Peer-to-Peer Netw. Appl. (2017) 10:954–982
DOI 10.1007/s12083-016-0455-2

HyCube: A distributed hash table based on a variable metric

Artur Olszak[1]

Received: 27 August 2015 / Accepted: 16 March 2016 / Published online: 22 April 2016

Abstract Distributed Hash Tables (DHT) proved to be scalable decentralized systems providing efficient resource location. This paper concentrates on efficiency and resilience to node failures of DHT systems and presents a novel model of a distributed hash table based on a hierarchical hypercube geometry, called *HyCube*. The DHT geometry, the choice of the metric defining logical distances between nodes, and the routing algorithm have fundamental influence on routing efficiency and resilience. The use of the one-dimensional model (placing the nodes logically on a ring) allows the nodes to maintain sets of references called sequential neighbors - certain numbers of neighbors that are the closest existing nodes in both directions on the ring. Such a model yields a very high level of resilience to node failures. The new approach, presented in the paper, employs a variable multi-dimensional metric adopting the Steinhaus transform. Routing, lookup and search algorithms are discussed, as well as routing table nodes selection and self-organization techniques. It is shown that the new approach allows reaching a higher level of resilience to node failures, as well as a shorter average routing path length than with the use of the sequential neighbors sets.

Keywords Distributed hash table · Routing algorithm · Lookup algorithm · Search algorithm · Failure resilience

✉ Artur Olszak
A.Olszak@ii.pw.edu.pl

[1] Institute of Computer Science, Warsaw University of Technology, Warsaw, Poland

1 Introduction

In recent years, we observe a growing interest in large-scale distributed systems based on the distributed hash table algorithm (DHT). DHT is a distributed system storing large amount of information in a way allowing efficient lookup for the information being stored. DHT systems are used to store key-value pairs, similarly to a hash table. However, the key-value pairs set is stored in a distributed manner and is shared among the nodes of the distributed hash table (computers connected to the DHT network). Distributed hash tables provide efficient resource location, are extremely scalable and have a lot of possible applications - storing resources in a distributed way, distributed file systems [8], [14], communication, Internet telephony [32], live data streaming [1] and many other. DHT systems are also widely utilized for accomplishing efficient resource/service discovery for grid computing systems [26, 31, 34].

The fundamental functionality provided by distributed hash tables is efficient node/resource location - routing, lookup and search. In DHT systems, every node is assigned a unique identifier, used to locate the node in the virtual ID space, and resources are usually stored by nodes which are determined based on resource keys. The resource key is typically a result of a hash function on the resource. Because of the dynamic nature of DHT systems and the stored key set changing continuously, most DHTs use some variant of consistent hashing[1] to map keys to nodes. Most often, the node ID space is isomorphic with the resource

[1]Consistent hashing is a hashing technique such that removing or adding one node (a hash table slot) does not significantly influence the key distribution among other nodes. Usually changes are required only in the closest neighborhood. In traditional approaches, usually adding or removing a hash table slot would cause the hashing algorithms to remap all the keys.

key space, and nodes responsible for resources (nodes in which the resources are stored) are those whose identifiers are the closest to the resource keys (among all nodes in the DHT), according to a DHT-specific metric defining logical distances between nodes. Thus, the resource lookup comes down to lookup for the node(s) closest to the resource key. DHT systems provide efficient node lookup and routing messages between nodes. Even in very large systems, every DHT node should be able to route messages to any other node (recursive routing), or find the node responsible for any resource (lookup - iterative routing) in a small number of steps.

To support the lookup/routing, every node in a distributed hash table maintains a list of references to other nodes in the system - a routing table. The routing table is built in a way that allows locating a node that is closer (than the current node) to any arbitrarily chosen node in the DHT, which consequently, allows decreasing the distance left to the destination node in each routing/lookup step. Although being founded on a common concept, individual distributed hash table designs are based on different overlay network geometries: tree [27, 30, 36], hypercube [28], ring [33], XOR metric [22]. The overlay network is a virtual network formed by the connections between DHT nodes, and the geometry determines the connection graph structure - constraints defining between which pairs of nodes connections may be established. Individual geometries, routing and lookup algorithms, as well as the chosen metric defining logical distances between nodes (and resource keys), yield different resulting system properties: efficiency (routing path length/number of lookup steps), the level of flexibility in the next hop selection (to how many nodes in the routing table a message may be routed in each routing/lookup step, preserving the routing convergence - decreasing the distance left), flexibility in the neighbor selection (choosing routing table nodes), resilience to node failures (ability of the system to deliver messages in the presence of node failures), and the ability of the system to recover good routing properties after node failures.

One of the most crucial characteristics of DHT systems is *static resilience* - the ability of the system to deliver messages in the presence of node failures, with the recovery mechanisms switched off. Static resilience is usually measured as a percentage of successful/failed routes and the average path length increase (the number of routing steps) in the presence of a given ratio of failed nodes - nodes that suddenly become unresponsive (do not respond to requests sent to them). A node failure might be caused by a node leaving the system without proper communicating that fact, not allowing its neighbors to reestablish connections, a network connection being broken, a node being overloaded and not processing messages, or due to existence of malicious nodes not processing requests according to the protocol. Static resilience characterizes the network tolerance to failures without the use of any maintenance/recovery mechanisms. However, it also directly translates to the behavior of the system under churn (the process of nodes joining and leaving the system - connecting to and disconnecting from the DHT [29]) - efficiency, ability to deliver messages, lookup accuracy, data availability and many other. It usually takes some time for the recovery and replication processes to react to topology changes, so maintaining a high level of static resilience is very important, especially in very dynamic systems, where the changes take place constantly. Moreover, static resilience also indirectly affects the efficiency of the system maintenance and recovery mechanisms. A different approach for analyzing resilience of DHT systems, presented in [15], discusses fault-tolerance in the worst-case joins and leaves scenario.

In [11], the authors discuss the influence of the geometry of distributed hash tables on their static resilience and average path length. Different geometries provide different degrees of flexibility in route selection (the number of nodes to which messages may be routed in each routing step). The authors prove that the flexibility in next hop selection is a crucial factor influencing the static resilience of the system. The flexibility determines how many other options remain for the next hop (preserving routing convergence to the destination), when the best next hop is down for any reason. If there are few of them (or none in some cases), the routing is very likely to fail very often under high failure rates.

Certain DHT architectures support the use of *sequential neighbors* sets - usually, sequential neighbors are some number of successors and the same number of predecessors of a node on a logical one-dimensional ring (which requires single global ordering of the nodes). Sequential neighbors, as an addition to the DHT-specific routing table nodes (providing effective routing/lookup), provide a very high level of flexibility in the next hop selection, as half of the sequential neighbors are able to route any message in either direction on the ring. That means that even under a high rate of node failures, every node is able to route any message, decreasing the distance left to the destination, unless all successors/predecessors fail. Some DHTs naturally support the existence of sequential neighbors [33], and other do not [22, 27, 28]. Due to many advantages of maintaining such sets, the DHT designs, not supporting this concept naturally, are often modified to include the support for sequential neighbors [30]. As presented in [11] and [37], introducing sequential neighbors sets dramatically increases the degree of flexibility in route (next hop) selection, making such systems highly resistant to node failures and targeted node attacks. These factors indirectly influence many other DHT characteristics, making them extremely important for large-scale DHT systems. Although such a one-dimensional model yields a high level of resilience to node failures, it may lead to path length increases in the presence of failures of many nodes. As the number of failed nodes increases, next hops are more often found only in the sequential neighbors sets, and at one extreme (routing using only sequential neighbors), the expected path length is proportional to the number of the nodes in the system. The use of a multidimensional routing metric may significantly decrease the expected path length when routing using only the closest neighbors sets. However, by using a multidimensional metric, the system loses the properties connected with the existence of sequential neighbors - ensuring that a certain number of closest nodes would match any potential direction.

This paper presents the architecture of a distributed hash table called *HyCube*. The presented model is based on a hierarchical hypercube geometry (a multiple-level nested hypercube, where vertices are lower-level hypercubes) and employs a variable (modified by nodes on the route) multidimensional metric adopting the Steinhaus transform [21]. Novel routing, lookup and search algorithms are discussed, as well as routing table nodes selection algorithms, and maintenance and recovery procedures. *HyCube* is scalable, efficient and significantly outperforms existing solutions in respect of resilience, routing and lookup efficiency under dynamically changing conditions, despite not using the sequential neighbors concept. A preliminary conception of the described DHT was presented at a conference and published in the conference proceedings [24]. This paper presents the complete design of *HyCube*, including numerous enhancements, supported by results obtained during extensive simulations.

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 introduces the routing architecture of *HyCube*. Section 4 describes the simulation methodology used for obtaining the experimental results presented in the paper. Sections 5 and 6 contain a study leading to improvement of the routing performance and resilience. The evaluation of the presented routing architecture is presented in Section 7. The routing table node selection mechanisms used in *HyCube* are discussed in Section 8. Section 9 describes algorithms used for locating nodes in the system - lookup and search. Self-organization and recovery algorithms are discussed in Section 10. Section 11 concludes.

2 Related work

There are currently several DHTs in use, and we observe an increasing number of applications of the DHT systems. This section briefly describes the most significant existing DHT architectures (in the descriptions below, N denotes the number of the nodes in the system).

Chord [33] is a DHT, in which nodes are logically located on a ring with 2^m possible positions (identifiers). Routing in *Chord* is based on passing messages to nodes being closer on the ring to the destination node than the current node. The routing tables are built in a way allowing decreasing the distance left to the destination at least by half in each routing step. The expected path length in *Chord* is $\log_2 N$. The ring topology naturally supports the sequential neighbors sets, mentioned in Section 1, which makes it very resistant to node failures.

CAN (*Content-Addressable Network*) [28] is an example of a distributed hash table built on a hypercube geometry. The *CAN* nodes are located in a d-dimensional hypercube (which in fact is treated as a d-dimensional torus - a ring in each dimension). The hypercube is partitioned into so-called zones (each node in the system has a zone assigned to it), and messages are routed between the zones - in every routing step, a message may be sent to a zone adjacent to the current zone. The average path length between any two nodes equals $\frac{d}{4} \cdot \sqrt[d]{N}$, where d is the number of dimensions. The routing flexibility is, however, limited in such a geometry, and the sequential neighbors sets cannot be used to increase the node failure resilience.

There have been several works regarding improvements of efficiency of *CAN* network. One of the approaches, described in [25] introduces shortcut zones and additional routing table nodes, corresponding to these zones. In [35], another approach is presented, where the hypercube is split into zones at multiple levels. The lowest-level zones correspond to the *CAN* zones, and zones at higher levels are so-called expressway zones - routing using these zones allows messages to be routed further than with the use of the regular *CAN* neighbors. In both solutions, introducing additional zones, decreased the average path length between nodes to $O(\log N)$.

Plaxton mesh [27] is a distributed hash table based on the tree geometry. In *Plaxton mesh*, identifiers of nodes consist of m d-bit groups. Every node maintains a routing table with multiple levels. For a given node A, the routing table level i represents node(s), which share first i initial groups of bits of the identifier with A, and the routing table slot at j-th position stores a node sharing i initial identifier groups with A and having the next group of bits equal to j. Thus, in every routing step, the routing algorithm is able to pass a message to a node, which shares a longer prefix with the destination node (in terms of bit groups) than the current node (if such a node exists). The expected path length equals $\log_{2^d} N$. As there is no flexibility in the next hop selection (in each routing step, the message may be routed to exactly one node in the routing table), the level of static resilience of the system in its pure form is very low.

Pastry [30] uses a routing algorithm based on the *Plaxton mesh*, but, in *Pastry*, the distance between pairs of nodes is defined as the distance on a ring. If a node sharing a longer prefix with the destination node is not found in a certain routing step, the message is routed to a node sharing the same prefix length as with the current node (in terms of the number of bit groups), but being closer to the destination in terms of the metric (ring). In addition to the routing table, every node maintains a set of closest nodes in either direction on the ring - the *leaf set*. A set built in this way is, in fact, the *sequential neighbors* set, which ensures that, with high probability, any message can be routed by any node, decreasing the distance left to the destination, even if the corresponding routing table slot is empty.

Tapestry DHT [36] is very similar to *Plaxton mesh* and *Pastry* in terms of the routing architecture. Main differences concern methods of ensuring locality, replication, as well as joining, leaving, and recovery algorithms.

Another type of a DHT geometry is the "Butterfly" geometry. An example of such a design is *Viceroy* [18]. In *Viceroy*, every node is assigned one of m levels, and a unique ID - a real number from 0 (inclusive) to 1 (exclusive). Each node at level l maintains two connections with nodes at level $l+1$. For a given node A, the first connection is made with the node with the closest ID to A at level $l+1$ (like on a ring, all arithmetic is done modulo 1), and the second connection is made with the node with the ID closest to $A + 1/2^l$. In addition, nodes at levels $l > 1$ maintain a connection to the closest node at level $l-1$. The nodes also connect to their successor and predecessor on their level-ring, as well as the successor and predecessor on the ring formed by all nodes at all levels. Routing proceeds in three phases. In the first phase, a message is sent up to the top level. Next, the routing proceeds to the bottom (using the first or the second link depending whether the destination is at a distance smaller or greater than $1/2^l$). When the lowest level (level m) is reached, the search is performed using the ring and level-ring links until the destination is reached. The expected number of steps is $O(\log N)$.

Kademlia [22] is a distributed hash table protocol which bases its search algorithm on the so-called XOR metric. The XOR metric defines the distance between any two nodes as a result of the bit XOR operation on their identifiers. Every node in the system maintains references to neighbors in structures called k-buckets. The number of k-buckets equals the number of identifier bits m, and i-th k-bucket stores at most k nodes, whose distances are between 2^i inclusive and 2^{i+1} exclusive. This means that nodes that can be included in the i-th bucket must have a differing i-th bit from the node's ID, and the first $i-1$ bits of the candidate ID must match those of the node's ID. With the use of the XOR metric and k-buckets, the lookup procedure quickly converges to the lookup key. The expected number of steps is $O(\log N)$. The resources in *Kademlia* are stored in k nodes closest to the resource key, so a node performing a lookup for a resource searches for k closest nodes to the resource key. *Kademlia* ensures its static resilience by the increased number of stored references to other nodes (multiple references in k-buckets), and sending lookup messages to multiple nodes in parallel. The *Kademlia* protocol gained a great popularity and is nowadays the most popular DHT used in variety of applications.

An interesting family of distributed hash tables are the systems based on degree-optimal graphs. The problem of degree-optimal graphs is formulated in [17]. *Koorde* [12] is a distributed hash table based on *Chord* and de Bruijn graphs [2]. The average path length between two nodes is $O(\log N)$ if nodes maintain routing tables containing only two nodes, and $O(\log N/\log\log N)$ for routing tables of size $O(\log N)$. Another DHT based on the de Bruijn graphs, *D2B* [10], for a d-dimensional de Bruijn graph achieves the average path length $O(\log_d N)$ and the expected number of references stored in the routing table $O(d)$. *Distance Halving* [23] is one more important DHT utilizing de Bruijn graphs. The graph structure is based on a dynamic decomposition of the space into cells and assigning the cells to nodes. The structure ensures the path length of $O(\log_d N)$, where d is the node degree (constant). The path lengths in these systems are therefore asymptotically even better than in *Plaxton mesh*. However, the constant hidden within the "O" notation and a smaller logarithm base cause that their efficiency exceeds *Plaxton mesh* only when the number of nodes is very large, in reality, difficult to reach. Moreover, fault tolerance of the constant-degree DHTs in their pure form is very limited.

Symphony [19] is another interesting distributed hash table protocol based on a ring topology, which models the small-world phenomenon [13] and creates links between nodes in a probabilistic way. A node in *Symphony* establishes connections with its predecessor and successor on the ring, and with k "long" links chosen according to a probability distribution, which makes the expected path length $O(\frac{1}{k}\log^2 N)$, where k is the number of the maintained "long" links.

An extensive study regarding randomization in the construction of the connection graphs is presented in [20]. The authors show that randomization of edges may reduce the average length of shortest path between nodes, and how neighbor-of-neighbor greedy routing algorithm (taking neighbor's neighbors into account in next hop selection) can achieve asymptotically optimal expected route lengths.

Although the research is constantly being conducted in the topic of distributed hash tables, majority of the currently used DHT systems and applications are based on the architectures already mentioned in this section. The recent research focuses mainly on extending DHT overlays with specific features, such as Sybil attack resistance [16], resource discovery for grid computing [4, 26, 31, 34], as well as adapting DHTs to particular environments or applications [1], [9].

This paper is focused on improving general robustness of DHTs and, as an effect, presents a novel DHT architecture. The architecture is conceptionally similar to the solutions based on the tree geometry (*Plaxton mesh*, *Pastry*, *Tapestry*), however, extending the hierarchical geometry with a possibility of employing a multidimensional routing metric, which, together with the use of the variable Steinhaus metric, yields significant improvement in the resilience and performance of the system in the presence of node failures.

3 System architecture of HyCube

This section presents the routing architecture of *HyCube* - a distributed hash table system based on a hierarchical hypercube geometry. Sections 3.1 and 3.2 introduce the hierarchical hypercube concept and describe the structure of routing tables maintained by nodes. Section 3.3 presents the basic routing algorithm, which is analyzed in detail and optimized in the subsequent sections.

3.1 Hierarchical hypercube geometry

The routing geometry of *HyCube* is a combination of the tree geometry and the hypercube geometry. It is similar to *Plaxton mesh* [27], but nodes are logically located in vertices of a d-dimensional hierarchical hypercube. A hierarchical hypercube is a hypercube whose vertices are also (lower level) hypercubes. Vertices of the hypercubes at the lowest level are positions which may be assigned to nodes.

Figure 1 presents the structure of an exemplary hierarchical hypercube with 3 dimensions and 2 hierarchy levels. Node IDs are determined by their positions - the identifier of a node is a string of d-bit groups determining positions of the node in hypercubes at individual levels (starting with the hypercube at the highest level). The position in a hypercube at a particular level is a number built of bits corresponding to the positions of the node in the hypercube in individual dimensions. The length of the identifier equals $d \cdot l$, where d is the number of dimensions, and l is the number of levels.

The hierarchical hypercube and the tree geometries are isomorphic. However, the geometry of *HyCube* has a dual nature. Visualizing the structure as a hierarchical hypercube gives an idea of the non-hierarchical, spatial arrangement of nodes in a d-dimensional space - the numbers formed from bits corresponding to individual dimensions relate to the coordinates of the node in these dimensions in the system of coordinates with the center in point 0. Thus, considering the ID space as a segment of $\mathbb{Z}^d$ space ($\mathbb{Z}$ denotes the set of integer numbers), the distance between nodes may be

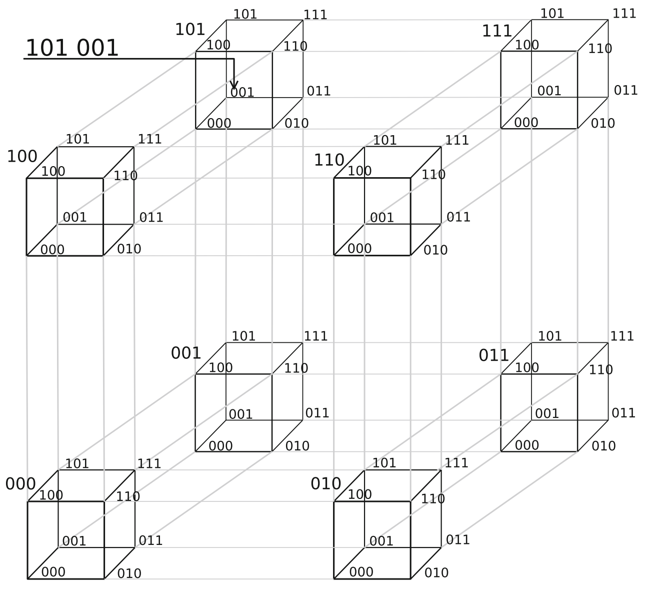


Fig. 1 A hierarchical hypercube (3 dimensions and 2 levels of hierarchy)

defined by any metric applicable to $\mathbb{Z}^d$, or $\mathbb{R}^d$ ($\mathbb{R}$ - the set of real numbers) as $\mathbb{Z}$ is a subset of $\mathbb{R}$. However, the geometry of *HyCube* should be seen as a d-dimensional torus with the perimeter equal to 2^l in each dimension (the set of coordinates in each dimension is treated as on a ring). That means that after the point $2^l - 1$, point 0 is located, and all arithmetic is done modulo 2^l. This fact is important in determining distances between nodes - in every dimension the distance is determined like on a ring - the shorter of the distances in either direction.

In *HyCube*, the default number of dimensions is 4 and the number of levels is 32, resulting in a 128-bit address space (which allows avoiding conflicts of identifiers in majority of applications).

3.2 Routing tables

3.2.1 Primary routing table

The primary routing table has the same structure as in *Plaxton mesh*. It has l levels (the number of hierarchy levels), and, at each level, there are 2^d slots (d - the number of dimensions). In the primary routing table of a node X, the slot j at level i ($i \geq 0$) contains a reference to a node that is located in the same hypercube at level $i+1$ and in the hypercube corresponding to the number j at level i (lower level). At each level $i > 0$, one slot corresponds to the hypercube in which the node X is located - this slot is left empty, as the routing table contains a whole level corresponding to this hypercube.

An exemplary primary routing table for a 2-dimensional hierarchical hypercube with 6 hierarchy levels for node $X =$

112013 is presented in Table 1. For clarity, groups of bits are represented by quaternary digits (base-4 numeral system). The digits in bold represent the sub-hypercube addresses corresponding to routing table slots at individual levels. The underlined digits represent the hypercubes corresponding to the routing table slots matching the hypercubes of node X at individual levels.

3.2.2 Secondary routing table

The secondary routing table of a node X contains nodes from adjacent hypercubes to the hypercube of node X in each dimension, in both directions, at each level. An adjacent hypercube (at any level) is the one whose coordinate in one dimension is greater or smaller by 1 than the coordinate of the same level hypercube of X (modulo 2^l), and coordinates in all other dimensions are equal to those of X. The secondary routing table does not contain nodes in slots at the highest level, as hypercubes corresponding to them are covered by the primary routing table. Also, one of the adjacent hypercubes at each level in each dimension is covered by a primary routing table slot.

The secondary routing table increases the level of flexibility in the next hop selection. If the distance between nodes is defined by a metric in $\mathbb{R}^d$ space, it is very likely that the secondary routing table contains nodes that are closer to any arbitrarily chosen node. Furthermore, it provides additional shortcut references when a message is routed to a node that is close in the $\mathbb{R}^d$ space, but is not close in terms of the *Plaxton mesh* distance. With the use of the primary routing table, routing a message between nodes sharing a short ID prefix would require many steps of traversing the tree.

Table 2 presents an exemplary secondary routing table for node X = 113012 (binary 01'01'11'00'01'10) - for a 2-dimensional hierarchical hypercube with 6 hierarchy levels. For clarity, in this example, the IDs are represented as binary numbers. Addresses of adjacent hypercubes corresponding to the routing table slots are marked in bold, and bits of addresses of adjacent hypercubes corresponding to the particular dimension are underlined - the numbers built of these bits are larger by 1 or smaller by 1 than the numbers formed of the corresponding bits of X. All other hypercube address bits (the remaining dimensions) are equal to the corresponding ones of X.

Table 1 The primary routing table for node 112013 (for a 2-dimensional hierarchical hypercube with 6 hierarchy levels

	0	1	2	3
Level 5	**0**11033	–	**2**31011	**3**00232
Level 4	1**0**2223	–	1**2**1301	1**3**0001
Level 3	11**0**113	11**1**201	–	11**3**302
Level 2	–	112**1**01	112**2**03	112**3**12
Level 1	1120**0**3	–	1120**2**1	
Level 0				–

3.2.3 Neighborhood set (closest neighbors set)

In addition to the routing tables described above, nodes maintain sets of closest to them (according to the chosen metric) nodes existing in the system - called *neighborhood sets*. These sets may allow finding a next hop (routing/lookup), decreasing the distance left to the destination, even if there are no appropriate nodes in both routing tables. Based on the same principle as sequential neighbors sets, the closest neighbors sets are expected to increase the probability of delivering messages in the presence of node failures. Furthermore, at one extreme, when next hops are found only in the nodes' neighborhood sets, the expected path length is $O(\sqrt[d]{N})$ (N is the number of nodes in the DHT). For sequential neighbors, the path length is proportional to N. Assuming the same probability of dropping a message by a single node along the path, the overall probability of dropping the message is smaller when the number of routing steps is smaller. If the probability of finding a next hop equals p, the probability of successful routing a message by k nodes equals p^k, which, for any $p < 1$ drops sharply for larger numbers of routing steps (k). The neighborhood sets have also very good properties for supporting joining and leaving procedures, as well as maintenance and recovery algorithms. Moreover, their existence is crucial for searching closest nodes for a given key (discussed in detail in Section 9), as well as for replicating resources. The default size of the neighborhood set is 16.

3.3 Basic routing algorithm

Let us consider routing a message from node X to node Y. Every node R along the route first checks if there is a reference to the destination node in its neighborhood set, in which case, the message is sent directly to that node. Otherwise, the routing tables are searched for an appropriate next hop - the node that shares at least one d-bit group longer prefix of ID with Y (than with R) or shares the same number of d-bit groups of the ID but is closer to Y than R in terms of the chosen routing metric. The routing metric of *HyCube* is discussed in Section 5. For the time being, let us assume that routing converges according to the Euclidean metric. The detailed algorithm is presented below:

1. Initially, the routing algorithm finds the slot in the primary routing table that corresponds to nodes sharing at least one group of d bits longer prefix of ID with Y than with the current node (R). In a hierarchical hypercube, this slot corresponds to a hypercube in which the destination node is located, at a lower level than the

Table 2 The secondary routing table for node 113012 (01’01’11’00’01’10) - for a 2-dimensional hierarchical hypercube with 6 hierarchy levels

	Dimension 0		Dimension 1	
	←	→	←	→
Level 5	–	–	–	–
Level 4	**01’00’**01’01’00’11	**00’00’**10’01’10’10	**11’11’**10’01’00’11	**01’11’**00’10’10’01
Level 3	**01’01’10’**10’01’00	**00’00’10’**11’01’01	**01’01’01’**11’01’01	**01’11’01’**01’00’01
Level 2	**01’01’10’01’**11’10	**01’01’11’01’**01’01	**01’01’01’10’**01’10	**01’01’11’10’**10’00
Level 1		**01’01’11’01’00’**10	**01’01’01’10’11’**00	**01’01’11’00’11’**10
Level 0				

lowest-level hypercube containing both, the current and the destination node. If for the current node and Y, the common prefix length equals i, and the next d-bit group of the ID of Y equals j, j-th routing table slot at level $l-1-i$ is used. If this slot is not empty, the message is routed to the node found in the slot - increasing the common prefix length with the destination node Y by at least one d-bit group.

2. If no node is found in the appropriate primary routing table slot, nodes sharing the same prefix length with Y as with R (in terms of the number of bit groups), but closer to Y in terms of the chosen metric, are also considered - both routing tables and the neighborhood set are checked. From the set of nodes found, the node sharing the longest prefix of the ID with Y (number of d-bit groups) is chosen, and, if there are more than one such nodes, the node closest to Y (according to the routing metric) is chosen for the next hop.

The primary routing table supports routing based on extending the ID prefix (in terms of d-bit groups) - tree-based routing (*Plaxton mesh*), and the secondary routing table supports finding a closer node in any dimension - such nodes are likely to be closer to the destination node also in terms of the Euclidean metric. Both routing tables and the neighborhood set are used for determining the best possible next hop, to which the message is routed.

It can be shown that the expected route length equals $\lceil \log_{2^d} N \rceil$ hops and, on average, $\lceil \log_{2^d} N \rceil \cdot (2^d - 1)$ slots are populated in the primary routing table and $(\lceil \log_{2^d} N \rceil - 1) \cdot d$ in the secondary routing table (N is the number of nodes in the network)[2].

3.4 Acknowledging message delivery and detecting duplicates

Optionally, after receiving messages (DATA messages - sent at the application level), depending on configuration, nodes may send acknowledgments (DATA_ACK messages) confirming receiving of the messages (either directly to the sending node, or routed via the system to the message sender). When the sending node receives the DATA_ACK message, it knows that the original message was successfully received and processed, and, when no acknowledgment is received (timeout), the node might resend the message (automatic resending may also be configured - up to the defined maximum number of retries). In cases when the acknowledgment mechanism is implemented at the application level (beyond the scope of *HyCube*), the native *HyCube* mechanism may be switched off.

HyCube also implements message duplicate detection. In case of receiving a message duplicate, the duplicate is dropped. Message duplicates may be received as a result of incorrect routing or network problems, or an ACK message not being delivered. Duplicates are detected based on the header of the message.

3.5 Number of dimensions versus number of levels

The address (node identifiers) space should be large enough to prevent potential conflicts of identifiers (existence of two nodes with the same ID). The size of the address space in *HyCube* equals:

$$N_{max} = 2^{d \cdot l} \tag{1}$$

which means that increasing the number of dimensions or the number of hierarchy levels would increase the number of possible identifiers that nodes may be assigned. However, the number or dimensions and levels of the hierarchical hypercube influences the system characteristics.

Adding additional levels of hierarchy causes the address space to increase, but it also proportionally increases the number of routing table slots that nodes maintain. Moreover, the pessimistic route length would also increase, as in the pessimistic scenario, routing steps correspond to individual routing table levels. Nevertheless, the expected route length remains at the same level, because, on average, similar number of routing table slots would be populated (with high probability the lower-level slots are empty).

Increasing the number of dimensions, on the other hand, has a very strong impact on routing characteristics. Although the routing tables grow sharply with the increase

[2]Based on the assumption that nodes are uniformly distributed in the hierarchical hypercube

of the number of dimensions, the routing algorithm is more specific in selecting next hops - in each routing step, the common prefix with the destination node ID is increased by a larger number of bits, maintaining the same pessimistic path length and decreasing the average path length. As the base of the logarithm (Eq. 2) grows exponentially, the expected path length is inversely proportional to the number of dimensions:

$$\log_{2^d} N = \frac{1}{d} \log_2 N \tag{2}$$

However, the maintenance cost is significant, as the primary routing table size grows exponentially with the number of dimensions:

$$\log_{2^d} N \cdot (2^d - 1) = \frac{\log_2 N \cdot (2^d - 1)}{d} \tag{3}$$

At one extreme, when the number of dimensions is equal to the number of identifier bits (1 level of hierarchy), every node would maintain references to all other nodes in the system, and the routing table slots would correspond to all possible values of node identifiers.

3.6 Prefix mismatch heuristic

In the final part of a route, when the message is already relatively close to the destination node, the routing algorithm may omit some nodes that are close to the destination, but do not share the same long or longer prefix of ID with the destination node than with the current node. This phenomenon becomes more significant when a multidimensional metric is used. That is why, like in [30], *HyCube* uses a heuristic switching to routing based only on the distance left, when a message is already in vicinity of the destination. Nodes should therefore be able to determine how close the message is to the destination in relation to the density of nodes in the space. In *HyCube*, before choosing the next hop, each node checks if the distance to the destination is shorter than the average distance to the nodes in the neighborhood set multiplied by the factor λ:

$$d_{dest} < \text{avg}(d_{neigh}) \cdot \lambda \tag{4}$$

If this condition is satisfied, all further nodes on the route are chosen based only on their distance to the destination node - they might not share the same long or longer prefix of the identifier. All nodes from both routing tables and the neighborhood set are checked and the closest node is chosen for the next hop.

The prefix mismatch heuristic may be also enabled, when no next hop is found with routing based on extending the common prefix length with the destination node. This behavior may be configured by changing a system parameter value. In many cases, the number of neighborhood set nodes matching the destination is much larger if the selection is based only on the distance. Obeying the prefix condition is a much stronger constraint on the next hops, and may cause more failed paths, especially in the presence of many node failures. Thus, although the path length might increase, by default, *HyCube* switches to routing based only on the distance left whenever no next hop is found. To maintain routing convergence, the prefix mismatch heuristic is followed by all subsequent nodes along the path.

The greater is the value of λ, the longer parts of routes will be determined based only on the distance left. The value should be large enough to ensure high probability of message delivery. However, too large values of λ could cause an increase in path lengths. Figure 2 presents the static resilience of *HyCube* consisting of 10’000 nodes, using the Euclidean metric, for several values of the λ parameter[3]. For better readability, the failed paths figure was scaled and cropped - the most significant differences were observed below 70 % of failed nodes. The performed simulations indicated that the value $\lambda = 1.5$ ensures good static resilience, without increasing the average routing path length.

4 Simulation methodology

To simulate the routing algorithm correctness, efficiency and resilience, an event-driven distributed simulator (multiple simulator instances running nodes) was created - a dedicated simulator that uses the real *HyCube* node library created by the author, in which case there is no need for any adjustments or modifications of the implemented algorithms - only the default network (transport) layer (UDP/IP) was replaced by a communication model based on JMS queues maintained by individual simulator instances. The created library and simulator were designed based on several guidelines presented in [3] - processing events for individual nodes was performed by only a certain number of threads on each simulator instance, which allowed simulations of very large systems (even much larger than 100’000 nodes) to be preformed on a limited number of physical machines.

The simulation scenario consists of the following steps. First, a given number of nodes with random identifiers are initialized, and each node joins the system by connecting to a randomly chosen node already connected to the DHT. When the DHT system is initialized, for varying numbers of failed nodes and for different system variants (algorithms used, parameter values), a given number of test messages are sent between random pairs of nodes remaining in the system. A node failure means removing the node from the

[3]The simulations were performed with forcing uniform distribution of neighborhood set nodes in terms of directions (discussed in Section 6.1)

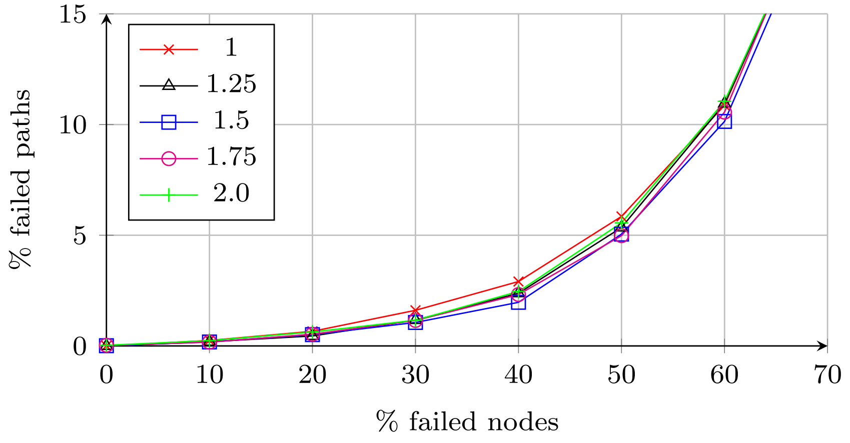


Fig. 2 Simulation results: static resilience of *HyCube* consisting of 10'000 nodes using Euclidean metric for different values of the prefix mismatch heuristic factor λ

network and waiting for other nodes to remove the reference from their routing tables (the mechanism described in Section 8.1). During the simulation, the simulator registers the number of messages successfully delivered to the destination, the number of failed routes, and calculates the average route length.

While simulating different network variants (comparisons), for all simulations run in a batch, the same sets of generated nodes (IDs) are used. They are connected to the system in the same sequence, using the same bootstrap nodes. Moreover, in order to eliminate differences in the results caused by different random simulation runs and to allow proper comparison of the simulated algorithms, the sets of nodes removed from the system (node failures), as well as the pairs of nodes between which test messages are sent, are generated once and are used for all simulations in the batch. When all DHT variants are simulated following the same scenario, it is possible to conduct a detailed comparative study of individual algorithms and parameter values.

5 Routing metric

In DHT systems, distances between pairs of nodes are defined by a certain metric, and the routing converges according to this metric. The choice of the metric has a great impact on the average route length and the probability of message delivery. Choosing a one-dimensional metric allows the use of sequential neighbors, which has proved to greatly improve the static resilience. If the number of sequential neighbors is s, half are predecessors and half are successors of the node, any message would be dropped only if all $s/2$ nodes in the appropriate direction failed. However, the use of sequential neighbors may cause a significant increase in path lengths in the case of node failures, when many routing table slots are empty, and a large number of nodes along the paths are found in the sequential neighbors sets. In *HyCube*, nodes are organized in a d-dimensional space, which allows the use of any metric applicable to $\mathbb{R}^d$ space. In comparison with sequential neighbors, the use of a multidimensional metric significantly decreases the expected path length when routing using only neighborhood sets. It can be shown that the expected path length is proportional to $\sqrt[d]{N}$, while for sequential neighbors, it is proportional to N (N - the number of nodes in the DHT). This fact becomes very important when considering network properties under churn or in the presence of node failures. However, by using a multidimensional metric, the network loses some properties connected with the existence of sequential neighbors. In a multi-dimensional space, it is not trivial to ensure uniform distribution of the closest neighbors set, ensuring that a certain subset of these nodes would match any potential direction. For a ring topology (sequential neighbors), there are only two possible directions, while for any number of dimensions larger than one, the number of possible directions is infinite.

As the neighborhood set nodes play a crucial role in maintaining high static resilience, the metric should provide the highest possible level of flexibility in next hop selection within the neighborhood set, which would directly translate to the overall resilience of the system. Let us consider routing using only neighborhood sets and assume that in each step, next hops are chosen only by the distance left to the destination (without ensuring the prefix condition). When a node chooses the next hop for a message, only a subset of the neighborhood set is closer to the destination node than the current node. It is crucial that the number of such nodes be as large as possible, so even in the case of many node failures, the message would not be dropped. The expected ratio of the number of matching nodes to the number of all nodes in the neighborhood set may be interpreted as the probability that a message may be routed to a single node in the neighborhood set. The following sections present a study maximizing this probability.

5.1 Common metrics in $\mathbb{R}^d$ space

The most common metrics in $\mathbb{R}^d$ space are Minkowski distances, defined by the following equation:

$$L_m(x, y) = \left(\sum_{i=0}^{d-1} |x_i - y_i|^m\right)^{\frac{1}{m}}, m \geq 1 \tag{5}$$

where x_i and y_i are i-th coordinates of x and y. In particular, L_1 is so-called Manhattan (or taxicab) distance, L_2 is the Euclidean distance, and L_∞ is the Chebyshev distance. L_1 defines the distance as a sum of distances in individual dimensions (absolute values of differences of coordinates). The set of points located in a constant distance from any point X, in 2-dimensional space, forms a square rotated by 45°, as presented in Fig. 3. In a 3-dimensional space, these points are located on an octahedron, and for spaces having larger numbers of dimensions, the equal-distance points are located on a cross-polytope (hyperoctahedron) having its center in the point corresponding to the node ID, and vertices located on lines parallel to the coordinate axes, passing through point X. The L_∞ metric is so-called "maximum" metric. The distance between two points is defined as the maximum of the distances in individual dimensions. In a 2-dimensional space, points being in equal distances to a certain point X are located on edges of a square with the center in the point corresponding to the X and edges parallel to the coordinate axes (Fig. 3). For larger numbers of dimensions, equal-distance points are located on a cube (3 dimensions) or a d-dimensional hypercube. L_2 (Euclidean) metric defines distance in a way that is the most natural and intuitive, corresponding to the way of measuring distances in everyday life. Equal-distance points are located on a circle in 2-dimensional space, a sphere (3 dimensions) or a d-sphere (d-dimensions).

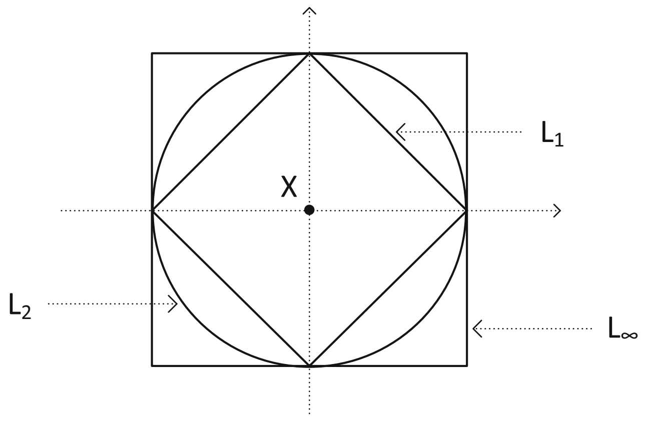


Fig. 3 Sets of points located in a constant distance from a node (L_1, L_2 and L_∞ metrics)

If the Euclidean metric is used, the probability that a message may be routed to a node in the neighborhood set may be calculated as a function of $k = \frac{d}{r}$, where d is the distance left to the destination node and r is the distance from the neighborhood set node to the current node. It is the ratio of the number of points (possible nodes positions) that are in the distance r from the current node and are closer to the destination than d, to the number of all points being at the distance r from the current node. Because coordinates of nodes are integer numbers, nodes may take only certain positions in the space. However, such accurate calculations would obscure the picture. For simplicity, instead of the number of positions, let us consider lengths of curves, areas of the surfaces and their equivalents for higher dimensional spaces, which is a good approximation, because normally we are not interested in numbers of nodes within exact distances, but rather in a more general estimation in a certain distance range. Figure 4 presents an exemplary visualization for a 2-dimensional space. If r is the distance to a certain node R being considered as the next hop, R is closer to the destination than the current node only if it is located on the dotted part of the circle of radius r (around the current node). The situation is analogous for higher numbers of dimensions (circle $\rightarrow$ d-sphere). The probability that a node in the neighborhood set is closer to the destination (for 2, 3 and 4 dimensions) equals (formulas derived by the author):

2d: $P(k) = \frac{\arccos\left(\frac{1}{2k}\right)}{\pi}$

3d: $P(k) = \frac{1-\frac{1}{2k}}{2}$

4d: $P(k) = \frac{2\arccos\left(\frac{1}{2k}\right)-\sin\left(2\arccos\left(\frac{1}{2k}\right)\right)}{2\pi}$

The calculated probability values are visualized in Fig. 5. The calculations show that the number of nodes in the neighborhood set, to which a message may be routed, strongly depends on $k = \frac{d}{r}$. The closer the message is

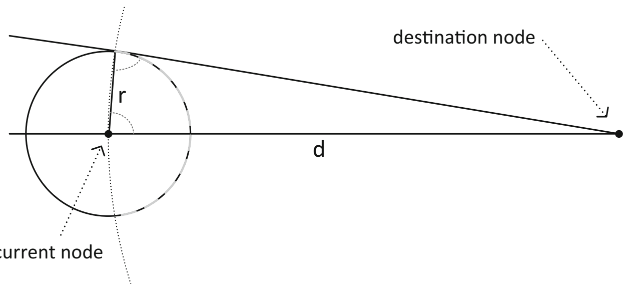


Fig. 4 Matching nodes for next hop selection (L_2 - Euclidean metric)

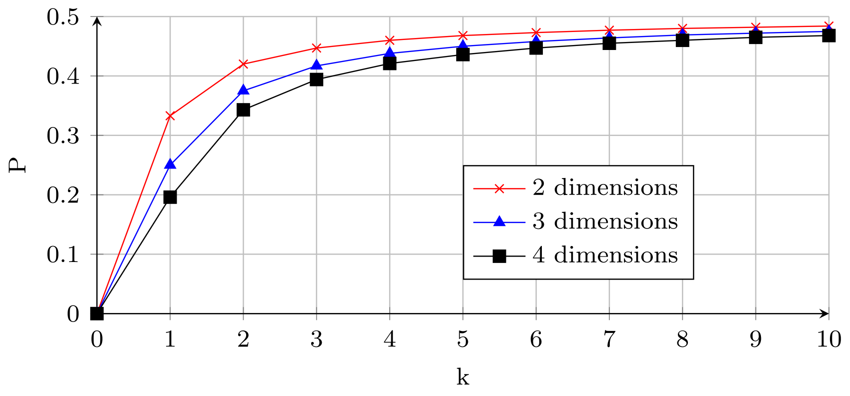


Fig. 5 Probability that a node in the neighborhood set is closer to the destination node than the current node ($k = \frac{d}{r}$)

to the destination node, the fewer appropriate nodes in the neighborhood set. The phenomenon becomes more significant as the number of dimensions increases. This fact has a great impact on static resilience - fewer node failures may cause the message to be dropped in the final parts of routing paths.

L_2 (Euclidean) is the only metric (from Minkowski distances) preserving distances between nodes after space rotation. This causes some undesirable properties of metrics L_1 and L_3 to L_∞. For instance, if these metrics are used, depending on the direction of the vector from the current node to the destination, the expected numbers of nodes in the neighborhood set to which the message may be routed are different. Figures 6 and 7 present examples of different scenarios of next hop selection (destination nodes located in different directions relative to the current node) for L_1 and L_∞ metrics. The bigger squares represent the points being in the same distance from the destination node as the current node, and the smaller squares represent the points located in the distance from the current node, in which a certain node R, being considered as the next hop, is located. The message may be routed to R, decreasing the distance left to the destination, only if it is located on the fragment of the smaller square that is inside the bigger square. It can be seen that the expected number of matching nodes (closer to the destination than the current node) strongly depends on the direction. For higher dimensional spaces, this becomes an even more serious problem, as there might be very few or no nodes matching certain directions. Due to this fact, only L_2 metric will be considered in the further discussion.

5.2 Steinhaus transform

In [6] and [7], the authors present the Steinhaus transform. The terminology comes from the fact that this distance was used in biological problems for the study of biotopes [21]. The theorem presented says that if D is a metric on a set X, D' is also a metric on X for any $a \in X$, where:

$$D'(x, y) = \frac{2D(x, y)}{D(x, a) + D(y, a) + D(x, y)} \tag{6}$$

Applying a metric with the Steinhaus transform to every route, setting the value of a to the ID of the source node, causes the next hops to be chosen in such a way that they are closer to the destination node and more distant from the source node. Such an approach increases the expected number of neighborhood set nodes to which messages may be routed in each routing step - although some closer nodes (according to metric D) might be considered more distant when the Steinhaus transform is applied, it allows sending messages using more roundabout routes, while still being convergent to the destination node.

One important remark should be made regarding the Steinhaus transform. In the case when $x = y = a$, Eq. 6 does not have a value (division by zero). Therefore, the

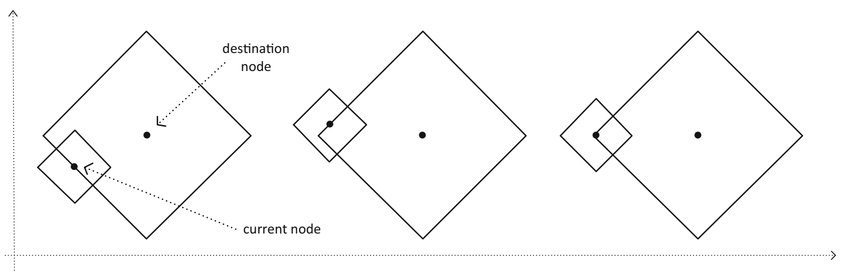


Fig. 6 Matching nodes for next hop selection (L_1 metric)

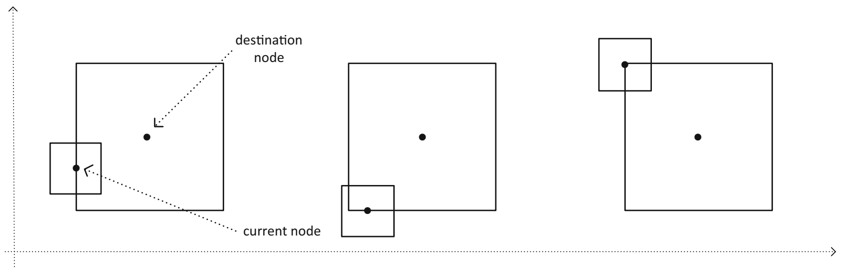


Fig. 7 Matching nodes for next hop selection (L_∞ metric)

value of the distance should be considered 0 if $x = y$, regardless of the value of a.

5.3 Variable metric adopting Steinhaus transform

The use of the Steinhaus transform yields very good routing parameters and very high static resilience for networks containing relatively few nodes. However, for networks containing much more nodes (denser), in final parts of routes, the addend $D(x, a)$ of the denominator in Eq. 6 (where x is the current node), has less influence on the value of the distance as its changes in individual steps are very small compared to the value of the entire denominator. Thus, the more nodes in the network, the lesser is the influence of the Steinhaus transform on the static resilience. However, a certain modification can be introduced - a variable metric adopting the Steinhaus transform, where point a would be changed by intermediate nodes along routes. The value of a would initially be set to the source node ID, and subsequent nodes, before choosing the next hops, would check whether they are closer (in terms of the Euclidean metric) to the destination than the current point a. In such a case, point a would be updated - would be given the value of the current node. Such a way of changing point a ensures that the routing is convergent to the destination (there will be no cycles on routes) and yields a very high level of flexibility in the next hop selection along the whole route, regardless of the network size. It is easy to notice that whenever the Steinhaus point has the value of the current node ID, messages may be passed to any other node, decreasing the Steinhaus distance left to the destination. With great probability the node closest to the destination in terms of the Euclidean metric is chosen. It may however not be a node that is closer (Euclidean) to the destination. Nevertheless, in such a case, the subsequent next hop selections would be more restrictive and converge to the destination faster. The expected route length is still proportional to $\sqrt[d]{N}$, and, owing to the increase in the flexibility in the next hop selection, the static resilience reached is even better than in systems using sequential neighbors, which can be seen in the simulation results presented in Section 7. Figure 8 presents a comparison of simulation results performed by the author - the probability that a message may be routed to a node in the neighborhood set for different metrics for a 4-dimensional network containing 1’000 nodes. The probability was calculated as the average fraction of

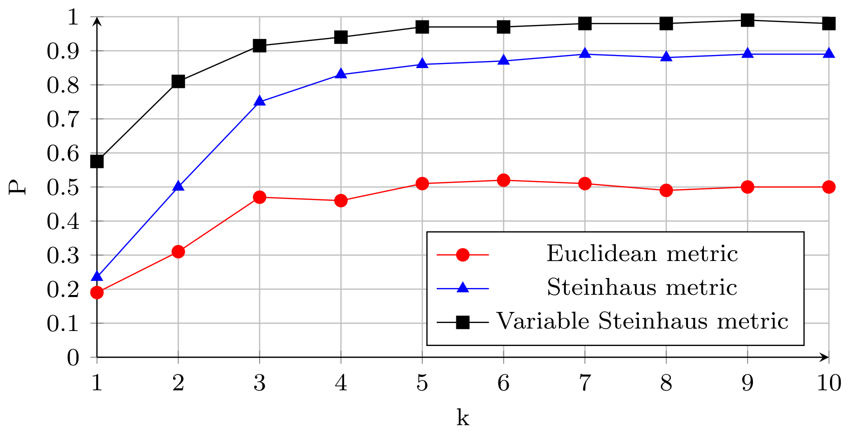


Fig. 8 Simulation results: Probability that a message may be routed to a node in the neighborhood set for Euclidean metric, Steinhaus metric and variable Steinhaus metric (network containing 1’000 nodes) - basic routing algorithm of *HyCube*. k is the ratio $\frac{d}{r}$, where d is the distance to the destination node, and r is the distance to the neighborhood set node being considered

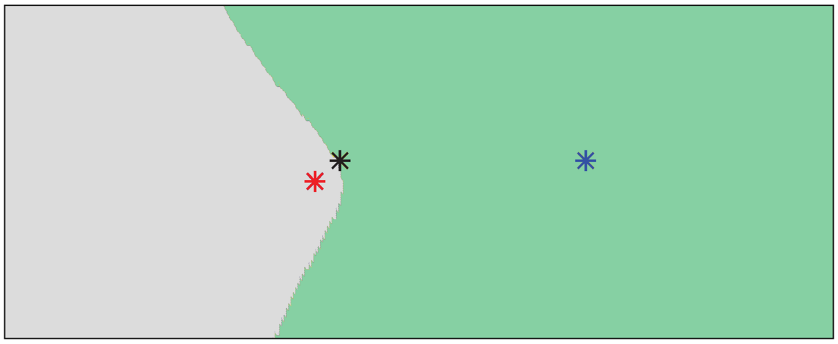


Fig. 9 Visualization of the influence of applying the Steinhaus transform on the next hop selection. The destination node is represented by the blue point, the current node is represented by the black point, and the red point is the current point a. The green part denotes the points that are closer to the destination than the current node, according to Steinhaus metric

neighborhood set nodes closer to the destination node than the current node - gathered from nodes involved in routing 1'000 messages between random pairs of nodes. The routing algorithm simulated did enforce the common ID prefix length condition. For comparison, if messages were routed using sequential neighbors in one-dimensional space (ring geometry), the line would remain approximately at the level 0.5, regardless of the distance left. With the variable Steinhaus metric, the level of flexibility is higher than for sequential neighbors. Any message would be dropped only if all the matching nodes failed, and the probability of such a case is significantly reduced with the use of the variable metric.

Figures 9 and 10 visualize the influence of applying the Steinhaus transform on the next hop selection. The destination node is represented by the blue point, the current node is represented by the black point, and the red point is the current point a. The green part denotes the points that are closer to the destination node than the current node, according to Steinhaus metric. It can be seen that the expected number of nodes in the neighborhood set to which the message may be routed is much larger than with the use of the Euclidean metric, and in case when the message was previously routed to a more distant node, the next hop selection is more restrictive.

It should be noted that whenever the Steinhaus point is equal to any of the node IDs between which the distance is measured, the distance value equals 1, regardless of the second argument value:

$$D'(x, y)\Big|_{a=x} = \frac{2D(x, y)}{D(x, x) + D(y, x) + D(x, y)} = 1 \tag{7}$$

$$D'(x, y)\Big|_{a=y} = \frac{2D(x, y)}{D(x, y) + D(y, y) + D(x, y)} = 1 \tag{8}$$

When calculating distances from multiple nodes to the destination node, if the Steinhaus point is given the value of the destination node ID, it is impossible to differentiate the nodes, as all the distances are then equal to 1. The routing algorithm is not exposed to such a situation - the Steinhaus point value would be equal to the destination point only

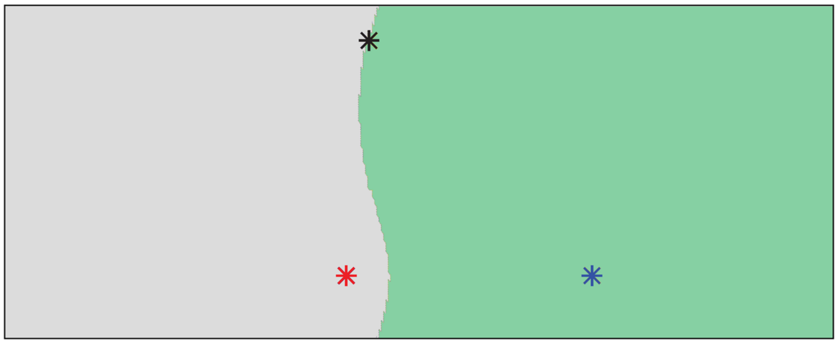


Fig. 10 Visualization of the influence of applying the Steinhaus transform on the next hop selection in the case when the message was previously routed to a more distant node. The destination node is represented by the blue point, the current node is represented by the black point, and the red point is the current point a. The green part denotes the points that are closer to the destination than the current node, according to Steinhaus metric

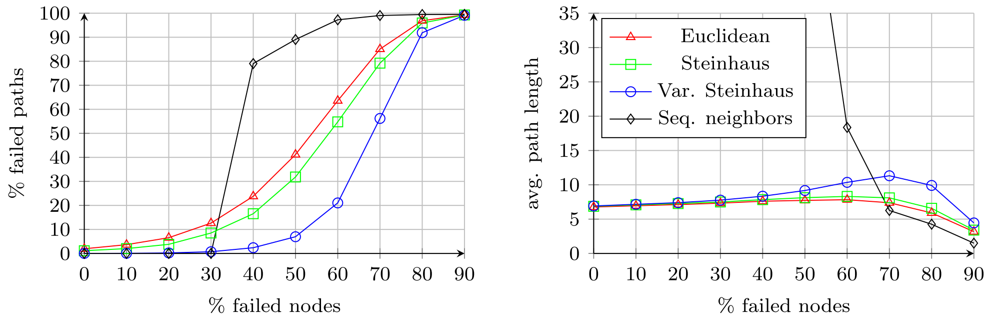


Fig. 11 Simulation results: Static resilience (% failed paths) and average path length in the presence of varying numbers of random node failures - basic routing algorithm with the use of closest neighbors sets only (without enforcing the prefix condition), network containing 10’000 nodes: Euclidean metric, Steinuaus metric, variable Steinhaus metric and sequential neighbors

when the destination is already reached. However, any other algorithm using Steinhaus distances, modifying the Steinhaus point, should take that fact into account (e.g. search algorithm, described in Section 9).

Figure 11 presents simulation results of static resilience and path length increase in the presence of varying numbers of node failures for Euclidean metric, Steinhaus metric, variable Steinhaus metric, and 1-dimensional metric using sequential neighbors - with the use of closest neighbors sets only, without enforcing the prefix condition (the routing converges only in respect to the routing metric). The figure clearly shows the advantage of adopting the Steinhaus transform. In particular, using the variable Steinhaus metric yields significant decrease of failed paths rate. Routing with the use of sequential neighbors yields very poor performance and resilience due to the fact that the average number of hops (path length), in this case, is proportional to the number of nodes in the system. For larger systems, the results become much worse. For 10’000 nodes, the path lengths (not covered by the figure) reach values exceeding 300 routing hops. Thus, sequential neighbors should not be used for routing on their own without any shortcut routing table support.

The simulation results of resilience and path length increase for the basic routing algorithm (all routing tables being used) of the system containing 10’000 nodes are presented in Fig. 12. The variable Steinhaus metric still has the best characteristics, although a longer average path length is achieved, compared to the sequential neighbors variant. However, for larger numbers of node failures, this increase is caused mainly by nonexistence of direct short paths between pairs of nodes and forcing the messages to be routed using more roundabout paths (for other metrics, many of these messages would be simply dropped).

Because using the variable Steinhaus metric increases the average path length, especially for higher node failure rates, another simulation was performed to test the behavior of the system with the Steinhaus transform enabled only for the final routing steps, when the prefix mismatch heuristic is already applied. Theoretically, such a modification could decrease the path length, preventing messages from being routed to more distant areas (according to the Euclidean metric) in initial steps, when the average hop distances are much larger. The use of the Steinhaus transform is the most important when the message is already in the proximity of the destination node, and the probability of finding the next hop in the neighborhood set drops sharply with the use of the Euclidean metric. If the Steinhaus transform is used only when the prefix mismatch heuristic is applied, it would

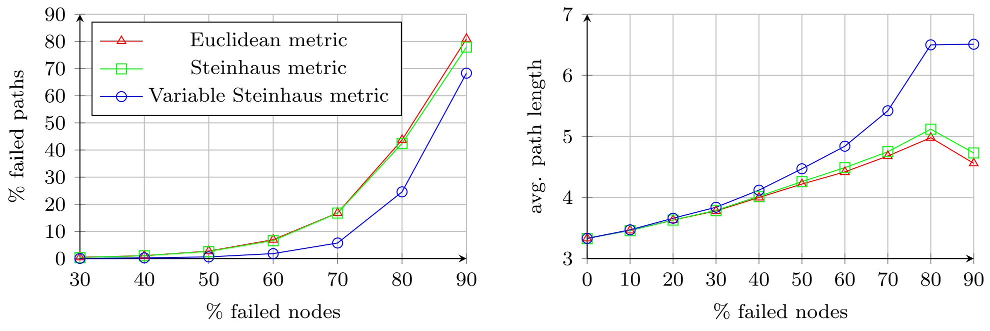
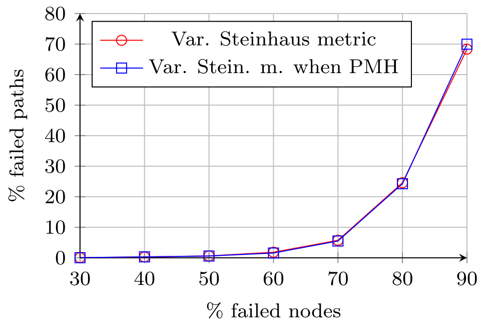


Fig. 12 Simulation results: Static resilience (% failed paths) and average path length in the presence of varying numbers of random node failures - basic routing algorithm (with the prefix mismatch heuristic applied), 10’000 nodes: Euclidean metric, Steinuaus metric, variable Steinhaus metric and sequential neighbors



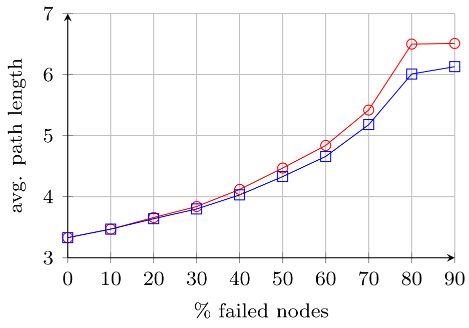


Fig. 13 Simulation results: Static resilience (% failed paths) and average path length in the presence of varying numbers of random node failures - basic routing algorithm (with the prefix mismatch heuristic applied), 10'000 nodes, variable Steinhaus metric used only when the prefix mismatch heuristic is already applied

be enabled when the message gets close to the destination, or when no next hop is found (which would also enable the prefix mismatch heuristic, allowing the Steinhaus transform to be used). Figure 13 presents the simulation results - a comparison of the system with the variable Steinhaus metric used in every next hop selection with the variant enabling the variable Steinhaus metric only when the prefix mismatch heuristic is already applied. The obtained results prove that, indeed, the proposed modification decreases the average path length. However, despite a significant decrease of the average path length, the modification did not cause any static resilience decrease.

5.4 Euclidean distance versus Steinhaus distance - re-routing using regular metric

The final steps of routing with the use of a metric with the Steinhaus transform applied may cause a message to be sent to a node that is more distant from the destination than it would be if the Steinhaus transform was not applied. Therefore, when, at some point, a node on a route cannot find the next hop in its routing tables and neighborhood set, it is possible that the route is ended in a point (node) that is not the closest one to the destination in terms of the Euclidean metric. For some applications, if the destination node itself cannot be reached, it is crucial to reach the closest possible node. Thus, *HyCube* introduces one more modification - when a message cannot be routed by a node, the node tries to route it again, based only on the Euclidean distance left to the destination. All consecutive next hops after that should be chosen in the same way. Such an approach will cause that, in the case the message is dropped, a relatively close node to the destination is reached. From the experiments (for a network containing 10'000 nodes, with 50 % failed nodes, routing using only neighborhood sets), it appears that applying this phase in routing allows messages to be sent to a closer node in about 80 % cases. Furthermore, as it can be seen in Fig. 14, this additional phase of routing increases the static resilience of the system.

There is, however, one drawback of such an approach - the failed path lengths (undelivered messages) may possibly increase due to re-routing with another metric after the next hop is not found. Nevertheless, messages usually get relatively close to the destination node, and, in most cases, only a small number of additional hops (if any) are performed.

6 Further enhancements

This section presents several modifications of the routing algorithm, which lead to the increase of the level of static resilience. The enhancements include ensuring uniform distribution of neighborhood set nodes, hypercube-aware next hop selection and preventing storing the same node in routing table slots corresponding to overlapping hypercubes.

6.1 Uniform distribution of neighborhood set nodes

The neighborhood set should provide possibility to route messages regardless of the direction in which the destination node is located. Therefore, it is important that nodes in neighborhood sets be uniformly distributed in terms of directions. There might be a scenario where some nodes would have more closest neighbors in one direction and no or very few neighbors in other directions. In such a case, the nodes would not be able to route messages in all directions (using neighborhood sets). The issue becomes more important in the presence of node failures, when the number of matching next hops should be as large as possible. When using sequential neighbors concept, the problem might be solved by splitting the set into two equal size sets - successors and predecessors. In this case, half of the closest nodes matches any message destination (successors or predecessors depending on the direction on the ring). In a multi-dimensional space, however, it is not trivial to ensure uniform distribution of the closest neighbors set, ensuring

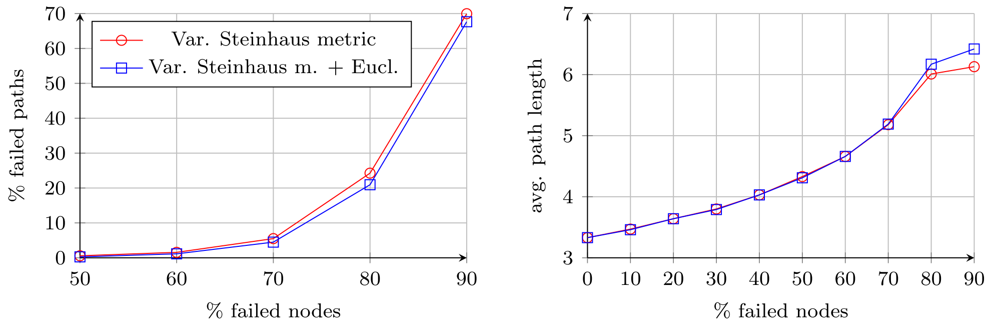


Fig. 14 Simulation results: Comparison of static resilience and path length increase of routing using the variable Steinhaus metric with and without the additional phase - routing according to the Euclidean metric (network containing 10'000 nodes)

that certain subset of these nodes would match any potential direction (message destination). Uniform distribution of nodes in terms of directions may be defined in a variety of ways, and many different algorithms may be employed to maximize this uniformity. Both, proximity and even distribution, should be considered, because ensuring uniform distribution of nodes in respect of directions may cause some more distant nodes to be included in neighborhood sets and pass over some closer nodes. The key is to maintain the good properties of neighborhood sets (as being the closest existing nodes), and to make these good properties valid regardless of the direction.

HyCube adopts a simple technique for ensuring uniform distribution of neighborhood set nodes in respect of directions. The technique splits the space into fragments (orthants[4] of the system of coordinates with the center at the address of the node whose neighborhood set is considered) and attempts to ensure that the number of neighbors is the same in each orthant. Within individual orthants, nodes are chosen based on their distances. Such a solution is very simple, efficient and does not require much computational overhead. The simulations (Figs. 15 and 16) proved the correctness of this technique.

An interesting fact was observed when comparing the path length increase of results obtained from simulations with the routing tables enabled and disabled (Fig. 15 compared to Fig. 16). When the uniformity of neighborhood set nodes arrangement is forced, the average path length gets decreased when messages are routed using only neighborhood set nodes. However, when the routing tables are used for finding next hops, the decrease is no longer visible (only for higher node failure rates). For lower failure rates, we even observe a slight increase in the path lengths. The obtained path length decrease (Fig. 15) is caused by larger distance decreases (to the destination node) in individual routing hops - the average neighborhood set node distance is larger. However, it may also lead to performing additional routing hops in the final part of the route, if the destination node is not included in the neighborhood set despite being close enough to be included if the uniform distribution was not forced. When most of the routing steps are found in the routing tables (routing tables enabled, at low node failure rates), the path length decrease resulting from faster convergence to the destination is lower than the average increase in the final part of the route. The observed path length increase (Fig. 16) is, however, insignificant.

6.2 Hypercube-aware next hop selection

When the routing algorithm cannot find a node sharing a longer ID prefix with the destination node than with the current node, a node sharing the same long prefix, but closer to the destination according to the chosen metric is selected as the next hop. When the message is routed within a hypercube corresponding to the common prefix length according to the distance only, it is completely independent of the hypercube hierarchy at lower levels. It is however possible to introduce one more criterion in next hop selection. If the message was routed to a node sharing the largest possible number of common bits in the first different digit (group of d bits), this would be the node in the closest possible hypercube at a lower level. If there were more than one such nodes, the next hop would be then chosen by the remaining distance (among the nodes with the same number of common bits in the first different digit). Theoretically, such an approach would increase the probability of finding the next hop sharing longer prefix (in terms of entire bit groups) by the next node(s) on the path - with the support of secondary routing tables. The simulation results presented in Fig. 17 confirmed that, indeed, the static resilience is slightly increased (although a slight path length increase was also observed) when the number of common bits in the first different digit of the next hop address is given a priority over the distance left.

[4]An *orthant* is the generalization (in d-dimensional Euclidean space) of a quadrant (2-dimensional space)

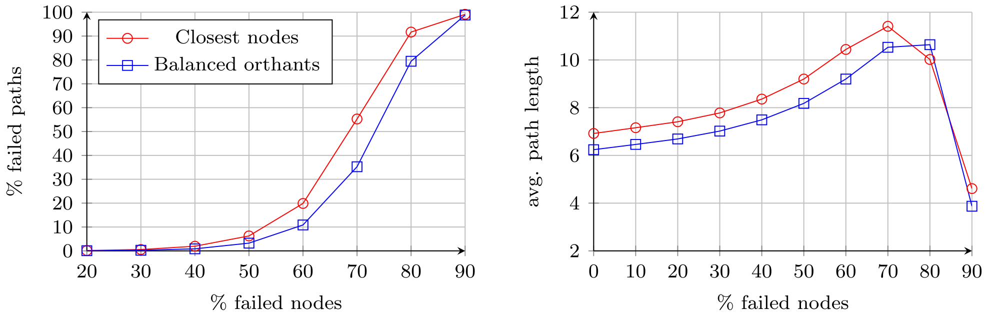


Fig. 15 Simulation results: Influence of ensuring uniformity of neighborhood set nodes arrangement in terms of direction (the algorithm ensuring balanced orthants) on static resilience - routing using **only closest neighbors sets** without enforcing the prefix condition, with the additional phase - routing according to the Euclidean metric, network containing 10’000 nodes

6.3 Routing table slots overlapping

Hypercubes corresponding to slots of primary and secondary routing tables may contain nodes that are covered by a slot at a lower level in the secondary routing table. If multiple different routing table slots contain the same node, when the node fails, all these slots become unusable. That is why, ideally, the routing tables should contain references to different nodes for all such overlapping hypercubes.

To overcome the overlapping problem, for the secondary routing table, it is enough to consider a node as a candidate for a matching routing table slot only if the corresponding adjacent hypercube is the lowest-level hypercube containing that node (for individual dimensions and directions). That would ensure that no secondary routing table slot would contain nodes that are located in overlapping lower-level adjacent hypercubes.

As far as the primary routing table slots are concerned, it may also be easily verified whether a certain node is covered by a lower-level secondary routing table slot (let us denote by Y its identifier, and by X the ID of the node whose routing table is being considered). X and Y are in adjacent hypercubes at levels $l-1-j$ to $l-1-i$ if and only if all $k < i$ first digits (d-bit groups) of X and Y are equal, and i-th to j-th digits differ on one (the same for all these digits) bit - this bit corresponds to the dimension in which the two hypercubes are adjacent. l is the number of levels, and d is the number of dimensions of the hierarchical hypercube. If $l-1-j$ is smaller than the corresponding primary routing table level of Y, l_{RT1}, that means that Y is located in an adjacent hypercube at a lower level than l_{RT1} and is thus covered by a lower-level secondary routing table slot. There might also be a case that Y is not in an adjacent hypercube in any dimension at any level - if the first different digit (d-bit group) of X and Y differs on more than one bit.

Figure 18 presents the influence of preventing routing table slot overlapping on static resilience. As expected, such a modification has a positive impact on static resilience, as it avoids situations when a failure of a single node would cause multiple routing table slots to become empty. A slight decrease in the average path length was also observed, as there are more potential matching next hops (possibly closer).

This modification, however, should not be applied to the neighborhood set, because the neighborhood set is crucial for maintaining high resilience and should always contain the closest nodes (as discussed in the previous sections). Thus, the neighborhood set might contain some nodes that are also included in routing tables. Furthermore, when considering a candidate for routing tables, no check whether the

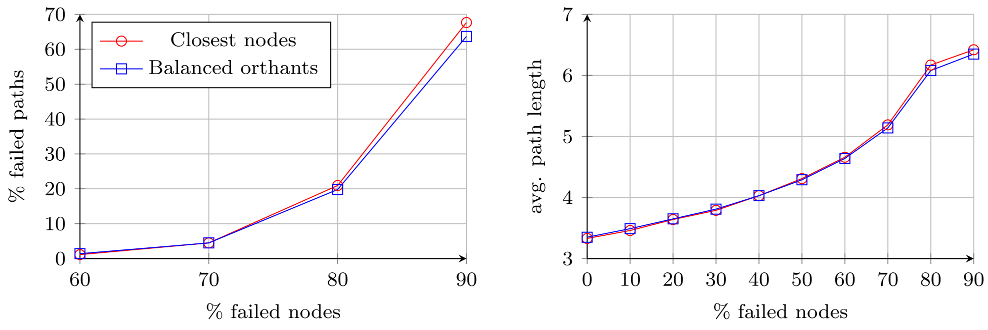


Fig. 16 Simulation results: Influence of ensuring uniformity of neighborhood set nodes arrangement in terms of direction (the algorithm ensuring balanced orthants) on static resilience (network containing 10’000 nodes)

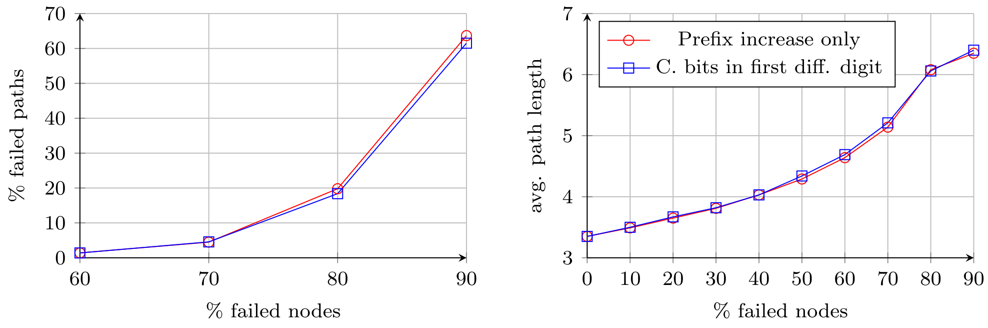


Fig. 17 Simulation results: Influence of respecting the number of common bits in the first different group on static resilience and path length increase (network containing 10’000 nodes)

node is in the neighborhood set should be performed. The neighborhood set is continuously updated, and such a check might very soon be “out-of-date”, while still having left the routing table slot empty. Therefore, due to its properties, the neighborhood set should be built independently from the routing tables.

7 Evaluation of static resilience and routing performance - comparison with sequential neighbors (*Pastry*)

The evaluation of *HyCube* was conducted in comparison with *Pastry* DHT, which is based on a similar geometry, uses sequential neighbors sets and is able to efficiently locate nodes/resources by routing a single message. Systems built on *Kademlia* protocol base their search algorithms and their resilience on increasing the routing state and at the cost of the number of messages being exchanged (messages being sent to multiple nodes in parallel). Thus, although *HyCube* also supports a search method based on sending queries to multiple nodes (see Section 9), the routing efficiency and resilience were compared to a system using routing-based node location - *Pastry*.

Figures 19 and 20 present experimental results obtained in simulations of *HyCube* and *Pastry* for systems containing 1’000 and 10’000 nodes. The properties evaluated were the static resilience and the average route length (based only on successful routes). To present simulation results for the same sizes of routing tables, results for *HyCube* with the secondary routing table switched off were also included. Because *Pastry* is supported by sequential neighbors sets, and routing in *HyCube* is based on the variable Steinhaus metric, for both DHTs, very high levels of static resilience were observed. Although for lower rates of failures, *Pastry* achieves slightly shorter path lengths (due to *HyCube* applying more roundabout routes with the use of the Steinhaus metric), *HyCube* achieves much better resilience for higher failure rates. Furthermore, for higher failure rates, the path lengths achieved by *HyCube* are significantly shorter due to the use of a multidimensional metric. It should also be noted that for *HyCube*, a larger number of longer successful routes were included in the average path length calculation - to nodes to which only roundabout paths exist. As the number of failed nodes increases, next hops are more often found only in the closest neighbors sets, in which case, the path lengths increase. As discussed at the beginning of Section 5, with the use of a multidimensional metric, the average

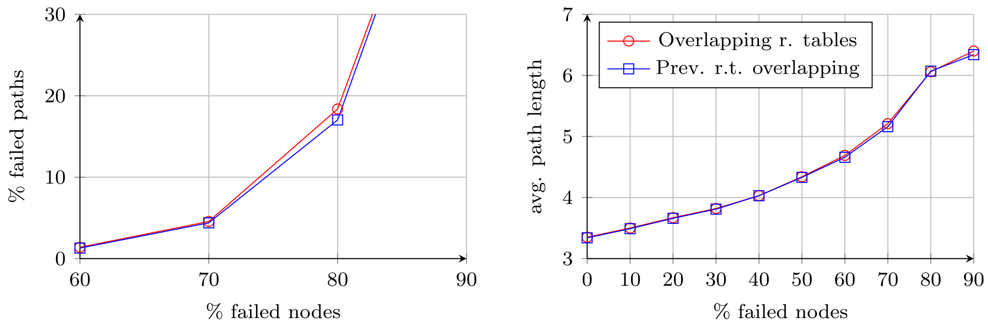


Fig. 18 Simulation results: Influence of preventing routing table slots overlapping on static resilience and path length increase (network containing 10’000 nodes)

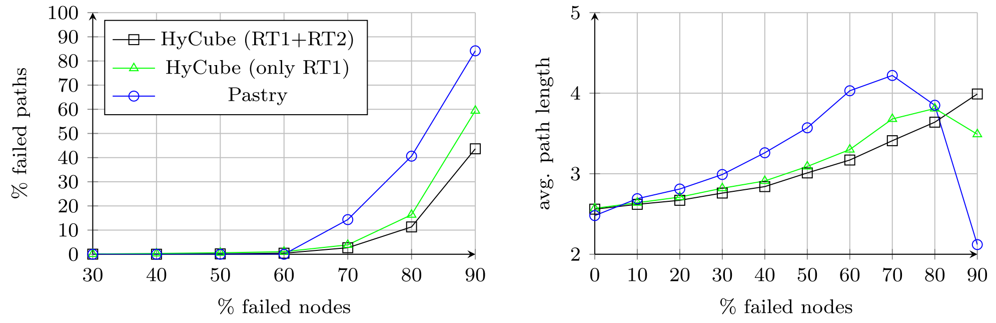


Fig. 19 Simulation results: Comparison of static resilience and average path length increase for varying rates of random node failures in *HyCube* and *Pastry* (1'000 nodes)

distance decrease (distance left to the destination node) is much more significant with each routing hop. For sequential neighbors, the expected length of a successful path grows sharply with the growth of the number of nodes, while the probability of a message delivery decreases exponentially in relation to the expected path length - a sequence of multiple independent experiments (binomial distribution). It can be seen (Figs. 19 and 20) that the superiority of the variable Steinhaus metric over sequential neighbors gets more and more distinct with the growth of the network size (the number of nodes).

8 Routing table nodes selection

For systems with a certain degree of flexibility in neighbor selection, additional priorities may be applied to choose nodes, based on some other criteria. In [11, 37] and [5], several common routing table node selection techniques are discussed. These techniques are useful, for example, for building topologies exploiting physical proximity between nodes (PNS), or taking into account node liveness information and maintaining connections to nodes that are likely to remain in the system for longer (LNS). The mechanism used in *HyCube* is a variant of the LNS approach, which, at the same time, provides means to remove failed nodes from routing tables.

8.1 LNS (liveness node selection) in HyCube

HyCube uses a variant of LNS which bases the neighbor choice on node liveness information discovered locally, working together with a background process checking node's responsiveness. However, the *HyCube* LNS implementation does not calculate the liveness information based on the node's join and leave times. Every node periodically sends keepalive (PING) messages to all the nodes in its routing tables and updates the stored nodes' liveness values. The value is increased when the node responds to the PING message (sending a PONG message), and is decreased if the keepalive response is not received (timeout). The initial liveness value (new nodes) is defined by the system parameter L_{init}, and the values are updated as follows (author's definition):

$$L = L_{prev} \cdot p + (1 - p) \cdot L_{max} \quad (9)$$

if the keepalive confirmation (PONG message) was received, or:

$$L = L_{prev} \cdot p \quad (10)$$

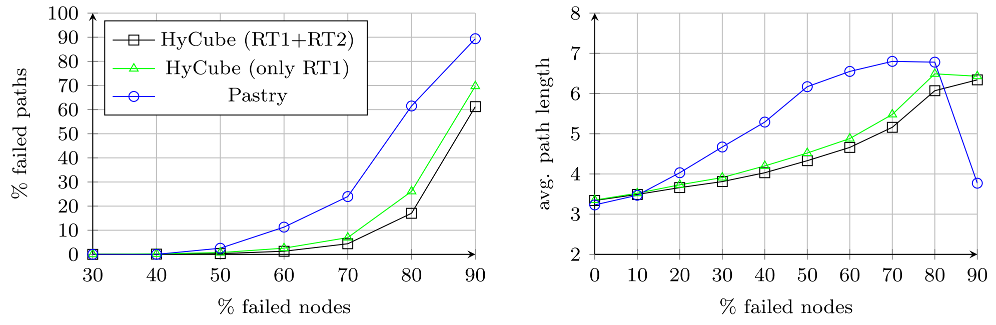


Fig. 20 Simulation results: Comparison of static resilience and average path length increase for varying rates of random node failures in *HyCube* and *Pastry* (10'000 nodes)

when the keepalive fails (no PONG message is returned). L_{prev} is the previous value of liveness, L is the new, updated liveness value, L_{max} and p are system parameters. When L falls below the threshold value $L_{deactivate}$ (system parameter), the node is marked as deactivated and is not used in the routing procedure until L falls below the threshold value L_{remove} (parameter), in which case the node is permanently removed from routing tables, or until L reaches above $L_{deactivate}$ again. Whenever the value of L is below the value of the parameter $L_{replace}$, when possible, the current node in the routing table slot is replaced by a new node, whose value L is then given the initial value L_{init}. The value of L for any node removed from the routing table should however be stored by nodes for some time to prevent replacing new nodes that temporarily fall below $L_{replace}$ with nodes that were permanently unresponsive. The values $0 < p < 1$ (update coefficient), $L_{deactivate}$, $L_{replace}$ and L_{remove} determine the sensitivity of the algorithm, and L_{max} defines the maximum value of L. This technique provides a good way to keep long-living responsive nodes in routing tables and eliminate failed or overloaded nodes that are not able to handle requests. Reasonable values of p, $L_{deactivate}$, $L_{replace}$ and L_{remove} should be chosen to prevent the algorithm from removing nodes from routing tables in case of temporary delays in sending responses by nodes, but still, to be able to efficiently remove failed nodes. If $L_{max} = 2$, the values of $L_{init} = 1.5$, $p = 0.5$, $L_{deactivate} = 1$, $L_{replace} = 0.5$ and $L_{remove} = 0.05$ are good defaults as they allow avoiding failed paths due to temporary nodes' unresponsiveness, at the same time allowing unresponsive nodes to be replaced by new nodes. Using these default values would cause deactivating nodes after a single failed keep-alive, however, not replacing/removing them immediately. The parameter values (as well as the keep alive interval) may be fine-tuned depending on the system characteristics. The flexibility in next hop selection guaranteed by *HyCube* ensures that no significant increase in failed paths rate, nor the average path length, is observed even if large portions of nodes temporarily do not respond to keep-alive messages and are deactivated.

8.2 *HyCube* mixed mode node selection

The LNS technique described above allows replacing a routing table node only when its value of L drops below $L_{replace}$ (or when L reaches L_{remove}, in which case the node is removed). In some cases, however, it would be desirable to take another criterion (or criteria) into account, like for example proximity or any application specific measure. To achieve that, the condition $L < L_{replace}$ may be relaxed, and the neighbor selection could be based on a function measuring both factors - the liveness and the second factor (algorithm-specific). The author proposes the following definition of the liveness fulfillment factor:

$$Fact_L = \left| \frac{L - L_{replace}}{L_{replace}} \right|^{e_L} \cdot \operatorname{sgn}(L - L_{replace}) \tag{11}$$

where the exponent e_L determines the exponential growth of $Fact_L$ depending on the difference $L - L_{replace}$. Depending on the algorithm, this factor may be used to calculate overall criteria fulfillment at a node level, improvement of the criteria fulfillment for a new node, relative to current node(s), or may be used in calculating the quality function value for a set of nodes (for example for the neighborhood set). An exemplary neighbor selection criterion at a node level might be a weighted sum of two factors:

$$Q = \alpha \cdot Fact_L + \beta \cdot Fact_X \tag{12}$$

where $Fact_X$ may be any neighbor selection algorithm specific function.

9 Lookup and search procedures

The previous section concentrated on the geometry and the routing algorithm of *HyCube*. From the DHT perspective, two other algorithms are also of a great importance - lookup and search. Instead of routing a message, in many cases, it is more adequate for a node to find a node or nodes that are the closest ones in the system to a given key (node ID) and communicate with these nodes directly. Because lookup and search algorithms are performed by multiple nodes, the whole lookup/search process will be referred to as lookup/search procedures. Lookup is a procedure performed by a node that results in finding the node closest to a given ID, while search results in finding a requested number of closest nodes to the given key in the system. During the lookup/search procedure, the initiating node communicates with other nodes in the system, retrieving information about their neighbors. Using the references returned, the searching node makes decisions which nodes should be contacted next, until the closest nodes are found. No message is routed further, and, in each step, the decision which nodes should be contacted next is made locally. The lookup and search procedures are crucial for serving fundamental DHT functions - storing and retrieving resources. This section describes the lookup and search algorithms of *HyCube* DHT, proposed by the author, and presents the results of performed simulations confirming their correctness.

In both, lookup and search procedures, the initiating node may require the requested nodes to return multiple references to nodes closest to the specified key. The routing tables are searched exactly as described in Section 3.3 - the next hop selection finds the requested number of best next hops, according to the same criteria. If multiple references are supposed to be returned by the next hop selection algorithm, the prefix mismatch heuristic (Section 3.6) or falling back to Euclidean routing (Section 5.4) is applied only when no nodes are found at all. Often, the next hop selection algorithm might not be able to find as many nodes fulfilling the next hop criteria as requested. In such cases, if at least one node is found, the next hop selection is considered successful, and the node/nodes found are returned to the requestor.

9.1 Lookup procedure

In the node lookup procedure, as opposed to routing, next hop selection is made by the initiating node. Nodes receiving the lookup request do not route the message, but they send back (to the initiating node) a reference to a node or nodes that are the best next hop candidates. However, the decisions regarding the next node selection are made locally. This is sometimes referred to as iterative routing, as opposed to recursive routing. To ensure that the procedure is convergent to the lookup node, next hops must satisfy the same conditions as in the routing procedure, i.e. they must share a longer prefix with the destination node than with the previous node or share the same prefix length but be closer to the lookup node than the current node. The metric used is the same as the one used in routing, and the prefix mismatch heuristic also applies whenever a node close enough is reached. When the lookup node is found, it may then be contacted directly, and application-specific operations may be performed. Although node lookup requires more network traffic, and the overall lookup latency is greater (the number of messages exchanged is at least doubled), the main advantage of such an approach is the possibility of returning and sending the lookup request to another node whenever any node requested does not return closer references. This reduces the risk of a route failure due to any single node not being able to route the message, and additionally makes it possible to find the lookup node using more roundabout paths. The node lookup procedure is initiated with three parameters:

- key - the ID of the lookup node
- β - the maximum number of nodes returned by intermediate nodes
- γ - the number of temporary nodes stored during the lookup

During the whole lookup, the initiating node maintains a set Γ containing γ closest (Euclidean) nodes found so far. The lookup procedure consists of two phases:

1. Initially, Γ is filled with at most γ closest nodes (to the lookup node ID) from local routing tables and the neighborhood set (variable Steinhaus metric), and the initial lookup request is sent to the closest node found. After receiving a response from the requested node (max. β references), Γ is updated so that it contains γ closest (Euclidean) nodes from the nodes currently stored in Γ and the returned nodes, and the next request is sent to the node returned. If, however, no node is returned, the next request is sent to the closest node in Γ that has not been yet requested. For every node stored in Γ, virtual route information is maintained - the Steinhaus point and flags indicating whether the prefix mismatch heuristic has already been enabled for this virtual route and whether the Steinhaus transform has been switched off (no closer nodes found with the use of Steinhaus transform). Initially, for all nodes found in local routing tables, the Steinhaus point is set to the ID of the initiating node, and the prefix mismatch heuristic flag is set based on the analysis of the local neighborhood set. These values are included in every lookup request, and the requested nodes should take these parameters into account during next hop selection (local closest nodes search). Every requested node, during the next hop selection, updates the Steinhaus point for the virtual route if it is closer to the destination (Euclidean) than the current Steinhaus point, determines (based on its neighborhood set) whether it is close enough to the lookup node to enable the prefix mismatch heuristic (which then remains enabled for this virtual route), and whether the Steinhaus transform should not be applied any more (no closer nodes found with the use of Steinhaus transform). The updated parameter values are sent back to the requestor within the lookup response message. The procedure continues until the lookup node reference is returned, or until no new nodes are returned.
2. When the lookup node is not found (the lookup request has been sent to all the nodes in Γ and no closer nodes were returned), the lookup procedure is repeated on the current set Γ, but is based on the Euclidean metric (no Steinhaus transform), and the prefix mismatch heuristic is enabled. Because for certain nodes (virtual routes) in Γ, the prefix mismatch heuristic might have already been enabled and the Steinhaus transform might have already been disabled, there is no need of sending additional requests to them. The closest node found after this phase is considered to be the closest node in the DHT.

The initial lookup (local), in addition to the closest nodes found in the local routing tables, should also consider adding the initiating node (itself) to Γ, as it may be one of the closest nodes to key. If the initiating node ID is one of the γ closest nodes found locally, it should be placed in Γ. Furthermore, the initiating node ID may be returned by any intermediate node (for any virtual route after changing the metric to Euclidean, the initiator node ID may become closer than the current node). No additional local search is however required in the first phase, as it was already performed initially. However, when this node ID (self) is still in Γ after switching to the second phase (Euclidean metric and prefix mismatch heuristic in use) a local search should be performed with the prefix mismatch heuristic enabled and based only on the Euclidean distances. Otherwise, some potential references to closer nodes (stored locally) might be omitted.

When $\beta = 1$ and $\gamma = 1$, the node lookup procedure creates one virtual route that is exactly the same as if the message was routed. By increasing β and γ, in case of failures, the procedure is capable of returning and creating alternative routes, and allows sidestepping inconsistent fragments of the connection graph.

9.2 Search procedure

The purpose of the search procedure is locating k nearest nodes to a given key (identifier) in terms of the Euclidean metric. Search is a basic functionality of a distributed hash table - it is used for storing and retrieving resources and when a new node joins the network. The closest nodes search, like the node lookup procedure, is managed locally by the initiator, which sends search requests to nodes and analyzes the responses. The decisions to which nodes next requests should be sent, are made by the initiating node. The search procedure is initiated with the following parameters:

- key - the ID for which k nearest nodes should be found
- k - the number of closest nodes to the key to be found
- α - the number of closest nodes to which search requests are sent (parallelism factor)
- β - the maximum number of nodes returned by intermediate nodes ($\beta \geq k$)
- γ - the number of temporary nodes used during the search ($\gamma \geq k$; $\gamma \geq \alpha$)
- ITN - "ignore target node" - determines whether the set of k nodes should contain the exact match (the node with the identifier equal to the requested key) or not

During the whole search, the initiating node maintains a set Γ containing at most γ nodes, closest (in terms of the Euclidean metric) to key found so far. The initiating node sends search requests to the closest nodes and processes the responses received. After receiving a response from any node (to which a search request has been sent), the set Γ is updated so that it contains at most γ nodes closest to key (Euclidean). The search procedure consists of two phases:

1. In the first phase, the set Γ is initially filled with maximum γ nodes from local routing tables and the neighborhood set, that are the closest to key (variable Steinhaus metric). In the consecutive steps, the initiating node sends the search request to nodes in Γ which have not yet been requested and are within α closest nodes to key found so far. After receiving a search request, the receiving node looks for at most β nodes closest to key in its routing tables and neighborhood set. As opposed to the lookup procedure, the closest β nodes returned in the search procedure do not have to satisfy the prefix condition, and do not have to be closer to key than the current node. However, the routing tables are searched for β nodes sharing the longest prefix with key, and among the nodes sharing the same prefix length, the choice is made based on the distance to key (with the use of the variable Steinhaus metric if the prefix mismatch heuristic is already switched on). Upon receiving a search response, the initiating node updates Γ with the nodes returned, so that it contains the closest (Euclidean) nodes to key from the nodes currently stored in Γ and the newly returned nodes. For every node in Γ, the current Steinhaus point is stored, as well as a flag indicating whether the prefix mismatch heuristic has already been applied for finding the node. These values are included in search requests sent to nodes, and the local searches are performed based on these parameters. These parameters may be modified by the nodes processing search requests (updating Steinhaus point if the current node ID is closer to key than the current Steinhaus point, and enabling the prefix mismatch heuristic based on the local neighborhood set), and the updated values are included in the search responses. The first search phase finishes when none of α currently closest to key nodes returns any node closer to key than the most distant node in Γ - all α closest nodes in Γ have been already requested and returned the responses (or the requests timed out).
2. The second search phase is a repetition of the first phase, however, requests are sent to all the nodes in Γ, the prefix mismatch heuristic is enabled, and only the Euclidean metric (no Steinhaus transform) is used in local searches. The search is finished when none of the nodes in $\varGamma$ returns any node closer to key than the most distant node in $\varGamma$. The result of the procedure is k closest nodes to key from $\varGamma$.

The second search phase (without the use of the Steinhaus transform) is extremely important, because during the search, nodes are allowed to return also more distant nodes to the key than themselves. There would never be a situation in which no node is found in the routing tables (unless the routing tables are empty, in which case switching to Euclidean routing would as well return no nodes). Thus, the routing tables would never be searched for nodes closest in terms of the Euclidean metric (finding no nodes is the condition for switching off the Steinhaus transform), which could lead to omitting some nodes that are close to key.

The initial search (local), in addition to the closest nodes found in the local routing tables, should also consider adding the initiating node itself to Γ (if the initiating node ID is one of the γ closest nodes found locally), as it might be one of the closest nodes to key in the system. Furthermore, a reference to the initiating node may be returned by any requested node, in which case, no additional local search is required in the first search phase, as it was already performed initially. However, when this node ID (self) is still in Γ after switching to the second phase (Euclidean metric and prefix mismatch heuristic in use), a local search should be performed again with the prefix mismatch heuristic enabled and based only on the Euclidean distances. Otherwise, some potential references to closer nodes (stored locally) might be omitted.

The value of γ should be at least equal to k. However, it may be greater than k to allow more roundabout search paths. Especially, for small values of k, such roundabout paths are important so that the procedure finds also the nodes that are located in the hierarchical hypercube on the opposite side of the point key than the initiating node.

Whenever a local search (next hop selection) is performed by a certain node for the key equal to its own identifier (which would update the value of the Steinhaus point to the value of key - it would be the closest point reached so far), the search should be continued without the use of the Steinhaus transform (all virtual routes starting from this node). If the value of the Steinhaus point is equal to the key (destination), as described in Section 5.3: $D'(x, y) = \frac{2D(x,y)}{D(x,a)+D(y,a)+D(x,y)} = 1$, regardless of the distance $D(x, y)$. This fact would make it impossible to differentiate nodes based on the distance to key. Such a scenario would never happen during the lookup procedure, as finding the exact match would immediately return it, and no further lookups would be performed. However, it is very important when considering the search procedure.

If the parameter ITN is set to *true*, the value of this parameter is included in every request message, and all local searches (next hop selections) should skip the exact match. This parameter is useful when a node performs a search for closest nodes to its own identifier. In such a case, returning a reference to itself would make no sense. There is also no need to re-check local routing tables after switching to the second search phase, because when the distance equals 0, the initial local search would be performed with the prefix mismatch heuristic enabled, and the Steinhaus metric would not be applied anyway.

As opposed to the lookup procedure, in the search procedure, the initial values of the Steinhaus points (for initial nodes in Γ found locally) are not set to the initiating node ID, but to the IDs of the initial search nodes themselves. If the initiating node is closer to key than the requested node (at one extreme equal to key, in which case the Steinhaus transform would never be applied), the influence of using the Steinhaus transform could possibly be reduced.

9.3 Resilience of the lookup and search procedures

The results of performed simulations show that both, lookup and search procedures presented, are able to find the closest nodes to a given key even in the presence of many node failures. Figures 21, 22, 23 and 24 present the results obtained from simulations of the accuracy of lookup and search procedures in the presence of varying rates of random node failures, for varying values of α, β, γ, and $k = 8$ (for search procedure). During the simulations, 1'000 lookups/searches for random node IDs were performed by randomly selected nodes, and the number of closest nodes missed was saved for every lookup/search. For the lookup procedure, the "missed nodes" value in the figures is the average number of closer nodes existing in the system than the node returned by the lookup procedure. For the search procedure, it denotes the average total number of nodes missed - nodes closer than the most distant node found, not included in the search result. Two independent simulations have been performed - for networks consisting of 1'000 and 10'000 nodes.

The simulations confirmed that increasing the value of γ has a great influence on resilience of lookup and search procedures, especially for higher rates of node failures, which is caused by allowing returns and continuing the lookup/search using different nodes. The changes of β have less effect on the resilience, especially for $\beta \geq 2$. Moreover, for higher values of γ (≥ 4), the resilience achieved for the search procedure does not change significantly with the increase of the number of nodes, which proves very good scalability of the system.

10 Self-organization

Previous sections discussed the architecture of *HyCube* and its behavior in static conditions. In practice, large-scale distributed hash tables are highly dynamic systems and should be backed by the support of algorithms for joining/leaving the network, as well as maintenance and recovery processes, maintaining good routing properties under dynamically changing conditions.

The algorithms described in this section are based on nodes exchanging information about references stored in routing tables and neighborhood sets. When a node receives a notification about existence of some other peer or receives references maintained by some other node (requested for

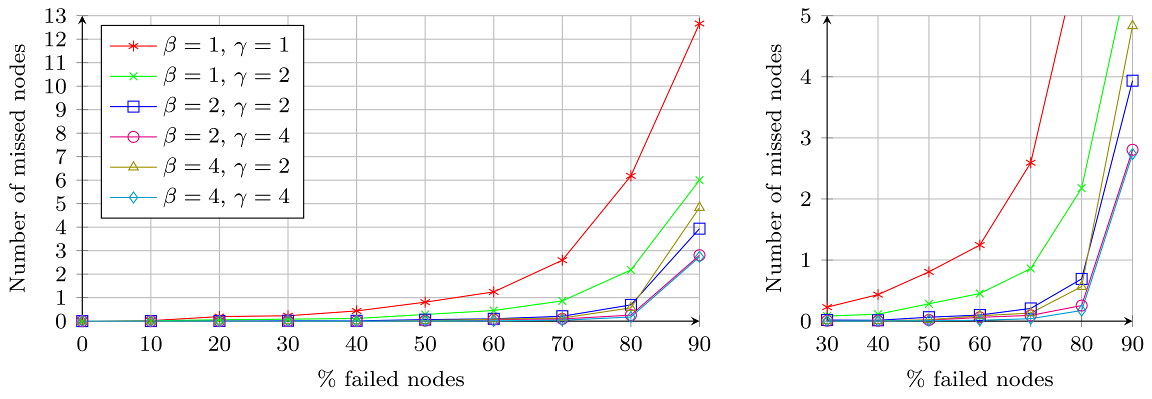


Fig. 21 Simulation results: Accuracy of the lookup procedure for varying rates of random node failures, 1'000 nodes

example in the recovery or joining process, or generated by the notifying node itself), it should analyze the node and check if the reference should be stored in the routing tables or the neighborhood set. Every node is analyzed as a candidate to the neighborhood set and appropriate routing table slots. For the primary routing table, this is the slot at the level calculated based on the common prefix length with the node ($l - 1 - commonPrefixLength$, l - the number of hierarchy levels) at the position equal to the first different digit (group of d bits, d - the number of dimensions). For the secondary routing table, the slot corresponds to the lowest-level hypercube containing the analyzed node, adjacent to the hypercube of the analyzing node. A simple check may be performed to verify this condition: nodes X and Y are in adjacent hypercubes at levels $l - 1 - j$ to $l - 1 - i$ if and only if i first digits (d-bit groups) of X and Y are equal, and i-th to j-th digits (numbered starting from 0) differ on one (the same for all these digits) bit - this bit corresponds to the dimension in which the two hypercubes are adjacent, and only the slot at the lowest level should be considered (to prevent placing the same nodes in overlapping routing table slots at different levels). Furthermore, depending on settings, additional rules described in Section 6.3 may apply to prevent storing nodes in the primary routing table slots overlapping with the lower-level secondary routing table slots. For selecting nodes within individual routing table slots and within the neighborhood set, the algorithms described in Sections 8 and 6.1 apply.

10.1 Maintenance and recovery

Maintenance is a process of ensuring good system properties that takes place continuously (ensuring the relevance of routing tables, updating information about neighbors). The maintenance mechanism used in *HyCube* is directly connected with the routing table neighbor selection (described in detail in Section 8.1). Every node periodically sends keepalive messages to all the nodes in its routing tables, and, depending on whether the keepalive response is received or not, the node's liveness value is updated. Based on the liveness value, the routing table reference may be skipped in the next hop selection (to avoid failed paths), may be replaced, or removed from the routing tables.

Recovery is a process of repairing the network topology and propagating necessary information over the network on account of a topology change (new nodes joining/leaving the system), topology breakdown (node failures) or attacks.

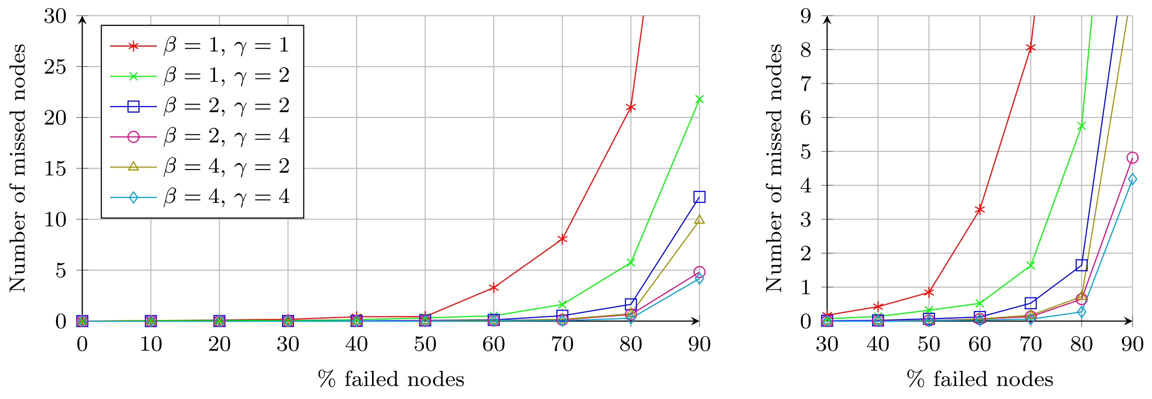


Fig. 22 Simulation results: Accuracy of the lookup procedure for varying rates of random node failures, 10'000 nodes

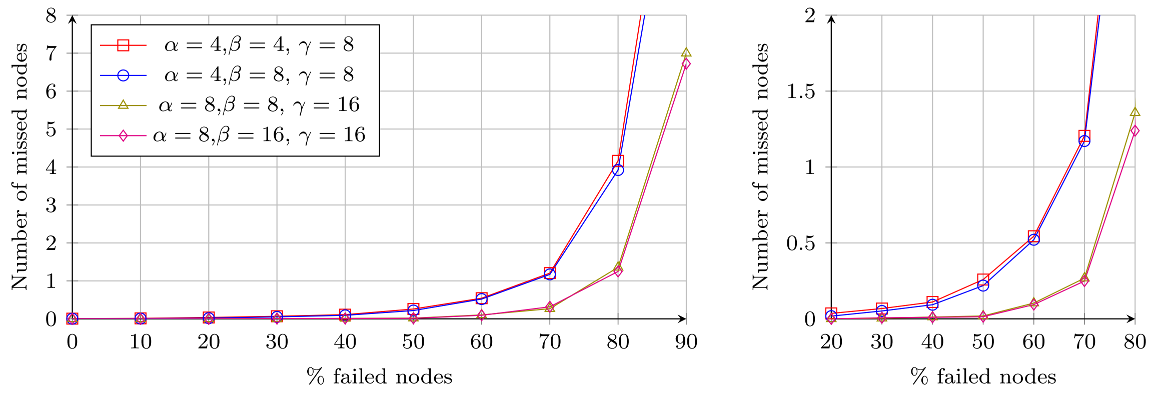


Fig. 23 Simulation results: Accuracy of the search procedure for varying rates of random node failures, 1'000 nodes

It is a crucial element of the design of highly dynamic systems. There are two main approaches to the network recovery - reactive and periodic recovery. The reactive approach immediately reacts to failures - loss of a reference in the routing table, while, in contrast, periodic recovery simply processes the recovery procedure every certain time interval, and the process takes place independently of the routing table change detection. The authors of the study presented in [29] claim that there is little difference in efficiency of periodic and reactive recovery. However, there is a large difference in the bandwidth consumed under different churn rates. Reactive recovery appears to be very efficient, and periodic recovery is wasteful if there is no churn. However, as the churn increases, reactive recovery becomes very expensive.

The proposed recovery technique, employed by *HyCube*, is a periodic procedure, run every specified time interval (parameter value). The value of the interval should be adjusted depending on the DHT nature. The higher is the level of churn, the recovery should be run more often, while if the system has a very stable nature, this interval might be larger. The recovery algorithm in *HyCube* proceeds in two phases:

1. In the first phase, the node sends a recovery request to all nodes in its routing tables, which it turn return their routing tables to the requesting node. Upon receiving the responses, the requesting node processes the references returned and updates its routing tables and neighborhood set.
2. In the second recovery step, the node performing the recovery sends a notification (NOTIFY message) about its existence to all nodes in its neighborhood set and routing tables (immediately after sending the recovery requests). The notification is also sent to the routing tables nodes to spread the information about the node to more distant parts of the hypercube. However, to limit the overhead, the maximum number of the routing table nodes to which notification is sent should be limited - the message would be sent to a limited number of nodes (random selection from all routing table nodes).

Because such an approach may lead to exchanging a large number of messages and also increase processing overhead at the node level, neighborhood set recovery - a variant of the recovery was introduced. Nodes performing the neighborhood set recovery send the recovery requests only to nodes in their neighborhood sets, which dramati-

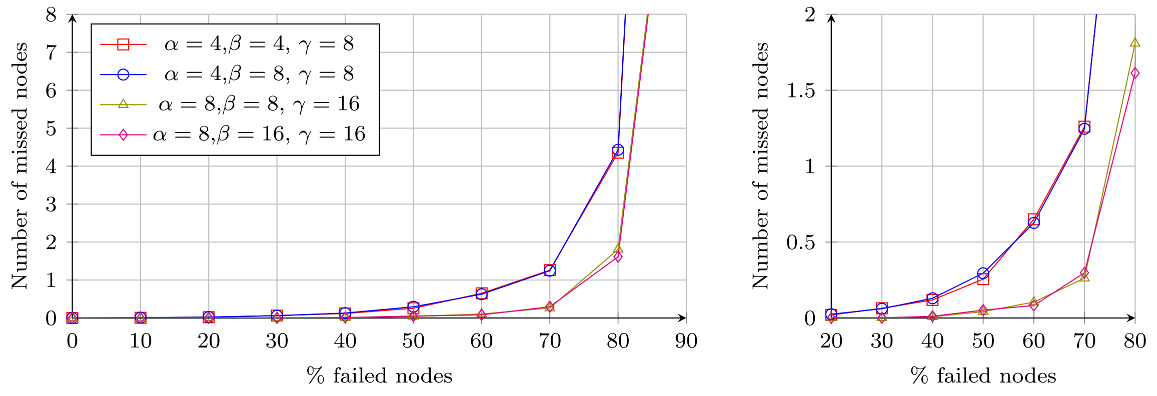


Fig. 24 Simulation results: Accuracy of the search procedure for varying rates of random node failures, 10'000 nodes

cally decreases the network traffic and processing overhead. In addition to the recovery interval, *HyCube* allows defining the recovery plan - determining a sequence of recovery variants (for individual recovery runs, successive recovery types from the recovery plan are performed - following a cyclical pattern). Full recovery has better properties in terms of keeping the routing tables up-to-date. However, especially for larger networks, the overhead is significantly larger. Although there might be situations in which the neighborhood set is completely corrupt, and sending recovery requests to the routing table nodes would return much more nodes, usually it is sufficient to repeat the neighborhood set recovery instead. Therefore, the neighborhood set recovery is the default recovery method used in *HyCube*.

To spread the information about the node in an even greater degree, it is possible to send a notification also to every node to which a reference is returned in recovery responses. However, this approach proved to generate enormous network traffic, as well as very high nodes' processing overhead. Performed simulations proved that notifying all neighborhood set nodes and 16 random routing table nodes during every recovery does not significantly increase the overhead, and, if the recovery procedure is run frequently enough, it is sufficient to properly propagate the information about the node's existence. Furthermore, depending on configuration, nodes may process the recovery request messages as notifications (analyze the sender as a routing table candidate), in which case, a notification message does not have to be later sent to the nodes to which the recovery requests were sent.

Figure 25 presents the simulation results (numbers of failed routes) for a network initially containing 10'000 nodes in the presence of varying numbers of random node failures (routing messages between the nodes left in the DHT), after processing the recovery procedure by every node 0, 1, 2, 3, 4 and 5 times. The results obtained prove the correctness and very good efficiency of the recovery mechanism.

10.2 Joining the system (connecting to the DHT)

When a new node joins the existing DHT, it should have knowledge about any node already connected to the system. *HyCube* implements two different approaches for joining the system. One of them is based on routing a JOIN message through the system to the node closest to the new node's ID, and the routing tables are formed by the references returned by intermediate nodes along the path (including also themselves), as well as the closest node. The second method is based on searching for the nodes closest to the new node's ID, which form initial routing tables for a new node. The routing approach requires much less overhead - a smaller number of messages are exchanged. However the search method is not vulnerable to any single node possibly causing the join message to be dropped. This section presents both techniques as well as the evaluation of their efficiency.

10.2.1 "Route-join" procedure

The route-join procedure is based on the joining mechanism presented in [30]. To initiate the route-join procedure, the joining node should send a "join" request (JOIN message) to any node already participating in the system. The message is routed to the closest possible node to the joining node's ID (however, omitting the exact match, as the message would be possibly routed back to the joining node). All intermediate nodes send back (directly to the joining node) JOIN_REPLY messages that contain all references stored by them in their routing tables. Every node returned is analyzed by the joining node and its routing tables are updated based on the criteria described in Sections 8 and 6.1. As the message gets closer to the new node ID, references returned are more likely to be good candidates for the neighborhood set. Furthermore, as the common prefix length with the joining node ID increases, more and more routing table slots correspond to the same hypercubes as the one of the joining node,

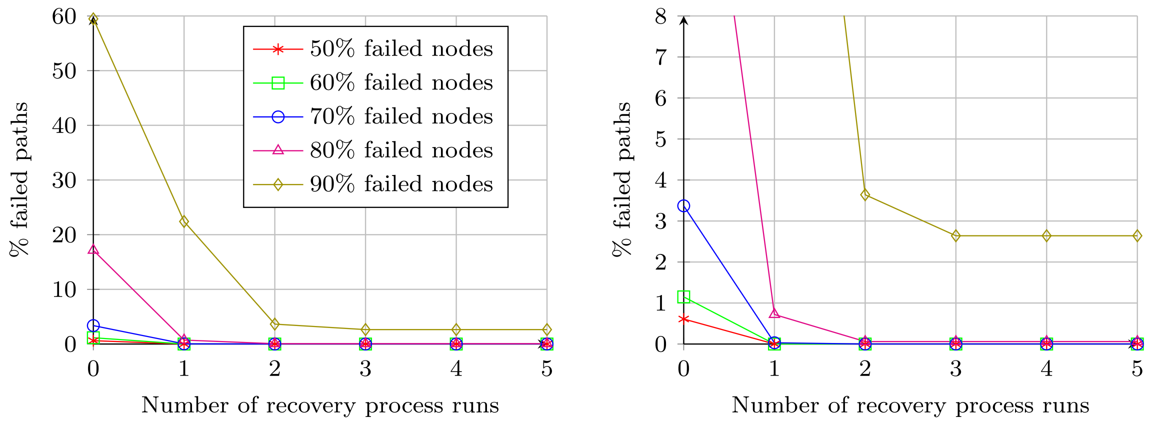


Fig. 25 Simulation results: Recovery procedure efficiency - numbers of failed routes for a network initially containing 10'000 nodes in the presence of varying numbers of random node failures, after processing the recovery procedure by every node: 0, 1, 2, 3, 4 and 5 times

so more and more references returned by the intermediate nodes might be also used for building the routing tables. For that reason, it is a good idea to disable the prefix mismatch heuristic for JOIN messages (the behavior is configured by a system parameter).

10.2.2 "Search-join" procedure

In the search-join procedure, the joining node initially retrieves all references from routing tables from a known node connected to the system. The references returned are used to perform a search (Section 9.2) for the closest nodes to the joining node ID with certain values of parameters α, β, γ: α_{join}, β_{join} and γ_{join} (system parameters), and the ITN (ignore target node) parameter set to *true*. The value of k is not important, because all nodes returned by intermediate nodes are processed (updating the routing tables). Initially, the set Γ is filled with γ closest (Euclidean) nodes retrieved in the initial phase - from the bootstrap node. Afterwards, the search procedure proceeds as described in Section 9.2.

Because the node performing the search is the node whose identifier is being looked for, to allow the use of the Steinhaus transform, the initial values of Steinhaus points should not be set to the joining node ID (Euclidean metric would be then used for local next hop selections), but should be given the values of IDs of the initial nodes themselves (only for the initial nodes returned by the bootstrap nodes - for any nodes added to Γ later, the Steinhaus point is updated based on the information contained in the response message that contained the node reference).

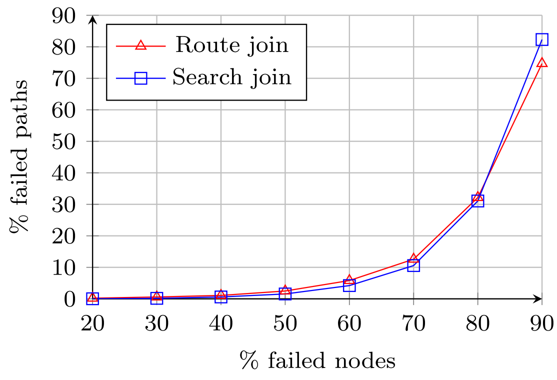
10.2.3 Evaluation of the join procedures

Figure 26 presents the static resilience (simulation results) obtained for the system built using the route-join and search-join procedures. The join procedures in these simulations were supported by a single run of the recovery procedure (by every node after joining), because join mechanisms themselves are usually not able to provide enough nodes to properly fill the routing tables. The closest nodes found are useful to serve as an initial neighborhood set and may match several routing table slots, but the recovery is still required to obtain additional references. Furthermore, the recovery procedure spreads the information about the joining node in the network. In real systems, the levels of resilience are much higher due to the fact that the recovery procedure is executed in the background at certain time intervals, which helps populating routing tables and spreading the information about new nodes. However, stopping the recovery background process (executing it only once by every node after joining) allowed an accurate comparison of both joining techniques. The comparison, however, didn't indicate significant difference between the route-join and search-join procedures in terms of static resilience (a very slight advantage of the search-join technique), and slightly shorter path lengths were achieved for the route-join. However, the search-join technique has been chosen as the default join mechanism in *HyCube* because of its resistance to a single node possibly causing the join message to be dropped. With the background recovery mechanism switched on, the differences in path lengths would be quickly eliminated. With one or more additional runs of the recovery procedure, the simulations show no differences between the two techniques.

Detailed simulations of the search-join showed that very good properties are reached (while still limiting the unnecessary overhead) for the values of β and γ equal to the size of the neighborhood set size, and $\alpha = \gamma/2$. These values may be decreased if the recovery process is conducted in a regular time interval after joining.

10.3 Leaving the system

The maintenance and recovery mechanisms should be able to maintain connection graph consistency and keep routing tables of nodes up-to-date, ensuring good routing and searching properties. However, these algorithms have



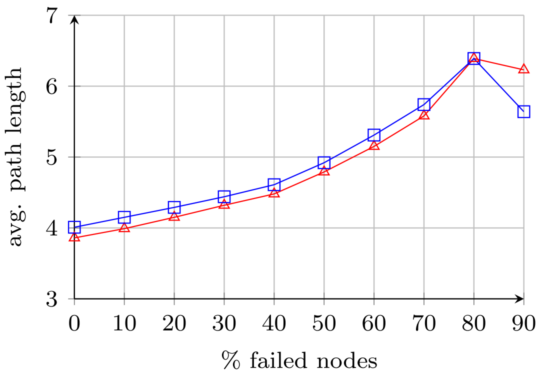


Fig. 26 Simulation results: Efficiency of route join and search join procedures - static resilience simulated after creating the system with the recovery procedure run once by every node after joining - 10'000 nodes

certain “inertia” and react with a certain delay. A simple solution to overcome this problem is sending the leave information (LEAVE message) to all neighbors in the leaving node’s neighborhood set. Such LEAVE messages should contain the list of neighborhood set nodes of the leaving node, and the nodes receiving it should remove references to the LEAVE sender from the routing tables and the neighborhood set, and process all the nodes included in the message to immediately fill the lost routing table or/and neighborhood references.

11 Conclusion

This paper presented a novel distributed hash table model, based on the hierarchical hypercube geometry and a variable metric adopting the Steinhaus transform - *HyCube*. In [11], it was proved that maintaining sequential neighbors sets dramatically improves the static resilience, as they provide a very high level of flexibility in next hop selection. As the flexibility in route selection directly translates to the level of static resilience, systems incorporating the sequential neighbors concept to their designs (e.g. *Pastry*) present extremely good routing properties and very high resistance to node failures and changes of the overlay structure. The use of the variable Steinhaus metric, discussed in the paper (Section 5), results in an even higher level of flexibility in next hop selection than provided by sequential neighbors. The number of the neighborhood set nodes to which messages may be routed (maintaining routing convergence), in most cases, significantly exceeds the number of matching sequential neighbors. The experimental results (Section 7) confirmed that the new approach results in a much higher level of resilience to node failures, as well as a significantly shorter average routing path length than the use of sequential neighbors sets (*Pastry*). Furthermore, the presented distributed hash table provides means of efficient key/identifier lookup and search (Section 9). The proposed lookup and search algorithms proved to be efficient and accurate even in the presence of high node failure rates.

Although static resilience characterizes the system tolerance to node failures without the use of any recovery mechanisms, it also directly influences the behavior of the system under churn. Although the routing table node selection mechanism presented in Section 8.1 proved to quickly eliminate failed nodes from the routing tables, and the recovery mechanism (Section 10.1) efficiently rebuilds lost connections, in systems where nodes are constantly joining and leaving, the routing tables of individual nodes always contain a certain number of failed nodes, as the maintenance and recovery mechanisms react with some delay. Therefore, it is even more appropriate to assume that such systems constantly work at a certain level of node failures. The static resilience also affects the efficiency of the recovery processes, as well as the procedures of joining the system (discussed in Section 10), which are very important aspects of real-life large-scale dynamic systems. The simulations demonstrated the correctness and effectiveness of the presented techniques.

The experimental results indicated that *HyCube* is efficient (route lengths), scalable and much more resilient to node failures than solutions exploiting the sequential neighbors concept. Moreover, during the study, a library implementing the protocol was created, resulting in a ready to use, efficient and credible implementation of a distributed hash table.

References

1. Bhattacharya A, Yang Z, Zhang S (2010) Temporal-DHT and Its Application in P2P-VoD Systems. Proceedings of the 2010 IEEE International Symposium on Multimedia (ISM ’10) pp 81–88, doi:10.1109/ISM.2010.21
2. de Bruijn N (1946) A combinatorial problem. Koninklijke Nederlandse Academie van Wetenschappen Proceedings 49:758–764
3. Buchmann E, Böhm K (2004) How to Run Experiments with Large Peer-to-Peer Data Structures. Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS04) doi:10.1109/IPDPS.2004.1302938
4. Cheema AS, Muhammad M, Gupta I (2005) Peer-to-peer discovery of computational resources for grid applications. Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing pp 179–185, doi:10.1109/GRID.2005.1542740
5. Chun BG, Zhao BY, Kubiatowicz JD (2005) Impact of Neighbor Selection on Performance and Resilience of Structured P2P Networks. Proc of the 4th International Workshop on Peer-To-Peer Systems (IPTPS 2005), LNCS 3640 pp 264–274, doi:10.1007/11558989_24
6. Clarkson KL (2006) Nearest-Neighbor Searching and metric space dimensions MIT press. Nearest-Neighbor Methods for Learning and Vision: Theory and Practice
7. Deza M, Laurent M (1997) Geometry of Cuts and Metrics, Algorithms and Combinatorics, vol 15. Springer, Verlag. ISBN 978-3-642-04295-9
8. Druschel P, Rowstron A (2001) PAST: A large-scale, persistent peer-to-peer storage utility. Proceedings of the Eighth Workshop on Hot Topics in Operating Systems pp 75–80, doi:10.1109/HOTO.2001.990064
9. Fersi G, Louati W, Jemaa MB (2013) Distributed Hash table-based routing and data management in wireless sensor networks: a survey. Wirel. Netw 19(2):219–236. doi:10.1007/s11276-012-0461-0
10. Fraigniaud P, Gauron P (2006) D2b: A De Bruijn Based Content-addressable Network. Theor. Comput. Sci. - Complex networks 355(1):65–79. doi:10.1016/j.tcs.2005.12.006

11. Gummadi K, Gummadi R, Gribble S, Ratnasamy S, Shenker S, Stoica I (2003) The impact of DHT routing geometry on resilience and proximity. Proceedings of the 2003 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM '03) pp 381–394, doi:10.1145/863955.863998
12. Kaashoek M, Karger D (2003) Koorde: A simple degree-optimal distributed hash table. Proc of Second International Workshop on Peer-to-Peer Systems, (IPTPS'03), LNCS 2735 pp 98–107, doi:10.1007/978-3-540-45172-3_9
13. Kleinberg J (2000) The Small-World phenomenon: an algorithmic perspective. Proceedings of the thirty-second annual ACM symposium on Theory of computing pp 163–170, doi:10.1145/335305.335325
14. Kubiatowicz J, Bindel D, Chen Y, Czerwinski S, Eaton P, Geels D, Gummadi R, Rhea S, Weatherspoon H, Weimer W, Wells C, Zhao B (2000) Oceanstore: an architecture for global-scale persistent storage. ACM SIGPLAN Not 35(11):190–201. doi:10.1145/356989.357007
15. Kuhn F, Schmid S, Wattenhofer R (2010) Towards worst-case churn resistant peer-to-peer systems. Distrib. Comput 22(4):249–267. doi:10.1007/s00446-010-0099-z
16. Lesniewski-Laas C, Kaashoek MF (2010) Whanau: A Sybil-proof Distributed Hash Table. Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation NSDI'10 8–8
17. Loguinov D, Kumar A, Rai V, Ganesh S (2003) Graph-theoretic Analysis of Structured Peer-to-Peer Systems: Routing Distances and Fault Resilience. Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications (SIGCOMM '03) pp 395–406, doi:10.1145/863955.863999
18. Malkhi D, Naor M, Ratajczak D (2002) Viceroy: a scalable and dynamic emulation of the butterfly. Proc of the 21st ACM Annual Symposium on Principles of Distributed Computing (PODC '02) pp 183–192, 10.1145/571825.57 1857
19. Manku GS, Bawa M, Raghavan P (2003) Symphony: Distributed hashing in a small world. Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems USITS'03 4:10–10
20. Manku GS, Naor M, Wieder U (2004) Know Thy Neighbor's Neighbor: The Power of Lookahead in Randomized P2P Networks. Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing STOC'04 pp 54–63, doi:10.1145/1007352.1007368. ISBN: 1-58113-852-0
21. Marczewski E, Steinhaus H (1957) On a certain distance of sets and the corresponding distance of functions. Collect. Math 6(1):319–327. ISSN 0010–1354
22. Maymounkov P, Mazières D (2002) Kademlia: A Peer-to-peer Information System Based on the XOR Metric. Proc of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02) pp 53–65, doi:10.1007/3-540-45748-8_5
23. Naor M, Wieder U (2003) Novel Architectures for P2P Applications: The Continuous-discrete Approach. Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA'03 pp 50–59, doi:10.1145/777412.777421
24. Olszak A (2010) HyCube: A DHT routing system based on a hierarchical hypercube geometry. Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics (PPAM 2009), Part II, LNCS 6068 pp 260–269, doi:10.1007/978-3-642-14403-5_28
25. Park C, Kim K, Lee J, Kim P (2007) Improved CAN routing using additional neighbors. The 9th International Conference on Advanced Communication Technology 2:1457–1461, doi:10.1109/ICACT.2007.358630
26. Pirrò G, Talia D, Trunfio P (2012) A DHT-Based semantic overlay network for service discovery. Futur. Gener. Comput. Syst 28(4):689–707. doi:10.1016/j.future.2011.11.007
27. Plaxton C, Rajaraman R, Richa A (1999) Accessing Nearby Copies of Replicated Objects in a Distributed Environment. Theory of Computing Systems pp 241–280, doi:10.1007/s002240000118
28. Ratnasamy S, Francis P, Handley M, Karp R, Schenker S (2001) A scalable Content-Addressable network. Proc of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01) pp 161–172, doi:10.1145/383059.383072
29. Rhea S, Geels D, Roscoe T, Kubiatowicz J (2004) Handling churn in a DHT. Proceedings of the USENIX 2004 Annual Technical Conference 127–140
30. Rowstron A, Druschel P (2001) Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. Proc of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), LNCS 2218 pp 329–350, doi:10.1007/3-540-45518-3_18
31. Shen H, Xu CZ (2012) Leveraging a compound graph-based DHT for multi-attribute range queries with performance analysis. IEEE Trans. Comput 61(4):433–447. doi:10.1109/TC.2011.30
32. Singh K, Schulzrinne H (2004) Peer-to-peer internet telephony using SIP. Tech. Report CUCS-044-04, Department of Computer Science. Columbia University Press, New York
33. Stoica I, Morris R, Karger D, Kaashoek M, Balakrishnan H (2001) Chord: A scalable Peer-to-peer lookup service for internet applications. Proc of the ACM SIGCOMM 2001 Technical Conference pp 149–160, doi:10.1145/383059.383071
34. Trunfio P, Talia D, Papadakis C, Fragopoulou P, Mordacchini M, Pennanen M, Popov K, Vlassov V, Haridi S (2007) Peer-to-Peer Resource Discovery In Grids: Models and systems. Futur. Gener. Comput. Syst 23(7):864–878. doi:10.1016/j.future.2006.12.003
35. Xu Z, Zhang Z (2001) Building Low-maintenance Expressways for P2P Systems. HPL-2002-41 Technical Report
36. Zhao BY, Huang L, Stribling J, Rhea SC, Joseph AD, Kubiatowicz JD (2004) Tapestry: A resilient global-scale overlay for service deployment. IEEE J. Sel. Areas Commun 22(1):41–53. doi:10.1109/JSAC.2003.818784
37. Zhu Y, Yang X (2006) Implications of Neighbor Selection on DHT Overlays. Proc of the 14th IEEE International Symposium on Modeling. Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '06) pp 197–206, doi:10.1109/MASCOTS.2006.27

Artur Olszak received the M.Sc. degree in Computer Science from Warsaw University of Technology (The Faculty of Electronics and Information Technology) in 2008, and the Ph.D. degree in Computer Science at the Institute of Computer Science, Warsaw University of Technology in 2015. His research concentrates on parallel and distributed computing architectures, environments, tools and algorithms, especially on large-scale distributed architectures.