Available online at www.sciencedirect.com
ScienceDirect
Computer Communications 31 (2008) 280–291
computer communications
www.elsevier.com/locate/comcom

Range queries on structured overlay networks

Thorsten Schütt *, Florian Schintke, Alexander Reinefeld

Zuse Institute Berlin, Takustraße 7, 14195 Berlin-Dahlem, Germany

Available online 31 August 2007

Abstract

The efficient handling of range queries in peer-to-peer systems is still an open issue. Several approaches exist, but their lookup schemes are either too expensive (space-filling curves) or their queries lack expressiveness (topology-driven data distribution).

We present two structured overlay networks that support arbitrary range queries. The first one, named Chord#, has been derived from Chord by substituting Chord's hashing function by a key-order preserving function. It has a logarithmic routing performance and it supports range queries, which is not possible with Chord. Its O(1) pointer update algorithm can be applied to any peer-to-peer routing protocol with exponentially increasing pointers. We present a formal proof of the logarithmic routing performance and show empirical results that demonstrate the superiority of Chord# over Chord in systems with high churn rates.

We then extend our routing scheme to multiple dimensions, resulting in SONAR, a *Structured Overlay Network with Arbitrary Range queries*. SONAR covers multi-dimensional data spaces and, in contrast to other approaches, SONAR's range queries are not restricted to rectangular shapes but may have arbitrary shapes. Empirical results with a data set of two million objects show the logarithmic routing performance in a geospatial domain.

Keywords: Structured overlay networks; Routing in P2P networks; Consistent hashing; Multi-dimensional range queries

1. Introduction

Efficient data lookup is at the heart of peer-to-peer computing: given a key, find the node that stores the associated object. Many structured overlay protocols like CAN [23], Chord [30], Kademlia [20], Pastry [26], P-Grid [1], or Tapestry [31] use *consistent hashing* [16] to store (*key*, *value*)-pairs in a distributed hash table (DHT). The hashing distributes the keys uniformly. A data lookup needs $O(\log N)$ communication hops with high probability in networks of N nodes.

The basis of DHTs [2] is the *key space* which defines the names of items storable in the DHT, e.g. bitstrings of length 160. A partitioning scheme splits the ownership of the key space uniformly among participating nodes, which form a ring structure by maintaining links to neighboring nodes in the *node space*. The ring structure allows to find the owner of any given key.

A typical use of this system for storing and retrieving files might work as follows. Suppose the key space contains keys in the range $[0, 2^{160})$. To store a file with a given filename in the DHT, the SHA1 hash of the filename is computed, producing a 160-bit *key*, and a message `put(key,value)` is sent to an arbitrary participating node, where *value* is the content of the file. The message is forwarded from node to node through the overlay network until it reaches the node responsible for *key* as specified by the key space partitioning, where the pair $(key, value)$ is stored. Any other client can now retrieve the contents of the file by again hashing its filename to produce *key* and asking any node to find the value associated with *key* with a message `get(key)`. The message will again be routed through the overlay to the node responsible for *key*, which will reply with the stored $(key, value)$ pair.

The described hashing uniformly distributes skewed data items over nodes and thereby supports the logarithmic

* Corresponding author. Tel.: +49 3084185361.
E-mail addresses: schuett@zib.de (T. Schütt), schintke@zib.de (F. Schintke), reinefeld@zib.de (A. Reinefeld).

0140-3664/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.comcom.2007.08.027

routing, it also incurs a major drawback: none of the mentioned P2P protocols is able to handle queries with partial keywords, wildcards, or ranges, because the hashing spreads lexicographically adjacent identifiers over all nodes.

1.1. Schemes for range queries on structured overlays

Two different approaches for range queries on structured overlays have been proposed in the literature: space-filling curves on DHTs and key-order preserving mapping functions for various network topologies.

The approaches based on space-filling curves [5,10,14] incur higher maintenance costs, because they are built on top of a DHT and therefore require one additional mapping step. Even worse, space-filling curves cover the key-space by discrete non-overlapping patches, which makes it difficult to support large multi-dimensional range queries covering several patches in a single lookup. Depending on the patch size, such queries require several independent lookups (Ref. Fig. 4).

The second type of structured overlays map adjacent key ranges to contiguous ranges in the node space. These methods are key-order-preserving and therefore capable of supporting arbitrary range queries. Since the key distribution is not known *a priori*, one approach, Mercury [9], approximates a density function of the keys to ensure logarithmic routing. The associated maintenance and storage overhead is not negligible, despite a mediocre accuracy. MURK [14] goes one step further. It is similar to our approach in that it also splits the data space into hypercuboids that are managed by separate nodes. But in contrast to our approach MURK uses a heuristic based on skip graphs whereas our SONAR builds on an extension of Chord#'s ring topology. More detailed information on related work can be found in Section 4.

1.2. Structured overlays without consistent hashing

In this paper, we argue that it is neither necessary nor beneficial to use DHTs in structured overlays. We first introduce a scheme that matches the key space directly to the node space and is thereby able to support range queries. Thereafter we generalize the scheme to multi-dimensional data spaces.

1.2.1. Chord#

Section 2 presents Chord#. It has been derived from Chord by eliminating the hashing function and introducing an explicit load balancing mechanism. Chord# has the same maintenance cost as Chord, but is superior in a number of aspects: it has a proven upper-bound lookup cost of $\log_b N$ for arbitrary b, it has a constant-time finger update algorithm and it supports range queries. It does so by routing in the node space rather than the key space. We describe the finger placement algorithm (Section 2.1), prove its logarithmic lookup performance (Section 2.1.2 and Appendix A), and demonstrate its improvements over Chord in highly dynamical systems (Section 2.3) with simulations.

1.2.2. SONAR

Section 3 extends the principle idea of Chord# to multiple dimensions. The resulting algorithm, SONAR, performs efficient multi-attribute queries. While other systems [9,14] also support multi-dimensional range queries, to the best of our knowledge there exist no other approach that is equally deterministic in its finger placement. We present details on SONAR's finger placement (Section 3.2), routing strategy (Section 3.4), range query execution (Section 3.5), and demonstrate its practical lookup performance in networks storing two million objects (Section 3.6).

Section 4 discusses related work and Section 5 presents a summary and conclusion.

2. Chord#

Since its introduction in the year 2001, Chord [30] was deployed in many practical P2P systems, making it one of the best-understood overlay protocols. Chord's lookup mechanism is provably robust in the face of node failures and re-joins. It uses consistent hashing to map the keys and node IDs uniformly onto the identifier space. In the following, we introduce our Chord# algorithm by deriving it step by step from Chord as illustrated in the four parts of Fig. 1.

Let us assume a key space of size 2^8 and a node space of size 2^4. Chord organizes the nodes $n_0, \ldots, n_{15}$ in a logical ring, each of them being responsible for a subset of the keys $0, \ldots, 2^8 - 1$. Each node maintains a *finger table* that contains the addresses of the peers halfway, quarter-way, 1/8-way, 1/16-way, ..., around the ring. When a node (e.g. n_0) receives a query, it forwards it to the node in its finger table with the highest ID not exceeding `hash(key)`. This halves the distance in each step, resulting in $O(\log N)$ hops in networks with N nodes, because the hashing ensures a uniform distribution of the keys and nodes with a high probability [30] (Fig. 1a).

Because this scheme does not support range queries, we eliminate the hashing of the keys. All keys are now sorted in lexicographical order, but unfortunately the nodes that are responsible for popular keys (e.g. 'E') will become overloaded – both in terms of storage space and query load. Hence, this approach is impractical, even though its routing performance is still logarithmic (Fig. 1b).

When we additionally eliminate the hashing of the *node IDs*, the nodes can be placed at any suitable place in the ring to achieve a better load distribution. We must introduce an explicit load balancing scheme [17] that dynamically removes keys from overloaded nodes. Unfortunately, without adjusting the fingers in the finger table, much more hops are needed to retrieve a given key. The

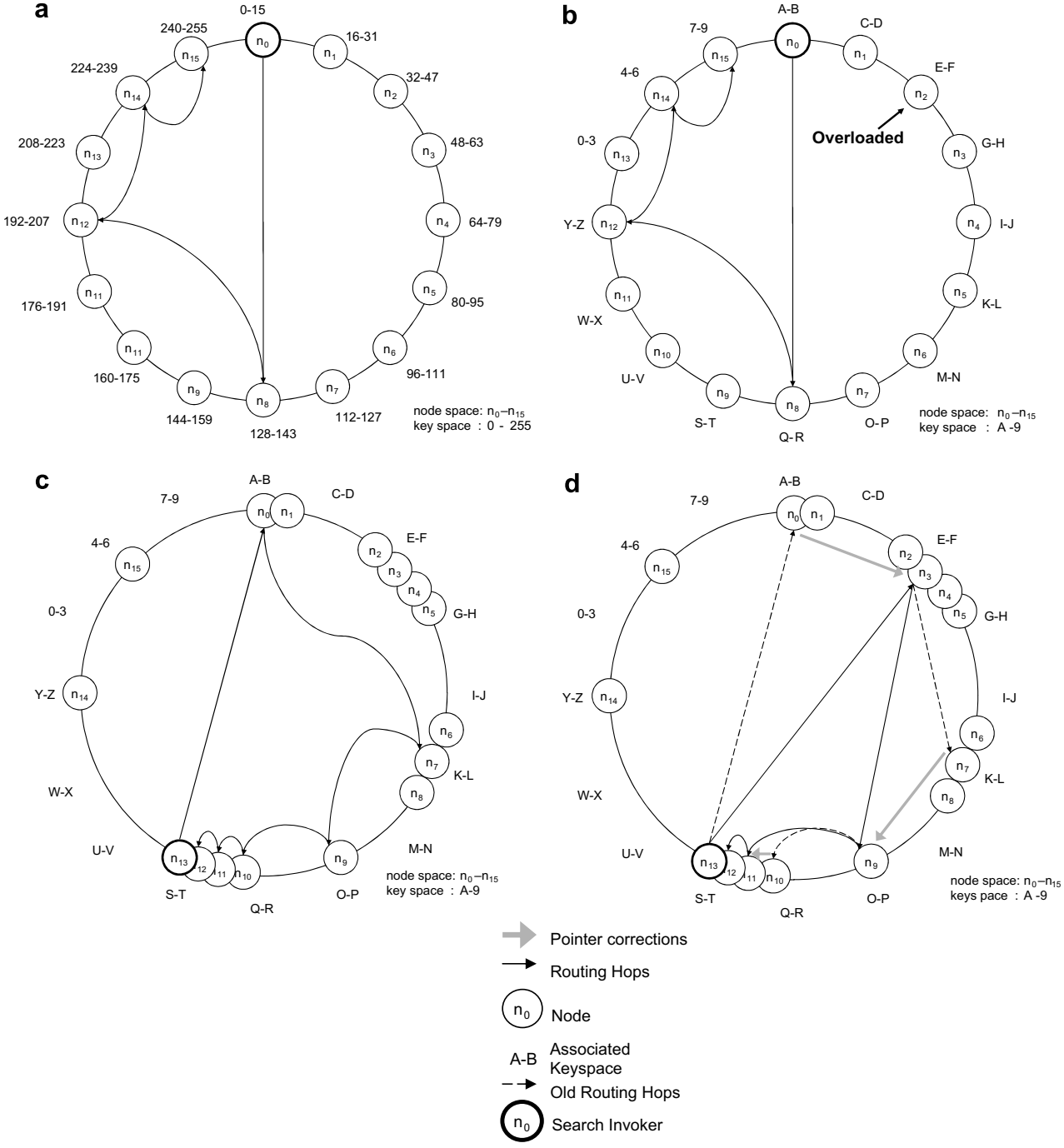


Fig. 1. Transforming Chord into Chord$^{\#}$.

lookup started in node n_{13} for key 'R', for example, needs six instead of four hops. In general, the routing degrades to O(N) (Fig. 1c).

In the final step, we introduce a new finger placement algorithm that dynamically adjusts the fingers in the routing table. The lookup performance is now again O($\log N$) – just as in Chord. But in contrast to Chord this new variant does the routing in the node space rather than the key space, and it supports complex queries – all with logarithmic routing effort (Fig. 1d).

We now present the new finger placement algorithm and discuss its logarithmic performance. A formal proof can be found in Appendix A.

2.1. Finger placement

In the above transformation we substituted Chord's hash function by a key-order preserving function. When doing so, the keys are no longer uniformly distributed over the nodes but they follow some unknown density function.

To still obtain logarithmic routing, we must ensure that the fingers in the routing tables cross an exponential number of *nodes* in the ring. This can be achieved as follows: to calculate the longest finger (i.e., the finger pointing half-way around the ring) we ask the node referred to by our second longest entry (i.e., quarter-way around) for its second longest entry (again quarter-way). For calculating our second longest finger, we follow our third-longest finger, and so on.

In general, to calculate the ith finger in its finger table, a node asks the remote node, to which its $(i-1)$th finger refers to, for its $(i-1)$th finger. The fingers at level i are set to the fingers' pointers in the next lower level $i-1$. At the lowest level, the finger refers to the direct successor.

$$\text{finger}_i = \begin{cases} \text{successor} & : i = 0 \\ \text{finger}_{i-1} \rightarrow \text{getFinger}(i-1) & : i \neq 0 \end{cases}$$

2.1.1. Logarithmic finger table size

The finger table of Chord$^{\#}$ has a maximum of $\lceil \log N \rceil$ entries. This can be seen by observing its construction. Chord$^{\#}$ first enters the shortest finger (direct successor of the node) and recursively doubles the distance in the node space with each further entry. Formally: a node n inserts an additional finger_i into its routing table as long as the following equations holds true:

$$\text{finger}_{i-1} < \text{finger}_i < n$$

This construction process guarantees that – in contrast to Chord – no two entries point to the same node. Since the fingers in the table point to nodes at exponentially increasing distance, it becomes apparent that the routing table has a total of $\lceil \log N \rceil$ entries.

2.1.2. Logarithmic routing performance

Like the original Chord, Chord$^{\#}$ has a routing performance of $O(\log N)$ hops. Unlike Chord, the logarithmic routing performance is not only proven 'with high probability', but it constitutes a guaranteed upper bound. In the following, we outline the basic idea; the formal proof can be found in Appendix A.

Let the key space be $0 \ldots 2^{m-1}$ and let $\oplus$ be an addition modulo 2^m. In the original Chord, the ith finger (finger_i) in the routing table of node n refers to the node responsible for the key f_i with

$$f_i = (n.key \oplus 2^{i-1}) \quad \text{for} \quad 1 \leqslant i \leqslant m$$

The original Chord calculates finger_i by sending a query to the node responsible for f_i. This requires $O(\log N)$ communication hops for each single entry in the routing table, which sums up to $O(\log^2 N)$ hops per routing table.

Chord$^{\#}$'s finger placement algorithm is derived by reformulating the above equation as

$$f_i = (n.key \oplus 2^{i-2}) \oplus 2^{i-2}$$

Having split the right hand side into two terms, the recursive structure becomes apparent and it is clear that the whole calculation can be done in just 1 hop. The first term represents the $(i-1)$th finger and the second term the $(i-1)$th finger on the node pointed to by finger_{i-1}.

Routing in the node space allows us to remove the hashing function and to arrange the keys in lexicographical order among the nodes so that no node is overloaded. This new finger placement has two advantages over Chord's algorithm: first, it works with any type of keys as long as a total order over the keys exists, and second, finger updates are cheaper than in Chord, because they need just one hop instead of a full search. This is because Chord$^{\#}$ uses the better informed remote information for adjusting the fingers in its finger table by recursive finger references.

2.2. Fewer hops with $\log_b$ routing

The routing performance of Chord$^{\#}$ can be further improved at the cost of additional storage space. The idea for this enhancement comes from DKS [3], which inserts extra fingers into the routing table to allow for higher-order search schemes than simple binary search. By this means, the $\log_2 N$ routing performance can be reduced to $\log_b N$ with arbitrary bases b.

In Chord$^{\#}$, the longest finger, finger_{m-1}, with $m = \lceil \log N \rceil$, and the position of the current node n split the ring into two intervals: $[n, \text{finger}_{m-1}]$ and $[\text{finger}_{m-1}, n]$. The intervals have about equal sizes because of the recursive finger update algorithm. The next shorter finger, finger_{m-2}, splits the first half again into two halves $[n, \text{finger}_{m-2}]$ and $[\text{finger}_{m-2}, \text{finger}_{m-1}]$. This splitting is recursively continued until the subsets contain only one key. It becomes obvious that each routing hop cuts the distance to the goal in half, resulting in $O(\log_2 N)$ hops.

By dividing the interval on the ring into b equally sized subsets at each level, each hop reduces the distance to $\frac{1}{b}$th and the overall number of hops per search to $O(\log_b N)$. To implement $\log_b$ routing, we need to extend the routing table by $b-2$ extra columns. The calculation of the fingers is similar to the finger placement algorithm presented in Section 2.1.

Note that with $\log_b$ routing ($b > 2$) there are several alternatives to calculate the long fingers via different intermediate nodes. This allows to eliminate inconsistent fingers in dynamic systems based on local information. Moreover, the update process may be improved by piggybacking information on routing table entries when sending search results. This on-thy-fly correction has only negligible traffic overhead and allows more frequent updates.

2.3. Experimental results

This section presents empirical results of Chord$^{\#}$ in a highly dynamic network. To allow the comparison to other P2P protocols, we followed the experimental set up of Li et al. [19] who simulated a network of 1024 nodes under

heavy churn. The network runs for 6 h and each node fails on average (exponentially distributed) after 1 h with an absent time of 1 h (again exponentially distributed). Hence, only 50% of the nodes are online at any moment. Each alive node issues every 10 minutes a lookup query for a randomly chosen key, where the time intervals are exponentially distributed. Messages have a length of 20 bytes plus 4 bytes for each additional node address contained in the message. The latency between the nodes is given by the King data set [15] which contains real data observed at Internet DNS servers.

Note that the simulation does not account for user data. Only the protocol overhead is measured, and hence the result gives the worst case.

We used the same parameters for testing the algorithm's performance as Li et al. [19], resulting in a total of 480 simulation runs:

(i) *Base* is the branching factor of each finger table entry. Each finger table contains a total of $(\text{base}-1)*\log_{\text{base}}(n)$ fingers. *Values: 2, 8, 16, 32*.
(ii) *Successors* is the number of direct successors stored in each nodes' successor list. *Values: 4, 8, 16, 32*.
(iii) *Successor stabilization interval* denotes the time spent between two updates of the nodes' successor lists. *Values: 30 s, 60 s, 90 s*.
(iv) *Finger update interval* is the time spent between two finger table updates. *Values: 60 s, 300 s, 600 s, 900 s, 1200 s*.
(v) *Latency optimizer* tells whether proximity routing was used for improving the latency. *Values: true, false*.

Fig. 2 shows the average lookup latency versus the maintenance overhead (measured in bytes per node per second). For simplicity, we plotted just the convex hull of the parameter combinations – all inferior data points above the graphs are omitted (more detailed results can be found in [28]). As can be seen, Chord# (dotted graph) strictly outperforms Chord: it has a lower latency with reduced maintenance cost. Much of this favorable performance is attributed to the better finger placement algorithm.

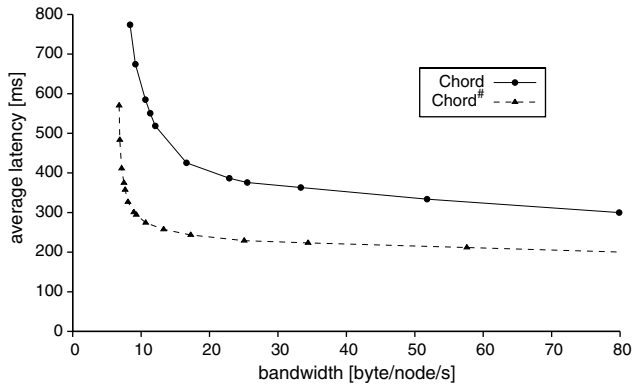


Fig. 2. The convex hull of 480 (resp. 960) experiments with different parameter combinations: Chord# outperforms Chord under churn.

Table 1 lists the number of hops for searching random keys in a network of 1024 nodes. As expected Chord# never needs more than $\log 1024 = 10$ hops while the original Chord requires up to 41 hops. This is because Chord computes the finger placement in the key space, which does not ensure that the distance in the *node space* is halved with each hop.

In summary, the experimental results indicate that Chord# has a lower maintenance overhead (Fig. 2) and requires often fewer lookup hops (Table 1) as compared to Chord. This is true for both, static and dynamic networks. Most of these favorable results are due to Chord#'s recursive finger update algorithm which uses O(1) instead of O(log *N*) communication hops to calculate a single finger entry in the routing table. Because of the recursive construction process, one could expect that the longer fingers are more likely to be misplaced – especially in networks with high churn rates. But our experimental results show the contrary: the recursive finger placement is beneficial even under a very high churn.

Table 1
Number of hops to find random keys in 1024 nodes

Hops	Chord	Chord#
0	27	44
1	549	320
2	1924	1433
3	3734	3342
4	4932	4783
5	4377	4930
6	2635	3212
7	1116	1365
8	317	315
9	78	29
10	9	3
12	9	–
13	5	–
14	4	–
15	1	–
16	7	–
17	6	–
18	7	–
19	6	–
20	4	–
21	3	–
22	2	–
23	3	–
24	1	–
25	2	–
26	2	–
27	4	–
28	3	–
29	3	–
30	1	–
31	1	–
33	1	–
37	1	–
38	1	–
41	1	–

Chord# needs a maximum of $\log(1024) = 10$ hops whereas Chord exhibits logarithmic routing performance only with 'high probability', i.e., there are some degenerated cases with considerably more hops.

3. SONAR

In this section, we extend the concept of Chord# to multiple dimensions. The resulting algorithm, named SONAR for Structured Overlay Network with Arbitrary Range Queries, is capable of handling non-rectangular range queries over multiple dimensions with a logarithmic number of routing hops. Non-rectangular range queries are important, for example, for geoinformation systems, where objects are sought that lie within a given distance from a specified position. Other applications include Internet games with thousands or millions of online-players concurrently interacting in a virtual space or grid resource management systems [12,25].

3.1. d-Dimensional data space

SONAR operates on a virtual d-dimensional Cartesian coordinate space with d attribute domains. Keys are represented by attribute vectors of length d. Like in MURK [14,21], the total key space is dynamically partitioned among the nodes such that each node is responsible for approximately the same amount of keys. Explicit load balancing allows to add or remove nodes when the number of objects increases or shrinks or when additional storage space becomes available.

Similarly to CAN [23], each SONAR node participates in d dimensions and has direct neighbors in all directions. The torus is made up of hypercuboids, each of them managed by a single node. Taken together, all hypercuboids cover the complete key space.

During runtime the system balances the key load in such a way that each hypercuboid contains about the same number of keys, and hence each node has to handle a similar amount of data. As a consequence, the hypercuboids have different sizes and a node usually has more than one direct neighbor per direction. Compared to Chord# or CAN, this slightly complicates the routing, because there are usually several options for selecting a 'direct neighbor' – we chose the one that is adjacent to the center of the node (see the small ticks in Fig. 3).

3.2. Building the routing table

Fig. 3 illustrates a routing table in a two-dimensional data space. The keys are specified by attribute vectors (x,y) and the hypercuboids (here: rectangular boxes), which are managed by the nodes, cover the complete key space. Their different area is due to the key distribution: at runtime, the load balancing scheme ensures that each box holds about the same number of keys.

SONAR maintains two data structures, a neighbor list and a routing table. The neighbor list contains links to all neighbors of a node. The node depicted by the grey box in Fig. 3, for example, has ten neighbors.

The routing table comprises d subtables, one for each dimension. Each subtable s with $1 \leqslant s \leqslant d$ contains fingers

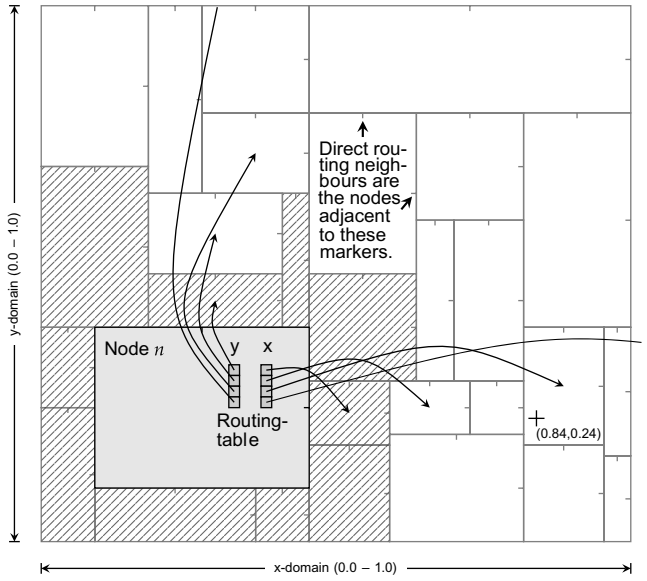


Fig. 3. Routing fingers of a SONAR node in a two-dimensional data space.

that point to remote nodes in exponentially increasing distances. We use the same recursive finger placement algorithm as in Chord# (Section 2.1.2) for inserting fingers into each subtable s of node n: starting with the neighbor that is adjacent to the center of node n in routing direction s, we insert an additional finger_i into the subtable as long as the following equations holds true:

$$\text{finger}_{i-1} < \text{finger}_i < n$$

This construction process ensures that each subtable s contains $\log N_s$ entries, where N_s is the number of nodes in dimension s. Taken together, all subtables s of a node n will have $\log N$ entries – just as in Chord#. Note that this is a self-regulating process [4]: the exact number of entries per dimension depends on the actual data distribution of the application. All that can be said, is that the total number of entries is $\log N$.

3.3. Routing performance

Assuming that the d-dimensional torus is filled uniformly by $N_1 * N_2 * \ldots * N_d = N$ nodes, the routing performance is $\log N_1 + \log N_2 + \cdots + \log N_d = \log N$. So, even though SONAR supports multi-dimensional range queries, it exhibits a similar logarithmic routing performance as Chord#. Our simulations presented in Section 3.6 confirm this observation even for highly skewed, real world data distributions.

3.4. Routing in d dimensions

Chord# uses greedy routing: it always selects the finger that is nearest to the target and forwards the request to this node. In SONAR the situation is more complex, because in

each routing step there are d dimensions to chose from. For regular grids the order of dimensions does not make a difference for the number of routing hops. Here it is best to choose the dimension for the next hop based on network latency, called proximity routing. When the grid is irregular (as in Fig. 3), it is important to select the 'best' dimension in each routing hop – either by distance (greedy routing), by the volume of the managed data space, or simply at random.

3.5. Non-rectangular range queries

The one-dimensional range queries supported by Chord# are defined by a lower and an upper bound: the query returns all keys between those bounds. Extending this approach to d dimensions results in *rectangular* range queries. In contrast to other approaches, SONAR also efficiently supports *non-rectangular* range queries.

Fig. 5 shows a circular range query in two dimensions, defined by a center and a radius. In this example, we assume that a person in the governmental district of Berlin is searching for a hotel. The center of the circle is the location of the person and the radius is chosen to find hotels in 'walking distance'. In a first step, SONAR routes the query to the node responsible for the center of the circle, taking $O(\log N)$ hops. Thereafter, the query is broadcasted to the neighbors which partially cover the circle, which is proportional to the size of the range query (a single hop for each neighbor). The query is performed on the local data and the results are returned to the requesting node.

Other structured overlays, which support multi-dimensional range queries, like [5,14,27,29], use space filling curves to map the multi-dimensional space to one dimension. As shown in Fig. 4 z-curves can be used for this. Different parts of the z-curve are assigned to different nodes. Even for simple range queries several non-contiguous parts of the curve may be responsible for the queried range. For each part a complete lookup with $O(\log N)$ hops is necessary (eight in the given example).

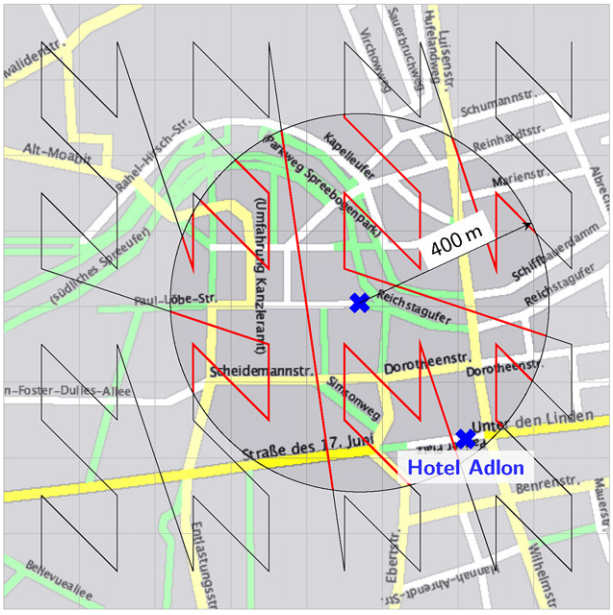


Fig. 4. 2-Dimensional range query using z-curves.

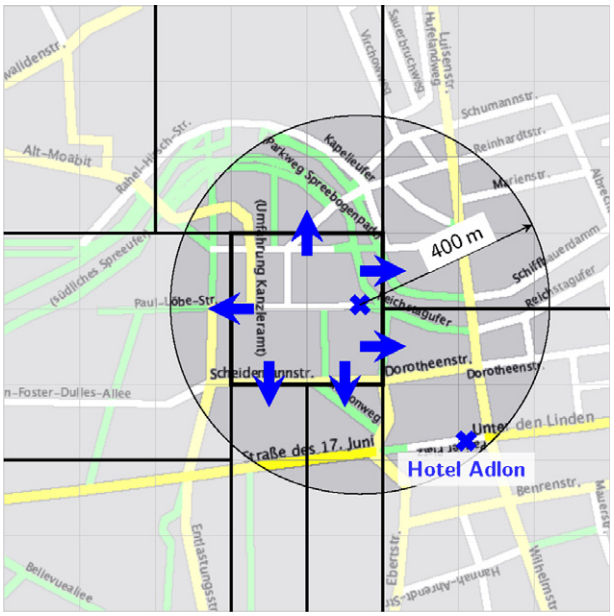


Fig. 5. 2-Dimensional range query by SONAR.

3.6. Performance results

For our experiments we selected a large data set of a traveling salesman problem with the 1,904,711 largest cities worldwide [6]. Their GPS locations follow a Zipf distribution [32], which is a common distribution pattern of many other application domains. In a preprocessing step we partitioned the globe into non-overlapping rectangular patches so that each patch contains about the same amount of cities. We did this by recursively splitting the patches along alternate sides until the number of cities in the area dropped below a given threshold. We mapped the coordinates onto a doughnut-shaped torus (Fig. 6) rather than a sphere, because the poles of a sphere would become a routing bottleneck and the rings for the western, respectively, eastern hemisphere (southwards vs. northwards) would be in opposite directions.

Fig. 7 gives an overview of the results for networks of 128–131,072 nodes. It shows the routing performance (average hop number) and the routing table size in x- and y-direction. As expected, all data points increase logarithmically with the network size.

Most interesting are the results on the routing table sizes: although the nodes autonomously determine the table sizes solely on their local knowledge, the result perfectly matches the theoretical expectation of $\log_2 N$ entries. In the two-dimensional case (depicted here), each routing table contains one subtable in x-direction, which is a bit

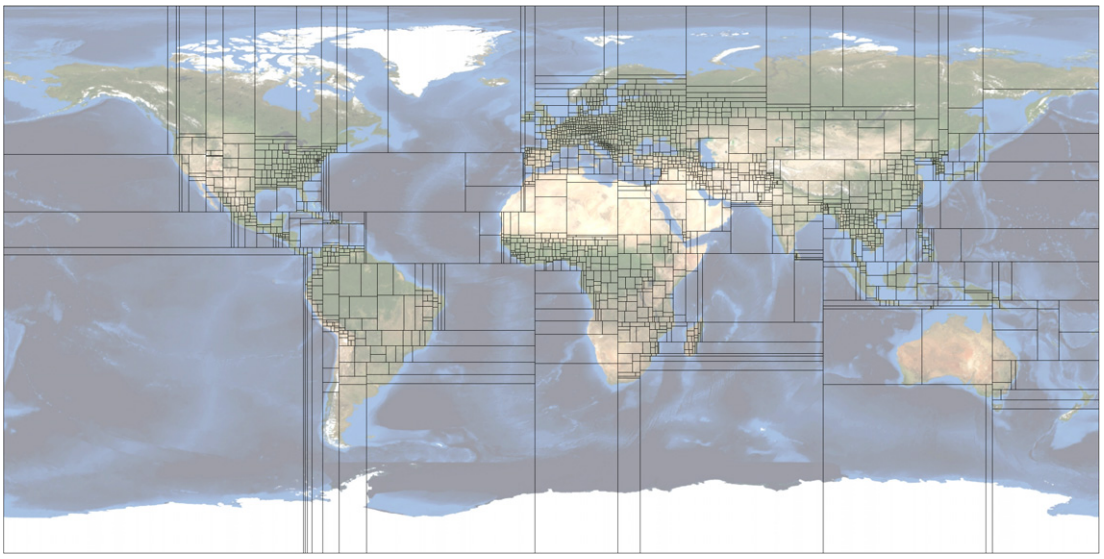


Fig. 6. SONAR overlay network with 1.9 million keys (city coordinates) over 2048 nodes. Each rectangle represents one node.

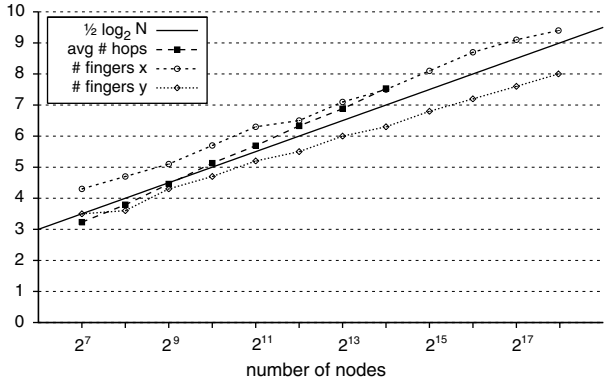


Fig. 7. Lookup performance (avg. #hops) and size of routing tables (#fingers) for various network sizes.

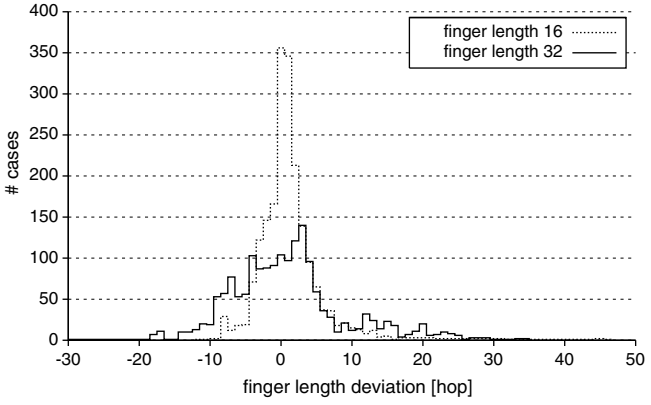


Fig. 8. Deviation of finger lengths due to local knowledge in a torus of 1024 nodes. The measured lengths are centered around their expected length of 16 resp. 32.

larger, and one subtable in y-direction, which is a bit smaller. The deviations are due to the given key distribution of this domain. Together, both subtables contain $\log_2 N$ entries.

Fig. 7 also shows the average number of routing hops. The slope of this graph is slightly above the expected value ($0.5\log_2 N$) because the routing table entries are calculated on local knowledge only, which may result in longer lookup chains. To verify this hypothesis, we checked the accuracy of the finger lengths. Fig. 8 shows the length deviations for fingers of length 16 and 32 in a torus of 2048 nodes. This data was obtained by comparing the entries in the routing table to the actual distance computed with global knowledge (based on a breadth-first search in all directions along the neighbors). The results seem to indicate that the longer fingers of size 32 (straight line) more often overestimate the actual length.

The observed inaccuracy of the finger lengths may also be attributed to the specific data distribution in our experiment [24]. From Fig. 6 it is evident that the rectangles of the whole data space have very different sizes. This becomes more obvious when plotting the number of rectangles versus their size: Fig. 9 shows an almost perfect Zipf distribution, which is common for a large number of applications [11].

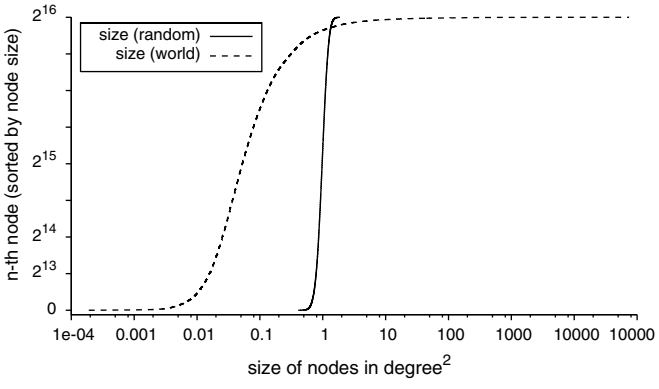


Fig. 9. Distribution of node sizes.

Assuming that large rectangles have more incoming and outgoing fingers than the smaller ones, we plotted the in-degree of all nodes, see Fig. 10. If all keys would be uniformly distributed over the key space, and consequently all nodes would be of the same size, one would expect an in-degree of $\log_2 2048 = 11$ for each node. Fig. 10 indicates that the in-degree is slightly smaller for most nodes, because there are a few (presumably large) nodes with an extremely high in-degree of up to 100.

Due to the different node sizes the rings of the torus are skewed, i.e., they do not end in the beginning node, but may form spirals or may even join in a common node.[1] For the routing process, this does not cause harm, because a data lookup never makes a full round. Only for performance reasons we need to address the problem of uneven in-degrees: some nodes may have to handle more routing requests than others which may affect the routing performance. One solution to this problem would be to include the in-degree into the load metric. The load balancing scheme would then autonomously balance the in-degrees by splitting larger nodes and thereby reducing the in-degree. Alternatively, we could provide more flexibility in the selection of neighbors by allowing also neighbors that are not adjacent to the center of the node for the routing.

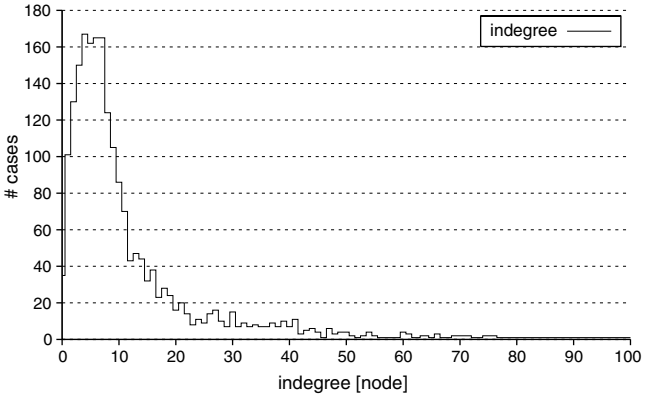


Fig. 10. In-degree of nodes in a torus with 2048 nodes. The expected value is $\log_2 2048 = 11$.

4. Related work

The first structured overlay networks like Chord [30] or CAN [23], published in 2001, allow to organize unreliable peers into a stable, regular structure, where each node is responsible for a part (ring segment in Chord, rectangle in CAN). Due to consistent hashing these system allow to distribute the nodes and keys equally along the system and to route with $O(\log N)$ resp. $O(\sqrt[d]{N})$ performance. Both handle one-dimensional keys but do not support efficient range queries, because adjacent keys are not mapped to adjacent nodes in the overlay.

4.1. One-dimensional range queries

To allow efficient range queries, structured overlays without hashing had to be developed, which put adjacent keys to nodes adjacent in the overlay. So, with one logarithmic lookup for the start of the range, the range query can be performed by that node and the nodes adjacent in the logical structure of the overlay. One major challenge for such systems is the uneven distribution of keys to nodes and the finger maintenance for efficient routing. SkipGraphs [7], a distributed implementation of skip lists [22], for example, use probabilistically balanced trees and thereby allow efficient range queries with $O(\log N)$ performance.

Ganesan et al. [13] further improved SkipGraphs with an emphasis on load-balancing. Their load-balancing scheme maps ordered sets of keys on nodes with formally proven performance bounds, similar to [17]. For routing, they deploy a SkipGraph on top of the nodes, guaranteeing the routing performance of $O(\log N)$ with high probability.

Mercury [9] does not use consistent hashing and therefore has to deal with load imbalance. It determines the density function (see Section A) with random walk sampling, which generates additional traffic for maintaining the finger table.

In contrast to Mercury, Chord$^{\#}$ [28] never needs to compute the density function and therefore has significantly less overhead. It is the recursive construction of the routing table in the node space using the successor information, which allows Chord$^{\#}$ to handle load-imbalance while preserving an $O(\log N)$ routing performance.

4.2. Multi-dimensional keys and range queries

There exist several systems, that support multi-dimensional keys *and* range queries. They can be split into two groups:

4.2.1. Space filling curves

Several systems [5,14,27] have been proposed that use space-filling curves to map multi-dimensional to one-dimensional keys. Space-filling curves are locality preserving, but they provide less efficient range queries than the space partitioning schemes described below. This is because a single range query may cover several parts which have to be queried separately (Fig. 4).

Chawathe et al. [10] implemented a 2-dimensional range query system on top of OpenDHT using z-curves for linearization. In contrast to many other publications, they report performance results from a real-world application. Due to the layered structure the query latency is only 2–3 s for 24–30 nodes.

4.2.2. Space partitioning schemes

Another approach is the partitioning and direct mapping of the key space to the nodes. SONAR belongs to this

[1] In fact, the overlay built by SONAR is only a torus when the keys are uniformly distributed. In all practical instances, SONAR builds a graph that only slightly resembles a torus.

group of systems. The main differentiating factor between such systems is the indexing and routing structure.

SWAM [8] employs a Voronoi-based space partitioning scheme and uses a small-world graph overlay [18] with routing tables of size O(1). The overlay does not rely on a regular partitioning like a kd-tree, but it must sample the network to place its fingers.

Multi-attribute range queries were also addressed by Mercury [9], but their implementation uses a large number of replicas per item to achieve logarithmic routing performance. Following this scheme, we could employ Chord# for the same purpose, but SONAR supports multi-dimensional range queries with considerably less storage overhead.

Ganesan et al. [14] proposed two systems for multi-dimensional range queries in P2P systems – SCRAP and MURK. SCRAP follows the traditional approach of using space-filling curves to map multi-dimensional data down to one dimension. Each range-query can then be mapped to several range queries on the one dimensional mapping.

MURK is more similar to our approach as it also divides the data space into hypercuboids with each partition assigned to one node. In contrast to SONAR, MURK implements a heuristic approach based on skip graphs.

5. Conclusion

We presented two structured overlay protocols that do not use consistent hashing and are therefore able to support range queries.

Our Chord# provides a richer query expressiveness with the same logarithmic routing complexity as Chord. Its finger update algorithm needs just one communication hop per routing table entry instead of $O(\log N)$ hops as in Chord. As shown in our experimental results (Fig. 2), this greatly reduces the maintenance overhead and it is beneficial in dynamic environments with a high churn rate.

The second part of the paper generalizes the concepts of Chord# to multi-dimensional data. The resulting algorithm, named SONAR for Structured Overlay Network with Arbitrary Range Queries, is capable of handling non-rectangular range queries over multiple dimensions with a logarithmic number of routing hops. Non-rectangular range queries are necessary for geo-information systems, where objects are sought that lie within a given distance from a specified position, or in Internet games with millions of online-players interacting in a virtual game space.

Acknowledgements

We thank the anonymous reviewers and the guest editors for their valuable comments. Our thanks go also to Slaven Rezić for his street map of Berlin (used in Figs. 4 and 5) and to NASA's Earth Observatory for the topographic images from the 'Blue Marble next generation' project in Fig. 6. Part of this research was supported by the EU projects SELFMAN and XtreemOS.

Appendix A. Proof of the logarithmic routing performance of Chord#

Before proving the routing performance of Chord# to be $O(\log_2 N)$, we briefly motivate our line of argumentation. Let the key space be $0 \ldots 2^{m-1}$. In Chord, the ith finger (finger_i) in the finger table of node n refers to the node responsible for the key f_i with[2]

$$f_i = (n.key \oplus 2^{i-1}) \quad \text{for} \quad 1 \leqslant i \leqslant m \tag{A.1}$$

This procedure needs $O(\log N)$ hops for each entry. It can be rewritten as

$$f_i = (n.key \oplus 2^{i-2}) \oplus 2^{i-2}$$

Having split the right hand side into two terms, the recursive structure becomes apparent and it is clear that the whole calculation can be done in just 1 hop. The first term represents the $(i-1)$th finger and the second term the $(i-1)$th finger on the node pointed to by finger $i-1$.

For proving the correctness, we describe the node distribution by the density function $d(x)$. It gives for each point x in the key space the reciprocal of the width of the corresponding interval. For a Chord ring with N nodes and a key space size of $K = 2^m$ the density function can be approximated by $d(x) = \frac{N}{2^m}$ (the reciprocal of $\frac{K}{N}$ and $K = 2^m$) because it is based on consistent hashing.

Theorem 1 (*Consistent hashing* [16]). *For any set of N nodes and K keys, with high probability*:

(i) *Each node is responsible for at most* $(1+\epsilon)\frac{K}{N}$ *keys.*
(ii) *node* $(N+1)$ *joins or leaves the network, responsibility for* $O(\frac{K}{N})$ *keys changes hands* (*and only to or from the joining or leaving node*).

The most interesting property of $d(x)$ is the integral over subsets of the key space:

Lemma 1. *The integral over* $d(x)$ *equals the number of nodes in the corresponding range. Hence, the integral over the whole key space is*:

$$\int_{\text{keyspace}} d(x)\mathrm{d}x = N.$$

Proof. We first investigate the integral of an interval from a_i to a_{i+1}, where a_i and a_{i+1} are the left and the right end of the key range owned by a single node.

$$\int_{a_i}^{a_{i+1}} d(x)\ \mathrm{d}x \stackrel{?}{=} 1.$$

Because a_i and a_{i+1} mark the begin and the end of an interval served by one node, d is constant for the whole range.

[2] We assume calculations to be done in a ring using $(\bmod 2^m)$.

The width of this interval is $a_{i+1} - a_i$ and therefore according to its definition $d(x) = \frac{1}{a_{i+1}-a_i}$. Because we chose a_i and a_{i+1} to span exactly one interval the result is 1, as expected.

The integral over the whole key space therefore equals the sum of all intervals, which is N:

$$\int_{\text{keyspace}} d(x)\ \mathrm{d}x = \sum_{i=0}^{N-1} \int_{a_i}^{a_{i+1}} d(x)\ \mathrm{d}x = N \qquad \square$$

Note that Lemma 1 could also be used to estimate the amount of nodes $\tilde{N}$ in the system, having an approximation of $d(x)$ called $\tilde{d}(x)$. Each node could compare $\frac{1}{2}\log(\tilde{N})$ to the observed average routing performance in order to estimate and improve its local approximation $\tilde{d}(x)$.

A.1. Finger placement in chord

Both, Chord and Chord$^{\#}$ use logarithmically placed fingers, so that searching is done in $O(\log N)$. Chord, in contrast to our scheme, computes the placement of its fingers in the key space. This ensures that with each hop the distance in the key space to the searched key is halved, but it does not ensure that the distance in the node space is also halved. So, a search may need more than $O(\log N)$ network hops. According to Theorem 1, the search in the node space still takes $O(\log N)$ steps *with high probability*. In regions with less than average sized intervals ($d(x) \gg \frac{N}{K}$) the routing performance degrades.

Chord places the fingers finger_i in a node n with the following scheme:

$$f_i = (n.key \oplus 2^{i-1}), \quad 1 \leqslant i \leqslant m \tag{A.2}$$

Using our integral approach from Lemma 1 and the density function $d(x)$, we develop an equivalent finger placement algorithm as follows. First, we take a look at the longest finger finger_{m-1}. It points to the node responsible for $n + 2^{m-1}$ when the key space has a size of 2^m. This corresponds to the opposite side of n in the Chord ring. With a total of N nodes this finger links to the $\frac{N}{2}$th node to the right with *high probability* due to the consistent hashing theorem.

With Lemma 1 key f_{m-1}, which is stored on the $\frac{N}{2}$th node to the right, can be predicted.

$$\int_n^{f_{m-1}} d(x)\mathrm{d}x = \frac{N}{2}$$

Other fingers to the $\frac{N}{4}$th, …, $\frac{N}{2^i}$th node are calculated accordingly.

As a result we can now formulate the following more flexible finger placement algorithm:

Theorem 2 (Chord finger placement)**.** *For Chord, the following two finger placement algorithms are equivalent*:

(i) $f_i = (n.key \oplus 2^{i-1}), \quad 1 \leqslant i \leqslant m$
(ii) $\int_n^{f_i} d(x)\,dx = \frac{2^{i-1}}{2^m}N, \quad 1 \leqslant i \leqslant m$

Proof. To prove the equivalence, we set $d(x) = \frac{N}{2^m}$ according to Theorem 1.

$$\int_n^{f_i} d(x)\,\mathrm{d}x = \frac{2^{i-1}}{2^m}N$$
$$\int_n^{f_i} \frac{N}{2^m}\,\mathrm{d}x = \frac{2^{i-1}}{2^m}N$$
$$\frac{N}{2^m}(f_i \ominus n) = \frac{2^{i-1}}{2^m}N$$
$$f_i = n.key \oplus 2^{i-1} \qquad \square$$

The equivalence of Chord's two finger placement algorithms will be used in the following section to prove the correctness of Chord$^{\#}$'s algorithm.

A.2. Finger placement in Chord$^{\#}$

Theorem 3 (*Chord$^{\#}$ finger placement*)**.**

$$\text{finger}_i = \begin{cases} \text{successor} & : i = 0 \\ \text{finger}_{i-1} \rightarrow \ \text{getFinger}(i-1) & : i \neq 0 \end{cases}$$

Proof. We first analyze Chord's finger placement (Ref. Theorem 2) in more detail.

$$\int_n^{f_i} d(x)\ \mathrm{d}x = \frac{2^{i-1}}{2^m}N, \quad 1 \leqslant i \leqslant m \tag{A.3}$$

First we split the integral into two equal parts by introducing an arbitrary point X between n (the key of the local node) and f_i (the key of finger_i):

$$\int_n^{X} d(x)\,\mathrm{d}x = \frac{2^{i-2}}{2^m}N \tag{A.4}$$
$$\int_X^{f_i} d(x)\,\mathrm{d}x = \frac{2^{i-2}}{2^m}N \tag{A.5}$$

In Eqs. (A.4) and (A.5), the only unknown is X. Comparing Eq. (A.4) to Theorem 2, we see that X is f_{i-1}.

In summary, to calculate finger_i we go to the node addressed by finger_{i-1} in our finger table (Eq. (A.4)), which crosses half of the nodes to finger_i. From this node the $(i-1)$th entry in the finger table is retrieved, which refers to finger_i according to Eq. (A.5). So, Eq. (A.3) is equivalent to

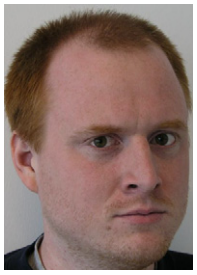
$$\text{finger}_i = \text{finger}_{i-1} \rightarrow \ \text{getFinger}(i-1)$$

Instead of approximating $d(x)$ for the whole range between n and f_i, we split the integral into two parts and treat them separately. The integral from n to f_{i-1} is equivalent to the calculation of finger_{i-1} and the remaining equation is equivalent to the calculation of the $(i-1)$th finger of the node at finger_{i-1}. We thereby proved the correctness of the pointer placement algorithm in Theorem 3. $\square$

With this new routing algorithm, the cost for updating the complete finger table has been reduced from $O(\log^2 N)$ in Chord to $O(\log N)$ in Chord$^{\#}$.

References

[1] K. Aberer, P-Grid: a self-organizing access structure for P2P information systems, CoopIS (2001).
[2] K. Aberer, L. Onana Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, M. Hauswirth, The essence of P2P: a reference architecture for overlay networks, IEEE P2P (2005).
[3] L. Alima, S. El-Ansary, P. Brand, S. Haridi, DKS(N,k,f): a family of low-communication, scalable and fault-tolerant infrastructures for P2P applications, GP2PC (2003).
[4] A. Andrzejak, A. Reinefeld, F. Schintke, T. Schtt, On adaptability in grid systems, in: V. Getov, D. Laforenza, A. Reinefeld (Eds.), Future Generation Grids, 2006, pp. 29–46.
[5] A. Andrzejak, Z. Xu, Scalable, efficient range queries for Grid information services, IEEE P2P (2002).
[6] D. Applegate, R. Bixby, V. Chvatal, W. Cook, Implementing the Dantzig-Fulkerson-Johnson algorithm for large traveling salesman problems, Mathematical Programming, Series B, 97 <http://www.tsp.gatech.edu/world>, 2003.
[7] J. Aspnes, G. Shah, Skip graphs, SODA (2003).
[8] F. Banaei-Kashani, C. Shahabi, SWAM: a family of access methods for similarity-search in peer-to-peer data networks, CIKM (2004).
[9] A. Bharambe, M. Agrawal, S. Seshan, Mercury: supporting scalable multi-attribute range queries, ACM SIGCOMM (2004).
[10] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, J. Hellerstein, A case study in building layered DHT applications, ACM SIGCOMM (2005).
[11] P.J. Denning, Network laws, CACM 47 (11) (2004).
[12] V. Gaede, O. Günther, Multidimensional access methods, ACM Computing Surveys 30 (2) (1998).
[13] P. Ganesan, M. Bawa, H. Garcia-Molina, Online balancing of range-partitioned data with applications to peer-to-peer systems, VLDB (2004).
[14] P. Ganesan, B. Yang, H. Garcia-Molina, One torus to rule them all: multidimensional queries in P2P systems, WebDB (2004).
[15] K.P. Gummadi, S. Saroiu, S.D. Gribble, King: estimating latency between arbitrary internet end hosts, in: Proceedings of the 2nd Usenix/ACM SIGCOMM Internet Measurement Workshop, 2002.
[16] D. Karger, E. Lehman, T. Leighton, R. Panigrah, M. Levine, D. Lewin, Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. 29th Annual ACM Symposium on Theory of Computing, 1997.
[17] D. Karger, M. Ruhl, Simple efficient load balancing algorithms for peer-to-peer systems, IPTPS (2004).
[18] J. Kleinberg, The small-world phenomenon: an algorithmic perspective, in: Proceedings of the 32nd ACM Symposium on Theory of Computing, 2000.
[19] J. Li, J. Stribling, R. Morris, M.F. Kaashoek, T.M. Gil, A performance vs. cost framework for evaluating DHT design tradeoffs under churn, Infocom (2005).
[20] P. Maymounkov, D. Maziéres, Kademlia: a peer-to-peer information system based on the XOR metric, IPTPS (2002).
[21] M. Naor, U. Wieder, Novel architectures for P2P Applications: the continuous-discrete approach, ACM SPAA (2003).
[22] W. Pugh, Skip lists: a probabilistic alternative to balanced trees, Communications of the ACM (1990).
[23] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A scalable content-addressable network, ACM SIGCOMM (2001).
[24] A. Reinefeld, F. Schintke, T. Schütt, P2P routing of range queries in skewed multidimensional data sets, ZIB-Report 07-23 (2007).
[25] A. Reinefeld, F. Schintke, T. Schütt, Scalable and self-optimizing data grids, in: Yuen Chung Kwong (Ed.), Annual Review of Scalable Computing, vol. 6, 2004, pp. 30–60 (Chapter 2).
[26] A. Rowstron, P. Druschel, Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems, Middleware (2001).
[27] C. Schmidt, M. Parashar, Enabling flexible queries with guarantees in P2P systems, IEEE Internet Computing 19–26 (2004).
[28] T. Schütt, F. Schintke, A. Reinefeld, Structured overlay without consistent hashing: empirical results, GP2PC (2006).
[29] Y. Shu, B. Chin Ooi, K. Tan, A. Zhou, Supporting multi-dimensional range queries in peer-to-peer systems, IEEE P2P (2005).
[30] I. Stoica, R. Morris, M.F. Kaashoek, D. Karger, H. Balakrishnan, Chord: a scalable peer-to-peer lookup service for Internet application, ACM SIGCOMM (2001).
[31] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, J. Kubiatowicz, Tapestry: a resilient global-scale overlay for service deployment, IEEE Journal on Selected Areas in Communications 22 (1) (2004).
[32] G. Zipf, Relative frequency as a determinant of phonetic change, reprinted from Harvard Studies in Classical Philiology, 1929.



Thorsten Schütt is a Ph.D. candidate with the Zuse Institute Berlin (ZIB). He got his Diploma with distinction in 2002 from the Technical University Berlin. Since then he works as a research staff member in the Computer Science Research Department at ZIB and participates in several EU projects. His research interests include distributed data management, scalable grid systems and P2P computing.

Florian Schintke is a Ph.D. candidate with the Zuse Institute Berlin (ZIB). He graduated in 2000 with distinction from the Technical University, Berlin. Since then, he works as a research staff member in the Computer Science Research Department at ZIB. He participates in several European and German research projects on Grid and Peer-to-Peer computing. He is a member of the German Computer Science Society and the IEEE Technical Committee on Scalable Computing. His research interests are in the areas of distributed data management, scalable systems and autonomic computing.

Alexander Reinefeld heads the computer science Department of the Zuse Institute Berlin and is a professor for parallel and distributed systems at the Humboldt-University Berlin. He received the Diploma degree (M.Sc) in 1982 and the Ph.D. in 1987, both from the University Hamburg. In 1984, he was awarded a Ph.D. scholarship by the DAAD and in 1987 a Sir Izaak Walton Killam Post Doctoral Fellowship of the University of Alberta in Edmonton/Canada. His research interests include grid and peer-to-peer computing, distributed data management, and algorithms for innovative HPC systems such as hardware accelerators. He organized international conferences (CCGrid2002, GGF 2004) and he participates in several editorial and advisory boards. He is a member of ACM, IEEE Computer Society and Gesellschaft fuer Informatik.