cost in case of massive node failures.

5.1. The Structure

Let $N$ be the number of nodes present in the network and let $n$ denote the size of data ($N \ll n$). 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.

The number of nodes of the PBT is not connected by any means to the number of elements stored in the structure. Each node $v$ of the $D^3$-Tree maintains an additional set of links 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 to 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 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.

5.2. Node Joins and Departures

5.2.1. Handling Node Updates

When a node $z$ makes a join request (Algorithm 1) to $v$, $v$ forwards the request to an adjacent leaf $u$. If $v$ is a binary 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 a 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.

**Algorithm 1 Join (Node newNode)**

Require: initialNode

1: if initialNode is BucketNode then
2:   initialNode.Representative.Join(newNode);
3: else if initialNode is InternalBinaryNode then
4:   initialNode.LeftInOrderAdjacent.Join(newNode);
5: else if initialNode is Leaf then
6:   mostLoadedNode = initialNode.FindMostLoaded();
7:   mostLoadedNode = AcceptAsAdjacent(newNode);
8:   SplitData(mostLoadedNode, newNode);
9: end if

It could be entered anywhere in the bucket, such as the first, the last, or an intermediate node, but we prefer to 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. Finally, the left and right adjacents of the newly inserted node update their links to the previous and next node, and $z$ creates new links to those nodes.

When a node $v$ leaves the network, it is replaced by an existing node, 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 binary 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. On the other hand, if $v$ is a leaf, it is directly replaced by $z$.

If the departing node belongs to a bucket, no replacement occurs, but simply all stored elements of $v$ are copied to the previous node of $v$ (or to the next node if the last does not exist). Then $v$ is free to depart.

After a node join or departure, the modified weight-based mechanism is activated and updates the sizes by $\pm 1$ on the path from leaf $u$ to the root (Algorithm 2), as long as the defined Invariants 1 or 2 do not hold. When the first node $w$ is accessed for which Invariants 1 and 2 hold, the nodes in the subtree of its child $q$ in the path, have their sizes recomputed (Algorithm 3).
Algorithm 2 UpdateVirtualSize (BinaryNode leaf, NodeUpdate operation)

1: currentNode = leaf;
2: while currentNode.Father ≠ null do
3:     if operation == NodeJoin then
4:         currentNode.Father.Size++;
5:     else if operation == NodeDeparture then
6:         currentNode.Father.Size--;
7:     end if
8: /* if Invariant 1 OR Invariant 2 do not hold */
9:     if currentNode.Father.Size ≤ minSize OR currentNode.Father.Size ≥ maxSize then
10:        currentNode = currentNode.Father;
11:     else
12:        return currentNode;
13:     end if
14: end while
15: return Root;

Algorithm 3 ComputeSizeInSubtree()

Require: node

1: if node is Leaf then
2:     node.Size = node.Bucket.Size;
3: else
4:     node.LeftChild.ComputeSizeInSubtree();
5:     node.RightChild.ComputeSizeInSubtree();
7:     node.minSize = (1 − εh) * node.Size;
8:     node.maxSize = (1 + εh) * node.Size;
9: end if

Afterwards, the mechanism traverses the path from leaf u to the root, in order to find the first node (if such a node exists) for which Invariant 4 is violated (Algorithm 4) and performs a redistribution in its subtree.

5.2.2. Node Redistribution

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 \([z/y]\) or \([z/y] + 1\) nodes. The redistribution cost is \(O(\log N)\), which is verified through experiments presented analytically in this work.
Algorithm 4 CheckNodeCriticality(BinaryNode leaf)
1: currentNode = leaf.Father;
2: while currentNode ≠ null do
3:     if currentNode.NodeCriticality IN [minValue, maxValue] then
4:         currentNode = currentNode.Father;
5:     else
6:         return currentNode;
7:     end if
8: end while
9: return null; /* redistribution isn’t necessary */

The redistribution in the subtree of v is carried out as follows (Algorithm 5). We assume that the subtree of v at height h has K buckets. A traversal of all buckets is performed in order to determine the exact value of |v| (number of nodes in the buckets of the subtree of v). Then, the first k buckets, will contain \( \lceil \frac{|v|}{2^h} \rceil + 1 \) nodes after redistribution, where \( k = |v| \mod 2^h \). The remaining \( K - k \) buckets will contain \( \lfloor \frac{|v|}{2^h} \rfloor \) nodes. The redistribution starts from the rightmost bucket b and it is performed in an in-order fashion so that elements in the nodes are not affected.

We assume that b has q extra nodes, which must be transferred to other buckets. Bucket b maintains a link dest to the next bucket b′ on the left, in which q extra nodes should be put. The q extra nodes are removed from b and are added to b′. The crucial step in this procedure is that during the transfer of nodes, internal nodes of PBT are also updated since the in-order traversal must remain untouched. More specifically, the representative z of b and its left in-order adjacent w are replaced by nodes of bucket b, and then, z and w along with the remaining q – 2 nodes of b are added to the tail of bucket b′. Afterwards, the horizontal and vertical links for the replaced binary nodes are updated for nodes that point to them. 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. In general, q nodes are removed from the tail of bucket b′, two of them replace nodes w and z and the remaining nodes are added to the head of b. The intriguing part comes when b′ contains less nodes than the q that b needs. In this case, b has to find the remaining nodes in the buckets on the left, so dest travels towards the leftmost bucket of the subtree, until \( q \leq \sum_{i=1}^{s} |b_i| \), where |bᵢ| is the size of the i-th bucket on the left. Then, nodes of bᵢ move to bᵢ−1, nodes from bᵢ−1 are transferred to bᵢ−2 and so on, until dest goes backwards to b′ and q nodes are moved from b′ into b.

5.2.3. Extension–Contraction

Throughout joins and departures of nodes, the size of buckets can increase undesirably or can decrease so much that some buckets may become empty. Either situations violate the \( D^3 \)-Tree principle for bucket size |b|:

\[
a_1 \log N \leq |b| \leq a_2 \log N
\]

(1)

where 0 < \( a_1 \) ≤ \( a_2 \).
Algorithm 5 RedistributeSubtree()

Require: BinaryNode node
1: node.ComputeSizeInSubtree();
2: newBucketSize = node.Size / $2^h$;
3: leftMostLeaf = node.LeftMostLeaf;
4: currentNode = node.RightMostLeaf;
5: while currentNode $\neq$ leftMostLeaf do
6:     currentNode = currentNode.RightMostLeaf;
7:     currentBucket = currentNode.Bucket;
8:     destinationBucket = currentNode.LeftAdjacents[0].Bucket;
9:     if currentBucket.Size $> \text{newBucketSize}$ then
10:        nodesToMove = currentBucket.Size − newBucketSize;
11:        for i=0; i < nodesToMove; i++ do
12:            Replace(currentNode, currentBucket.FirstNode)
13:            Replace(currentNode.LeftAdjacent, currentBucket.Representative);
14:            destinationBucket.InsertLast(currentNode.LeftAdjacent);
15:            UpdateAllLinks();
16:        end for
17:    else if currentBucket.Size $< \text{newBucketSize}$ then
18:        nodesToMove = newBucketSize − currentBucket.Size;
19:        tempDestNode = currentNode.LeftAdjacents[0];
20:        while tempDestNode $\neq$ leftMostLeaf AND availableNodes < nodesToMove do
21:            availableNodes += tempDestNode.Size;
22:            tempDestNode = tempDestNode.LeftAdjacents[0];
23:        end while
24:        tempNodesToMove = tempDestNode.Size − (totalNodesAvailable − nodesToMove);
25:        while tempDestNode $\neq$ currentNode do
26:            /* ... similar to previous case ...*/
27:            /* go back to the right, moving nodes */
28:        end while
29:    end if
30:    currentNode = currentNode.LeftAdjacents[0];
end while

The structure guarantees that (1) is always true, by employing two operations on the PBT, extension and contraction (Figure 1).

These operations are activated when a redistribution occurs at the root of the PBT and they add one level to the PBT when $|b| > a_2 \log N$ or delete one level from the PBT when $|b| < a_1 \log N$. 
The extension (Algorithm 6) is carried out as follows: the last level of the PBT is affected and a new level of leaves, as well as, a new set of buckets are created, using nodes from the old buckets. In particular, each leaf $u$ and its bucket $B$ with nodes $b_1, b_2, ..., b_s$ are replaced by a 2-level binary subtree $T_2$ preserving, at the same time, the in-order adjacency. Thus, the left leaf of $T_2$ is the old leaf $u$, the root of $T_2$ is the $i - \text{th}$ node of $b$, where $i = \left\lceil \frac{|b|}{2} \right\rceil$, the bucket $B_1$ of $u$ contains nodes $b_1, ..., b_{i-1}$, the right leaf of $T_2$ is the $(i + 1) - \text{th}$ node of $b$ and its bucket $B_2$ contains the remaining nodes of old bucket $B$, $b_{i+2}, ..., b_s$. During the process, all in-order adjacency links and father-child links are updated. After the extension has been carried out, leaves and nodes in height $h = 1$ reconstruct their routing tables and all binary nodes update the link to the rightmost leaf of their subtree (the leftmost leaf is not affected).

**Algorithm 6 PerformExtension()**

**Require:** BinaryNode Root

```
1: currentLeaf = Root.LeftMostLeaf;
2: while currentLeaf ≠ null do
3:     grandfather = currentLeaf.Father;
4:     oldBucket = currentLeaf.Bucket;
5:     i = \left\lceil \frac{\text{oldBucket.size}}{2} \right\rceil;
6:     B1 = oldBucket.RemoveNodes(i);
7:     newFather = oldBucket.RemoveFirstNode();
8:     rightLeaf = oldBucket.RemoveFirstNode();
9:     B2 = oldBucket.RemoveRemainingNodes();
10:    grandfather.ReplaceChild(currentLeaf, newFather);
11:    newFather.SetChildren(currentLeaf, rightLeaf);
12:    UpdateInOrderAdjacencyLinks();
13:    currentLeaf = currentLeaf.RightAdjacents[0];
14: end while
15: UpdateAllLinks();
16: Root.ComputeSizeInSubtree();
```

The contraction (Algorithm 7) is carried out as follows: the last level of the PBT is deleted and every pair of adjacent buckets is merged into one bucket. In particular, each 2-level subtree $T_2$ at height $h = 1$ and its buckets $B_1$ and $B_2$ are replaced by a leaf $u$ and a bucket $B$, preserving, at the same time, the in-order adjacency. The leaf $u$ is basically the left leaf of $T_2$ and the new bucket $B$ contains the nodes of $B_1$, the root of $T_2$, the right leaf of $T_2$ and finally the nodes of bucket $B_2$. During the process, all in-order adjacency links
and father-child links are updated. After the contraction has been carried out, the leaves reconstruct their routing tables and all binary nodes update their rightmost leaf.

**Algorithm 7** PerformContraction()

**Require:** BinaryNode Root

1. currentNode = Root.LeftMostLeaf.Father;
2. while currentNode ≠ null do
3.   currentLeaf = currentNode.LeftChild;
4.   newFather = currentNode.Father;
5.   newBucket = currentLeaf.Bucket;
6.   newBucket.Add(currentLeaf.Father);
7.   newBucket.Add(currentNode.RightChild.Bucket);
8.   newFather.ReplaceChild(currentNode, currentLeaf);
9.   UpdateInOrderAdjacencyLinks();
10.  currentNode = currentNode.RightAdjacents[0];
11. end while
12.  UpdateAllLinks();
13.  Root.ComputeSizeInSubtree();

It is obvious that these two operations are quite costly since they involve a reconstruction of the overlay, but this reconstruction rarely happens.

**5.3. 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. In case \( v \) is a node of the PBT (Algorithm 8), let \( z \) be the node with range of values containing \( a \), \( a \in [x_z, x_z'] \) and assume w.l.o.g. that \( x'_z < a \). The case where \( x_z > a \) 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 \( x'_z < a \) and \( x_w > a \), unless \( a \) is in the range of \( u \) or in the range of any visited node of \( l \) and the search terminates.

More specifically, we keep travelling to the right using the rightmost links of the routing tables of nodes in level \( l \) (the most distant ones), until we find a node \( q \) such that \( x_q > a \), or until we have reached the rightmost node \( qr \) of level \( l \). If the first case is true, then \( a \) is somewhere between \( q \) and the last visited node in the left of \( q \), so we start travelling to the left decreasing our travelling step by 1. We continue travelling left and right, gradually decreasing the travelling step, until we find the siblings \( u \) and \( w \) mentioned above in step = 0. If the second case is true, then \( x'_qr < a \) and according to the in-order traversal the search is confined to the right subtree of \( qr \).
Algorithm 8 Search(integer element)

Require: BinaryNode currentNode, step ≠ 0
1: BinaryNode leftMostNode, rightMostNode;
2: /* horizontal search */
3: while step ≠ 0 do
4:   if currentNode.Contains(element) then
5:     return currentNode;
6:   end if
7:   if currentNode.UpperValue ≤ element then
8:     if currentNode == RightMostNodeInLevel then
9:       return currentNode.SearchSubtree(element);
10:   end if
11:   leftMostNode = currentNode;
12:   if rightMostNode ≠ null then
13:     step --;
14:     currentNode = currentNode.RightAdjacents[step];
15:   else
16:     currentNode = currentNode.RightAdjacents.Last;
18:   end if
19: else if currentNode.LowerValue > element then
20:   /*... similar to previous case ...*/
21: end if
22: end while
23: /* step == 0, vertical search */
24: BinaryNode leftSibling; rightSibling;
25: if currentNode.UpperValue ≤ element then
26:   leftSibling = currentNode;
27:   rightSibling = currentNode.RightAdjacents[0];
28: else if currentNode.LowerValue > element then
29:   leftSibling = currentNode.LeftAdjacents[0];
30:   rightSibling = currentNode;
31: end if
32: targetNode = leftSibling.SearchSubtree(element);
33: if targetNode == null then
34:   targetNode = leftSibling.SearchAncestor(element);
35: end if
36: if targetNode == null then
37:   targetNode = rightSibling.SearchSubtree(element);
38: end if
39: return targetNode;
Having located nodes \( u \) and \( w \), the horizontal search procedure is terminated and the vertical search is initiated. Node \( z \) will either be the common ancestor of \( u \) and \( w \) (Algorithm 9), or in the right subtree rooted at \( u \) (Algorithm 10), or in the left subtree rooted at \( w \). If \( u \) is a binary node, it 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 \( w \). If \( z \) belongs to the subtree of \( w \), a symmetric search is performed.

**Algorithm 9**

```
Search(SearchAncestor(integer element))
```

**Require:** BinaryNode currentNode

```
1: if currentNode.UpperValue ≤ element then
2:   if currentNode is Leaf then
3:     ancestor = currentNode.RightInOrderAdjacent;
4:   else
5:     ancestor = currentNode.RightMostLeaf.RightInOrderAdjacent;
6:   end if
7: else if currentNode.LowerValue > element then
8:   /*... similar to previous case ...*/
9: end if
10: if ancestor.Contains(element) then
11:   return ancestor;
12: end if
13: return null;
```

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 and is verified through experiments presented in this work.

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.

### 5.4. Element Insertions and Deletions

#### 5.4.1. Handling Element Updates

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 \( u \). Element \( a \) is inserted in \( u \) or is deleted from \( u \), depending on the request.

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. However, node \( u \) can be any node in the structure, even an internal node of PBT. Therefore, if \( u \) is a bucket node or a leaf, \( a \) is inserted to \( u \), and no further action is necessary.

If \( u \) is an internal node of the PBT, element \( a \) is inserted in \( u \). Then the first element of \( u \) (note that elements into nodes are sorted) is removed from \( u \) and inserted to \( q \), the last node of the bucket of the left adjacent of \( u \), 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, as long as Invariants 1 or 2 do not hold. When the first node \( w \) is accessed for which Invariants 1 and 2 hold, then the nodes in the subtree of its child \( q \) in the path, have their weights
recomputed. Afterwards, the mechanism traverses the path from leaf \( u \) to the root, in order to find the first node (if such a node exists) for which Invariant 3 is violated and performs a load-balancing in its subtree.

**Algorithm 10 Search(SearchSubtree(integer element))**

```plaintext
Require: BinaryNode currentNode
1: if currentNode.Contains(element) then
2:    return currentNode;
3: end if
4: if currentNode is Leaf then
5:    return currentNode.Bucket.Search(element);
6: end if
7: if currentNode.UpperValue \leq element then
8:    rightLeaf = currentNode.RightMostLeaf;
9:    if rightLeaf.Contains(element) then
10:       return rightLeaf;
11: end if
12: if rightLeaf.UpperValue \leq element then
13:    return rightLeaf.Bucket.Search(element);
14: else
15:    /* search internal subtree */
16:    currentNode = currentNode.RightChild;
17:    while currentNode \neq null do
18:       if currentNode.Contains(element) then
19:          return currentNode;
20:      end if
21:      if currentNode.Node.UpperValue \leq element then
22:         currentNode = currentNode.RightChild;
23:      else if currentNode.Node.LowerValue > element then
24:         currentNode = currentNode.LeftChild;
25:      end if
26:      end while
27:      return currentNode.Bucket.Search(element);
28: end if
29: else if currentNode.LowerValue > element then
30:    /*... similar to previous case ...*/
31: end if
32: return null;
```

5.4.2 Load Balancing

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 \( \floor{\frac{w(v)}{|v|}} \) or \( \floor{\frac{w(v)}{|v|}} + 1 \) elements. The load-balancing cost is \( O(\log N) \), which is verified through experiments.
The load-balancing in the subtree of $v$ is carried out as follows (Algorithm 11). We assume that $v$ has $|v|$ nodes in its subtree. A bottom-up computation of the weights in all nodes of the subtree is performed, in order to determine the weight $w(v)$ of $v$. Then, the first $k$ nodes will contain $\left\lfloor \frac{w(v)}{|v|} \right\rfloor + 1$ elements after load-balancing, where $k = w(v) \mod |v|$. The remaining $|v| - k$ nodes will contain $\left\lfloor \frac{w(v)}{|v|} \right\rfloor$ elements. The redistribution starts from the rightmost node $w$ of the rightmost bucket $b$ and it is performed in an in-order fashion.

Algorithm 11: Search(LoadBalanceSubtree())

Require: BinaryNode node
1: node.ComputeSizeInSubtree();
2: node.ComputeWeightInSubtree();
3: newNodeDensity = node.Weight / node.Size;
4: leftMostLeaf = node.LeftMostLeaf;
5: currentNode = node.RightMostLeaf;
6: while currentNode $\neq$ leftMostLeaf do
7:   destinationNode = node.FindDestinationNode();
8:   if currentNode.Elements $>$ newNodeDensity then
9:     elementsToMove = currentNode.Elements $-$ newNodeDensity;
10:    SplitElements(currentNode, destinationNode, elementsToMove);
11:   else if currentNode.Elements $<$ newNodeDensity then
12:     elementsToMove = newNodeDensity $-$ currentNode.Elements;
13:    tempDestNode = destinationNode;
14:    while tempDestNode $\neq$ leftMostLeaf AND availableElements $<$ elementsToMove do
15:      availableElements += tempDestNode.Elements;
16:      tempDestNode = tempDestNode.FindDestinationNode();
17:    end while
18:    tempElementsToMove = tempDestNode.Elements $-$ (totalElementsAvailable $-$ elementsToMove);
19:    tempSourceNode = tempDestNode.FindSourceNode();
20:   while tempDestNode $\neq$ currentNode do
21:     SplitElements(tempDestNode, tempSourceNode, tempElementsToMove);
22:     tempDestNode = tempSourceNode;
23:     tempElementsToMove = tempDestNode.Elements;
24:     tempSourceNode = tempSourceNode.FindSourceNode();
25:   end while
26: end if
27: currentNode = destinationNode;
28: end while
We assume that \( w \) has \( m \) extra elements which must be transferred to other nodes. Hence, node \( w \) has a link to node \( w' \) in which the \( m \) additional elements should be inserted. To locate node \( w' \), we take into consideration the following cases: (i) if \( w \) is a bucket node, \( w' \) is its left node, unless \( w \) is the first node of the bucket and then \( w' \) is the bucket representative, (ii) if \( w \) is a leaf, then \( w' \) is the left in-order adjacent of \( w \) and (iii) if \( w \) is an internal binary node, then its left in-order adjacent is a leaf, and \( w' \) is the last node of its bucket. Having located \( w' \), the first \( m \) extra elements of \( w \) are removed from \( w \) and are added to the end of the element queue of \( w' \), to preserve the indexing structure of the tree.

Then, the ranges of \( w \) and \( w' \) are updated, respectively.

The case where \( m \) elements must be transferred to node \( w \) from node \( w' \) is completely symmetric. In general, the last \( m \) elements of \( w' \) are removed from \( w' \) and are inserted in the first \( m \) places in the element queue of \( w \). The intriguing part is when \( w' \) contains less elements than the \( m \) elements that \( w \) needs. In this case, \( dest \) travels towards the leftmost node of the subtree, following the in-order traversal, until \( m \leq \sum_{i=1} e(u_i) \), where \( e(u_i) \) is the number of elements of the \( i \)-th node on the left. Then, elements of node \( u_1 \) are transferred to \( u_{s-1} \), elements from \( u_{s-1} \) are transferred to \( u_{s-2} \) and so on, until \( dest \) goes backwards to \( w' \) and \( m \) elements are moved from \( w' \) into \( w \).

5.5. Fault Tolerance

Searches and updates in the \( D^3 \)-Tree do not tend to favour any node and in particular nodes near the root, which are therefore not crucial, and their failure will not cause more problems than the failure of any node. However, a single node can be easily disconnected from the overlay, when all nodes with which it is connected fail. This means that four failures (two adjacent nodes and two children) are enough to disconnect the root. The most easily disconnected nodes 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 \( v \) 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 \( v \). More specifically, we assume that node \( w \) discovered that \( v \) is unreachable during some operation.

Considering all possible relative positions between \( w \) and \( v \), we have the following cases, in which we want to locate the right child \( q \) of \( v \) that will lead to \( u \). First, if \( w \) and \( v \) are on the same level, by Lemma 1(i) we locate \( q \) and thus the right adjacent node \( u \) of \( v \) is the leftmost leaf in the subtree of \( q \). To be more clear, if \( w \) is connected to \( v \) by the \( i \)-th link of its routing table, then its right child is connected to \( q \) by the \((i+1)\)-th link of its routing table.

Second, if \( w \) is the father of \( v \), by Lemma 1(i) its left (right) child \( p \) has a link to the missing node \( v \) and the right child of \( p \) has a link to \( q \), so \( u \) is located. Third, if \( w \) is the left (right) child of \( v \), then \( u \) is easily located.

If \( v \) is a leaf, it should be replaced by the first node \( u \) in its bucket. However, in \( D^3 \)-Tree predecessor, 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 \( D^3 \)-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 \( v \).

In any case, the elements stored in \( v \) are lost. Moreover, the navigation data of \( u \) (left adjacent of \( v \)) are copied to the first node \( z \) in its bucket, which takes its place, and \( u \) has its routing tables recomputed.
5.6. Single Queries with Node Failures

The problem of searching an element in a network where several nodes have fallen introduces some very intriguing aspects. It can be considered two-dimensional since the search must be successful and cost-effective. A successful search for element $a$ refers to locating the target node $z$ for which $a \in [x_r, x_r']$.

An unsuccessful search refers to the cases where (i) $z$ is unreachable and (ii) there is a path to $z$, but the search algorithm couldn’t follow it to locate $z$, due to failures of intermediate nodes. Unfortunately, the $D^3$-Tree predecessor doesn’t provide a search algorithm in case of node failures since it doesn’t sufficiently confront the structure’s fault tolerance. In the following, we present the key features of our search algorithm, mainly through examples, due to the complexity of its implementation.

The search procedure is similar to the simple search described in Section 5.3. 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.

If all right adjacents are unreachable, $v$ contacts its left adjacents, afterwards it tries to contact its children, its father and as a last chance, when all other nodes have failed, it contacts its left/right in-order adjacents and its left/rightmost leaf. Contacting children, in-order adjacents and leaves means changing the search level.

In case $x_r' < a$ 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 directly contacts its nearest left adjacent (at step = 0) and asks it to search to the left. This improvement reduces the number of messages meant to fail because of the exponential positions of nodes in routing tables and the nature of binary horizontal search. For example, in Figure 2, search starts from $v_0$ and $v_8$ contacts $v_7$ since $v_4$ has fallen. No node contacts $v_4$ from then on, and the number of messages is reduced by 2.

![Figure 2. Example of binary horizontal search with node failures.](image)

A vertical search to locate $z$ is always initiated between two siblings $u$ and $w$, which are both active, or one of them is unreachable, as shown in Figure 3 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. Next, we contact the common ancestor, and then, if the other sibling is unreachable, the active sibling tries to reach 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.
We assume (w.l.o.g.) that the left sibling $u$ is active and $w$, the right one, is unreachable, as shown in Figure 3. First $u$ contacts its rightmost leaf $y$ of its subtree. If $y$ is reachable and $x_y > a$ or if $y$ is unreachable, then an ordinary top-down search from node $u$ is initiated to find $z$. If a node in the searching path is unreachable, a mechanism described below is activated to contact its children. Otherwise, if $x_y' < a$, 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 left subtree of $w$.

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 (right) child. If its child is unreachable, (Figure 3), 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 the initial node $v$ is a bucket node, if its range contains $a$ the search terminates, otherwise the search is forwarded to the bucket representative. If the bucket representative has fallen, 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 $O(\log N)$.

**Proof.** Let $O(N)$ be the number of failed nodes in a network of $N$ nodes and $n$ elements. In general $n \gg N$, and we may assume w.l.o.g. that $n = \beta N$, for some (large) constant $\beta \geq 1$ expressing the space capacity of each node (i.e., how many data elements can be stored in it).

In the worst case, we want to execute $O(n)$ search operations. If a node $v$ cannot be contacted during the search procedure, the network initiates a node withdrawal procedure to remove $v$. In case of massive failures, node withdrawals will not be negligible; hence the network will often activate the redistribution mechanism to avoid the structure’s imbalance. The redistribution cost is bounded by the amortized bound $O(\log N)$.

In the worst case, the maximum number of redistributions will be $O(N)$ (equal to the number of failed nodes), so the whole search process will require $T(N)$ hops, where

$$T(N) = O(N) \cdot O(\log N)$$

The amortized cost $\tilde{T}(N)$ will then be

$$\tilde{T}(N) = \frac{T(N)}{O(n)} = \frac{O(N) \cdot O(\log N)}{O(n)} = c \cdot O(\log N) = O(\log N)$$
6. The ART\(^+\) Structure

In this section, we briefly describe and present the theoretical background of ART\(^+\). ART\(^+\) is similar to its predecessor, ART \cite{17} regarding the structure’s outer level. Their difference, which introduces performance enhancements, lies in the fact that each cluster node of ART\(^+\) is structured as a D\(^3\)-Tree \cite{18}, instead of the BATON\(^*\) \cite{10} structure, used as the inner level of ART.

6.1. Building the ART\(^+\) Structure

The backbone structure of ART\(^+\) is similar to LRT (LRT: Level Range Tree), in which some interventions have been made to improve its performance and increase the robustness of the whole system. ART\(^+\) is built by grouping cluster nodes with the same ancestor and recursively organizing them in a tree structure. A cluster node is defined as a bucket of ordered nodes. The innermost level of nesting (recursion) will be characterized by having a tree in which no more than \(b\) cluster nodes share the same direct ancestor, where \(b\) is a double-exponentially power of two (e.g., 2, 4, 16, ...). Thus, multiple independent trees are imposed on the collection of cluster nodes. The height of ART\(^+\) is \(O(\log \log N)\) in the worst case. The ART\(^+\) structure remains WHP-number unchanged. Figure 4 illustrates a simple example, where \(b = 2\).

![Figure 4. The ART\(^+\) structure for \(b = 2\).](image-url)

The degree of the cluster nodes at level \(i > 0\) is \(d(i) = t(i)\), where \(t(i)\) indicates the number of cluster nodes at level \(i\). It holds that \(d(0) = b\) and \(t(0) = 1\). At initialization step, the 1st node, the \((\ln n + 1) - th\) node, the \((2 \cdot \ln n + 1) - th\) node and so on are chosen as bucket representatives, according to the balls in bins combinatorial game presented in \cite{20}.

Let \(n\) be \(w\)-bit keys, \(N\) be the total number of nodes and \(N'\) be the total number of cluster nodes. Each node with label \(i\) (where \(1 \leq i \leq N\)) of a random cluster, stores ordered keys that belong in the range \([(i - 1) \ln n, i \ln n - 1]\), where \(N = n / \ln n\). Each cluster node with label \(i'\) (where \(1 \leq i' \leq N'\)) stores ordered nodes with sorted keys belonging in the range \([(i' - 1) \ln^2 n, i' \ln^2 n - 1]\), where \(N' = n / \ln^2 n\) or \(N' = N / \ln n\) is the number of cluster nodes.
ART+ stores cluster nodes only, each of which is structured as an independent decentralized architecture, which changes dynamically after node join/leave and element insert/delete operations inside it. In contrast to its predecessor, ART, whose inner level was structured as a BATON†, each cluster node of ART+ is structured as a $D^3$-Tree.

Each cluster node is equipped with a routing table named Random Spine Index (RSI), which stores pointers (red arrows in Figure 4) to cluster nodes belonging to a random spine of the tree (instead of the LSI (LSI: Left Spine Index) of LRT, which stores pointers to the nodes of the left-most spine). Moreover, instead of using fat CI (CI: Collection Index) tables, which store pointers to the collections of nodes presented at the same level, the appropriate collection of cluster nodes is accessed by using a 2-level LRT structure. In ART+, the overlay of cluster nodes remains unaffected in the expected WHP-case when nodes join or leave the network.

6.2. Load Balancing

The operation of join/leave of nodes inside a cluster node is modelled as the combinatorial game of bins and balls presented in [20]. In this way, for an $\mu(\cdot)$ random sequence of join/leave node operations, the load of each cluster node never exceeds $\Theta(\log N)$ size and never becomes zero in the expected WHP-case. In skew sequences, though, the load of each cluster node may become $\Theta(N)$ in the worst case.

The load-balancing mechanism for a $D^3$-Tree structure, as described previously, has an amortized cost of $O(\log K)$, where $K$ is the total number of nodes in the $D^3$-Tree. Thus, in an ART+ structure, the cost of load-balancing is $O(\log \log N)$ amortized.

6.3. Routing Overhead

We overcome the problem of fat CI tables with routing overhead of $O(\sqrt{N})$ in the worst case, using a 2-level LRT structure. The 2-level LRT is an LRT structure over $\log 2^c Z$ buckets each of which organizes $Z / \log 2^c Z$ collections in an LRT manner, where $Z$ is the number of collections at the current level and $c$ is a big positive constant. As a consequence, the routing information overhead becomes $O(N^{1/4} / \log^c N)$ in the worst case.

6.4. Search Algorithms

Since the structure’s maximum number of nesting levels is $O(\log_b \log N)$ and at each nesting level $i$ we have to apply the standard LRT structure in $N^{1/2^i}$ collections, the whole searching process requires $O(\log^2 \log N)$ hops. Then, we have to locate the target node by searching the respective decentralized structure. Through the polylogarithmic load of each cluster node, the total query complexity $O(\log^2 \log N)$ follows. Exploiting now the order of keys on each node, range queries require $O(\log^2 \log N + |A|)$ hops, where $|A|$ is the answer size.

6.5. Join/Leave Operations

A node $u$ can make a join/leave request to a node $v$, which is located at cluster node $W$. Since the size of $W$ is bounded by a polylog $N$ size in the expected WHP-case, the node join/leave can be carried out in $O(\log \log N)$ hops.

The outer structure of ART+ remains unchanged WHP, as mentioned before, but each $D^3$-Tree structure changes dynamically after node join/leave operations. According to $D^3$-Tree performance evaluation, the node join/leave can be carried out in $O(\log \log N)$ hops.

6.6. Node Failures and Network Restructuring

In the ART+ structure, similarly to ART, the overlay of cluster nodes remains unchanged in the expected case WHP, so in each cluster node, the algorithms for node failure and network restructuring are according to the decentralized architecture used. Finally, $D^3$-Tree is a highly fault-tolerant structure since it supports procedures for node withdrawal and handles massive node failures efficiently.
7. Experimental Study

We built a simulator (Our simulator is a standalone desktop application, developed in Visual Studio 2010, available in https://github.com/sourlaef/d3-tree-sim (accessed on 5 March 2022)) with a user-friendly interface and a graphical representation of the structure, to evaluate the performance of $D^3$-Tree.

In Figure 5 we present the user interface of the $D^3$-Tree simulator. At the top left, the user can construct a new tree structure after setting the tree parameters. At the top right, the user can conduct experiments regarding node joins/departures, element insertions/deletions, single and range queries, setting the parameters for each experiment. In the center of the screen, helpful information regarding the tree construction and experiments is displayed. Finally, a graphical representation of the structure is shown at the bottom.

Figure 5. The $D^3$-Tree simulator user interface.

7.1. Performance Evaluation for $D^3$-Tree

To evaluate the cost of operations, we ran experiments with a different number of nodes $N$ from 1000 to 10,000, 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, $1000 \times N$ elements were inserted. We used the number of passing messages to measure the system’s performance.


To measure the network performance for node updates, we conducted experiments for three different value ranges of node criticality: $[0.25, 0.75]$, $[0.35, 0.65]$, $[0.45, 0.55]$. For 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 to negligible node update costs. Hence, we decided to perform only one type of update, $2N$ joins, that are expected to cause several redistributions.

Figure 6a shows the average amortized redistribution cost, while Figure 6b depicts the same metrics in the worst case, where the same node (the leftmost leaf) is the host node for all node joins. Note that we have taken into account only the amortized cost of node joins causing redistributions since otherwise, the amortized cost is negligible. Figure 6c shows the redistribution rate for $D^3$-Tree in average and worst case.

Through experiments, we observed that even in 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.

We also observed that in the average case, during node joins, the Invariant 4 is rarely violated to invoke a redistribution operation (Figure 6c). This also depends on the range
in which node criticality belongs. When the range is narrowed, more redistributions take place during node joins, but the amortized cost is low since the majority of redistributions occur in subtrees of low height, as shown in Figure 6d. This is more obvious in the worst case. When we use wide ranges, more node joins take place before a redistribution occurs, making the redistribution operation more costly since many nodes have been cumulated into the bucket of the leftmost leaf.

Figure 6d shows in detail the allocation of redistribution height for different node criticality ranges in a network of 10,000 initial nodes. We observe that in the worst case, the number of redistributions is more than twice the number of redistributions of the average case.

7.1.2. Cost of Element Insertions/Deletions

To measure the network performance for the operation of element updates, we conducted experiments for three different value ranges of criticality: [0.90, 1.10], [0.67, 1.50], [0.53, 1.90] formed from values of constant $c = \{1.1, 1.5, 1.9\}$ correspondingly. For 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 to negligible node update costs. Hence, we decided to perform only one type of update, $n$ insertions.

Figure 7a shows the average amortized load-balancing cost, while Figure 7b shows the load-balancing rate. Both average cases and worst cases are depicted in the same graph. The average cases for $c$ values of 1.5 and 1.9 led to negligible amortized cost, so they were disregarded. The same node (the leftmost leaf) is the host node for all element insertions in the worst case. Note that we have only considered the amortized cost of element insertions causing load-balancing operations since otherwise, the amortized cost is negligible.

![Figure 6](image-url)

Figure 6. Node update operations. (a) average case, (b) worst case, (c) redistribution rate of winner $D^3$-Tree, (d) redistribution height of winner $D^3$-Tree.

![Figure 7](image-url)

Figure 7. Cost of Element Insertions/Deletions.
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 $\text{fanout} = 10$ and P-Ring.

Moreover, we observed that in the average case, during element insertions, the Invariant 3 is rarely violated to invoke the load-balancing mechanism (Figure 7b). This also depends on the value range of criticality. When the range is narrowed, more load-balancing operations occur during element insertions, but the amortized cost is low since the subtree isn’t very imbalanced. However, the majority of load-balancing operations occur in subtrees of high height, as shown in Figure 7c. On the other hand, when we use wide ranges, more element insertions occur before the load-balancing mechanism is activated, leading to more frequent and costly load-balancing operations.

![Figure 7. Element update operations. (a) average messages, (b) load-balancing rate of winner $D^3$-Tree, (c) load-balancing height of winner $D^3$-Tree.](image)

7.1.3. 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 Figure 8a. 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 than the cost of BATON, BATON$^*$, and P-Ring. This is because 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 four groups, each of $M/2$ searches. To better estimate the search cost, we forced a different set of nodes to fail in each group.

Figure 8b depicts the increase in search cost when massive failures of nodes 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 $\geq 30\%$. 

Details about the behaviour of the enhanced search mechanism of $D^3$-Tree in case of node failures are depicted in Figure 8c,d, which show the average number of messages required and the success percentage, respectively.

Experimenting, we observed that when the node failure percentage is small (10% to 15%), the majority of single queries that fail are those 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 that the proposed structure is highly fault-tolerant.

7.2. Performance Evaluation for ART$^+$

We also evaluated the performance of ART$^+$ structure and compared it to its predecessor, ART [17]. Each cluster node of the ART and ART$^+$ is a BATON$^*$ and $D^3$-Tree structure respectively. BATON$^*$ was implemented and evaluated in [10], while ART was evaluated in [17], using the Distributed Java D-P2P-Sim simulator presented in [21]. The source code of the whole evaluation process, which showcases the improved performance, scalability, and robustness of ART over BATON$^*$ is publicly available (http://code.google.com/p/d-p2p-sim/) (accessed on 5 December 2021). For the performance evaluation of ART$^+$, we used the $D^3$-Tree simulator.

To evaluate the performance of ART and ART$^+$ for the search and load-balancing operations, we ran experiments with a different number of total nodes $N$ from 50,000 to 500,000. As proved in [17], each cluster node stores no more than $0.75 \log_2 N$ nodes in smooth distributions (normal, beta, uniform) and no more than $2.5 \log^2 N$ nodes in non-smooth distributions (powlow, zipfian, weibull). Moreover, we inserted elements equal to the network size multiplied by 2000, numbers from the universe $[1 \ldots 1,000,000,000]$. Finally, we used the number of passing messages to measure the performance.

Note here that, as proved in [17], ART outperforms BATON$^*$ in search operations, except for the case where $b = 2$. Moreover, ART achieves better load-balancing compared to BATON$^*$, since the cluster node overlay remains unaffected WHP through joins/de-
portures of nodes, and the load-balancing performance is restricted inside a cluster node. Consequently, in this work, ART$^+$ is compared directly to ART.

7.2.1. Cost of Search Operations

To measure the network performance for the search operations (single and range queries), we conducted experiments for different values of $b$, 2, 4 and 16. For each $N$, we executed 1000 single queries and 1000 range queries. The search cost is depicted in Figure 9. Both normal (beta, uniform) and worst cases (powlow, zipfian, weibull) are depicted in the same graph.

Experiments confirm that the query performance of ART and ART$^+$ is $O(\log_2 b \log N)$ and the slight performance divergences are because BATON$^*$, as the inner structure of ART’s cluster node, performs better than $D^3$-Tree in search operations. In case of massive failures, the search algorithm finds alternative paths to overcome the unreachable nodes. Thus, an increase in node failures increases search costs. To evaluate the system in case of massive failures, we initialized the system with 10,000 nodes and let them randomly fail without recovering. At each step, we check if the network is partitioned or not. Since the backbone of ART and ART$^+$ remains unaffected WHP, the search cost is restricted inside a cluster node (BATON$^*$ or $D^3$-Tree respectively), meaning that $b$ parameter does not affect the overall expected cost. Figure 9d illustrates the effect of massive failures.

We observe that both structures are fault-tolerant since the failure percentage has to reach the threshold of 60% to partition them. Moreover, even in the worst-case scenario, the ART$^+$ maintains a lower search cost compared to ART, since $D^3$-Tree handles node failures more effectively than BATON$^*$.

![Figure 9. Cost of search operations. (a) Case $b = 2$, (b) case $b = 4$, (c) case $b = 16$, (d) massive failures.](image)

7.2.2. Cost of Load-Balancing Operations

To evaluate the cost of load-balancing, we tested the network with a variety of distributions. For a network of $N$ total nodes, we performed $2N$ node updates. Both ART and ART$^+$ remain unaffected WHP, when nodes join or leave the network, thus the load-balancing
performance is restricted inside a cluster node (BATON∗ or $D^3$-Tree respectively), meaning that the $b$ parameter does not affect the overall expected cost. The load-balancing cost is depicted in Figure 10. Both normal and worst cases are depicted in the same graph.

Experiments confirm that ART+ has an $O(\log \log N)$ load-balancing performance, instead of the ART performance of $O(m \cdot \log_m \log N)$. Thus, even in the worst case scenario, the ART+ outperforms ART, since $D^3$-Tree has a more efficient load-balancing mechanism than BATON∗ (Figure 10).

![Figure 10. Cost of load-balancing operation.](image)

8. Conclusions

In this work, we presented a dynamic distributed deterministic structure called $D^3$-Tree. Our proposed structure introduces many enhancements over the solutions in literature and its predecessor regarding load-balancing and fault tolerance.

Verifying the theory, we have proved through experiments that $D^3$-Tree outperforms other well-known tree-based structures by achieving an $O(\log N)$ amortized bound in the most costly operation of load-balancing, even in a worst-case scenario.

Moreover, investigating the structure’s fault tolerance, both theoretically and through experiments, we proved that $D^3$-Tree is highly fault-tolerant, since, even for massive node failures of 30%, it achieves a significant success rate of 85% in element search, without increasing the cost considerably.

Afterwards, we went one step further to achieve sub-logarithmic complexity and proposed the ART+ structure, exploiting the excellent performance of $D^3$-Tree.

We proved that the communication cost of query operations, element updates, and node join/leave operations of ART+ scale sub-logarithmically as expected WHP. Moreover, the cost for the load-balancing operation is sub-logarithmic amortized. Experimental comparison to its predecessor, ART, showed slightly less efficiency towards search operations (single and range queries), but improved performance for the load-balancing operation and the search operations in case of node failures. Moreover, experiments confirmed that ART+ is highly fault-tolerant in case of massive failures. Note that, so far, ART outperforms the state-of-the-art decentralized structures. In the near future, we would like to incorporate new fuzzy randomized load balancing techniques [22], based on machine learning, as well as parallelism and concurrency [23] to increase the performance of all operations. Finally, it’s a big challenge to embed our novel techniques in real p2p architectures currently used in social networking, taking into account data privacy and anonymity issues [24]. In addition, we will incorporate algorithms and techniques that have recently been presented in the following decentralized systems [25–29].

Author Contributions: S.S., E.S., K.T., G.V. and C.Z. conceived of the idea, designed and performed the experiments, analyzed the results, drafted the initial manuscript and revised the final manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Competitiveness, Entrepreneurship and Innovation Operational Program (co-financed by EU and Greek national funds), under contract no. T2EDK-03472 (project iDeliver).
Acknowledgments: We are indebted to the anonymous reviewers whose comments helped us to improve the presentation.

Conflicts of Interest: The author has no conflicts of interest.

References


