A Structured Overlay for Multi-Dimensional Range Queries*

Thorsten Schütt, Florian Schintke, and Alexander Reinefeld

Zuse Institute Berlin

Abstract. We introduce SONAR, a structured overlay to store and retrieve objects addressed by multi-dimensional names (*keys*). The overlay has the shape of a multi-dimensional torus, where each node is responsible for a contiguous part of the data space. A uniform distribution of keys on the data space is not necessary, because denser areas get assigned more nodes. To nevertheless support logarithmic routing, SONAR maintains, per dimension, fingers to other nodes, that span an exponentially increasing number of *nodes*. Most other overlays maintain such fingers in the *key-space* instead and therefore require a uniform data distribution. SONAR, in contrast, avoids hashing and is therefore able to perform range queries of arbitrary shape in a logarithmic number of routing steps—independent of the number of system- and query-dimensions.
SONAR needs just one hop for updating an entry in its routing table: A longer finger is calculated by querying the node referred to by the next shorter finger for its shorter finger. This doubles the number of spanned nodes and leads to exponentially spaced fingers.

1 Introduction

The efficient handling of multi-dimensional range queries in Internet-scale distributed systems is still an open issue. Several approaches exist, but their lookup schemes are either expensive (space-filling curves) [2] or use probabilistic approaches like consistent hashing [10] to build the overlay.

We propose a system for storing and retrieving objects with d-dimensional keys in a peer-to-peer network. SONAR (Structured Overlay Network with Arbitrary Range-queries) directly maps the multi-dimensional data space to a d-dimensional torus. It supports range queries of arbitrary shape, which are useful, for example, in geo-information systems where objects in a given distance of a position are sought. SONAR can also be employed in Internet games with millions of online-players who concurrently interact in a virtual space and need quick access to the local surroundings of their avatars. In a broader context, SONAR can be employed as a hierarchical publish/subscribe system, where published events are categorized by several independent attributes. The category of published events addresses a data point in the d-dimensional space and consumers subscribing to subareas will receive all events published in their subarea.

* Part of this work was carried out under the SELFMAN and XtreemOS projects funded by the European Commission.

The paper is organized as follows: First, we discuss related work. Then, in Section 3, we introduce SONAR. In Section 4, we present empirical results and in Section 5 we conclude the paper with a brief summary.

2 Related Work

Several systems [1] have been proposed that support complex queries with multi-dimensional keys and ranges. They can be split into two groups.

a) Space filling curves. These systems [2, 9, 16] use locality preserving space-filling curves to map multi-dimensional to one-dimensional keys. They provide less efficient range queries than the space partitioning schemes described below, because a single range query may cover several parts of the curve, which have to be queried separately (Fig. 5a). Chawathe et al. [7] present performance results of a real-world application using Z-curves on top of OpenDHT. The query performance ($\approx$ 2 sec. for $\leq$ 30 nodes) is rather low due to the layered approach.

b) Space partitioning. The schemes using space partitioning split the key-space among the nodes. SONAR belongs to this group of systems. The proposed systems mainly differ by their routing strategies.

CAN [14] was one of the very first DHTs. It hashes the key-space onto a multi-dimensional torus. While the topology resembles that of SONAR and MURK (see below), CAN uses just the neighbors for routing and it does not support range queries.

SWAM [4] employs a Voronoi-based space partitioning scheme and uses a small-world graph overlay with routing tables of size $O(1)$. The overlay is not built by some regular partitioning scheme (e.g. kd-tree [5]) but uses a sample technique to place the fingers.

Multi-attribute range queries were also addressed by Mercury [6] which needs a large number of replicas per item to achieve logarithmic routing performance. SWORD [12] uses super-peers and query-caching to allow multi-attribute range queries on top of the Bamboo-DHT [15].

Ganesan *et al.* [9] proposed two systems for multi-dimensional range queries in peer-to-peer systems: SCRAP and MURK. SCRAP uses the traditional approach of mapping multi-dimensional to one-dimensional data with space-filling curves which destroys the data locality. Consequently, each single multi-dimensional range-query is mapped to several one-dimensional queries. MURK is more similar to our approach, as it divides the data space into hypercuboids with each partition assigned to one node. In contrast to SONAR, MURK uses a heuristic approach based on skip graphs [3] to set routing fingers.

3 System Design

We first present the overlay topology of SONAR and then discuss its routing and lookup strategy. Thereafter we present mechanisms that make SONAR robust under churn.

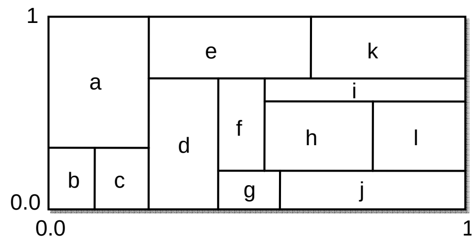


Fig. 1: Example two-dimensional overlay with attribute domains $[0, 1]$.

SONAR is used to store and retrieve *objects*. It works on a d-dimensional torus, the *key-space*. Objects have a name, the *key*, which is a vector of d components, the *attributes* of the key. Each *dimension* of the torus is responsible for one *attribute domain*. Figure 1 illustrates a two-dimensional key-space ($[0, 1]^2$). Arbitrarily located computers, the *nodes*, are each responsible for a dedicated area (hypercuboid) in the key-space of the overlay (rectangles in Fig. 1). The *node-space* has the same extent as the key-space, but is completely filled with nodes. Two nodes in the node-space are *adjacent* (or *neighbors*) when their key-space is adjacent. The direct mapping between key-space and node-space guarantees adjacent keys to be stored on the same or adjacent nodes, which enables efficient range queries across node boundaries by local query propagation.

Nodes are dynamically assigned to the key-space such that each node serves roughly the same number of objects. Load-balancing is done by changing the responsibility of nodes instead of moving around objects in the key-space. That becomes necessary when the number of objects or nodes in the system changes (Sect. 3.4).

3.1 Overlay Topology

As illustrated in Figure 1, the two-dimensional key-space is covered by rectangles, each of them containing about the same number of objects. Because the keys are generally not uniformly distributed, the rectangles have different sizes and thus may have more than one neighbor per direction. The neighbors are stored in *neighbor lists*, one per dimension.

The overlay described so far resembles that of CAN [14] except for the hashing in CAN, which prevents efficient range queries. Consequently, SONAR would also need $O(\sqrt[d]{N})$ network hops if it would just use the neighbors for routing. In the following, we introduce routing tables to achieve logarithmic routing performance.

3.2 Routing

For routing, SONAR uses separate *routing tables*, one per dimension. Each routing table contains *fingers* spanning an exponentially increasing number of nodes

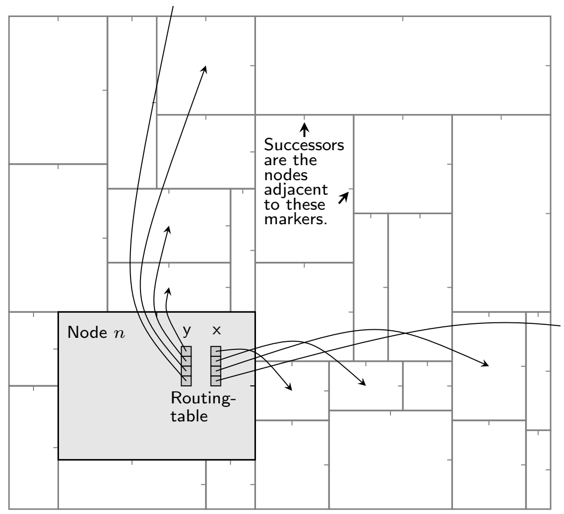


Fig. 2: Routing table for the two-dimensional case

(Fig. 2). With a total of $\log N$ routing fingers, the average number of hops is reduced to $O(\log N)$ [17].

To calculate its i^{th} finger in the routing table, a node looks at its $(i-1)^{th}$ finger and asks the remote node listed there for the $(i-1)^{th}$ finger. At the lowest level, the fingers point to the successor.

$$finger_i = \begin{cases} successor & : i = 0 \\ finger_{i-1}.getFinger(i-1) & : i \neq 0 \end{cases}$$

This update process works in a running system, but also during startup. Initially, all fingers are set to *unknown* except for the finger to the successor. Filling the second entry will always succeed, because the successor knows its successor. Filling further entries may fail (result *unknown*), because the remote node may not have determined the corresponding entry yet. But with subsequent periodic updates, eventually all nodes will get their entries filled. The resulting structure is similar to skip lists [13], but the behavior is more deterministic.

Successor used for routing. A node may have more than one neighbor per direction. We define the node adjacent to the middle of the respective side to be the *successor*. Successors are marked by small ticks in Figure 2.

Due to the different box sizes and the calculation of longer fingers from shorter ones, fingers are not necessarily straight in one direction. Slight deviations in y-direction might occur when following the fingers of the x-direction (and vice versa), as shown in Figure 2. Our empirical results indicate, however, that this does not affect the logarithmic routing performance (Sect. 4).

```
// calculates the entries of a routing table
void updateRoutingTable(int dim) {
  int  i = 1;
  bool done = false;

  rt[dim][0] = this.Successor[dim];

  while (!done) {
    Node candidate = rt[dim][i - 1].getFinger(dim, i - 1);
    if (IsBetween(dim, rt[dim][i - 1].Key, candidate.Key, this.Key)){
      rt[dim][i] = candidate;
      i++;
    } else
      done = true;
  }
}

// checks whether the resp coordinate of pos lies between start and end
bool IsBetween(Dim dim, Key start, Key pos, Key end);
```

Fig. 3: Finger calculation for dimension **dim**.

Routing table size. Each node holds approximately $\log N$ fingers in its routing tables. However, not knowing the total number of nodes N, how many fingers should a node put into each of its d routing table so that the total is $\log N$? Mercury [6] predicts the system size N by estimating the key density. SONAR uses a simpler, deterministic solution with less overhead.

For each dimension *dim*, SONAR's finger update algorithm (Fig. 3) inserts an additional finger $finger_i$ as long as its position is between that of the last routing table entry $finger_{i-1}$ and that of the node itself. Otherwise the new finger circles around the ring and is not inserted.

Our results in Section 4 confirm that each node holds indeed $\log N$ fingers. The construction process guarantees—in contrast to Chord [18]—that no two fingers point to the same node. Since the fingers in the routing tables span an exponentially increasing number of nodes, the routing table of each dimension has a total of $\lceil \log D \rceil$ entries on the average, where D is the number of nodes in this direction on the torus.

Cost of a finger update. Our periodically running finger update algorithm needs just one network hop to determine an entry in the routing table. Chord in contrast needs $O(\log n)$ for the same operation, because it performs a DHT lookup to calculate a finger.

3.3 Lookup and Range Queries

As in other DHTs, SONAR uses greedy routing. In each node the finger that maximally reduces the Euclidean distance to the target in the key-space is followed, independently of the dimension (see Fig. 4).

SONAR supports range queries with multiple attributes. In its most basic form a range query is defined by d intervals for the d attribute domains. The

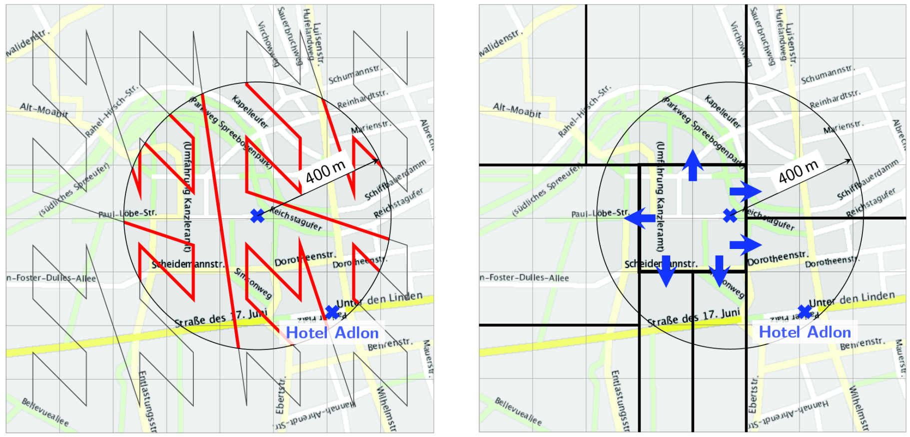
```
// find the responsible node for a given key
Node find(Point target) {
  Node nextHop = findNextHop(target);
  if (nextHop == this)
    return this;
  else
    return nextHop.Find(target);
}

double getDistance(Node a, Point b);

Node findNextHop(Point target) {
  Node candidate = this;
  double distance = getDistance(this, target);

  if (distance == 0.0)
    // found target
    return this;

  for (int d = 0; d < dimensions; d++) {
    for (int i = 0; i < rt[d].Size; i++) {
      double dist = getDistance(rt[d][i], target);
      if (dist < distance) {
        // new candidate
        candidate = rt[d][i];
        distance = dist;
      }
    }
  }
  // will never happen:
  Assert(candidate != this);
  return candidate;
}
```

Fig. 4: Lookup for a target.

range query finds all keys whose attributes match the respective intervals and returns the corresponding objects. Because of their shape, such range queries are called d-dimensional *rectangular* range queries.

In practice, users sometimes need to define circles, polygons, or polyhedra in their queries. Figure 5 illustrates a two-dimensional circular range query defined by a center and a radius. Here, we assume a person located in the governmental district of Berlin searching for a hotel in 'walking distance' (circle around the person). The query is first routed to the node responsible for the center of the circle and then forwarded to all neighbors that partially cover the circle (Fig. 5b). The query is checked against the local data and the results are returned to the requesting node. Figure 6 shows the pseudocode of this algorithm. Note that redundant messages are eliminated. `op` is an additional check for objects in the queried area—in this case for type hotel.

SONAR performs a range query with a single lookup. When the target node does not hold the complete key range, the query is locally forwarded. Systems with space-filling curves, in contrast, usually require more than one lookup for a single range query because they map connected areas to multiple independent line segments, see Figure 5a.



(a) Z-curve (8 line segments): $8 \cdot log_2(N)$ (b) Neighborhood broadcast: $log_2(N)+6$

Fig. 5: Circular range query.

```
// perform a range query
void queryRange(Range r, Operation op) {
  Node center = Find(r.Center);
  center.doRangeQuery(r, op, newId());
}

void doRangeQuery(Range r, Operation op, Id id) {
  // avoid redundant executions
  if (pastQueries.Contains(id))
    return;
  pastQueries.add(id);

  foreach (Node neighbor in this.Neighbors)
    if (r ∩ neighbor.Range != ∅)
      neighbor.doRangeQuery(r \ this.Range, op, id);

  // execute operation locally
  op(this, r);
}
```

Fig. 6: Range query algorithm.

3.4 Topology Maintenance to Handle Churn

Node Join. When a node joins the system, the key-space of a participating node has to be split and the key responsibilities subdivided. To achieve this, two things must be done (we first describe random splitting and then include load balancing):

1. Select a random target node: A random position in the key-space is routed to and a random walk is started from there. The final target node of this is the candidate to be split. The random walk ensures that nodes responsible for larger areas of the key-space are not preferred over smaller ones.

2. Split the key-space and transfer one part to the new node: Splits are parallel to one of the coordinate system axes. The selection of the axis to be split should not strictly favor one dimension over the others because the number of nodes to be contacted for a range query could become disproportionately high, when a large interval for the favored dimension is specified in the query. Also, node leaves could become more expensive.

Node Leave. Handling a leaving node is more difficult, because it is not always obvious which node can fill the area of the leaving node. For example, in Figure 1, the area of node f cannot be merged with any of its neighbors, because this would result in a non-rectangular node-space.

Therefore the node-space is constructed in such a way that the splitting plane forms a *kd-tree* [5]. KD-trees are used only for topology maintenance, similar as in MURK [9], but not as index structures like in database systems. The space of a leaving node can be taken over by a neighboring node which is also a sibling in the kd-tree. By keeping the tree balanced the probability of having a sibling as a neighbor increases. Each node must remember its position in the kd-tree, a bit-string describing the path from the root of the tree to the node itself.

If no neighboring nodes are siblings in the kd-tree, another node must be found to fill the gap. Either a neighboring node additionally takes over the responsibility of the separate area until a free node can be found, or two completely independent nodes that are siblings in the kd-tree have to be found to merge them and thus free a node that takes over the free area. The former concept, called *virtual nodes*, is also used for load-balancing in other systems.

Load Balancing. Load-balancing can be implemented by either adjusting the boundaries of the responsibilities locally or freeing nodes in underloaded areas and moving them to overloaded areas. The former has similar issues as a node leave—the boundaries are interlocked with limited room for adjustments. The latter was shown to be converging [11] with predictable performance.

The balancing can be based on different metrics for load, like object or query load or a combination of both. To avoid thrashing effects a threshold for performing a load-balancing round must be introduced.

4 Empirical Results

For testing the performance of SONAR we used a traveling salesman data set with 1,904,711 cities[1]. The cities' geographical locations follow a Zipf distribution [19] which is also common in other scenarios.

We assigned the responsibility of nodes by recursively splitting the key-space at the longer side, so that each part gets half of the cities until enough rectangles are created. Figure 7 shows a sample splitting for 256 nodes.

The coordinates were mapped onto a doughnut-shaped torus rather than a globe, because in a globe all vertical rings meet at the poles. This would not only

[1] http://www.tsp.gatech.edu/world

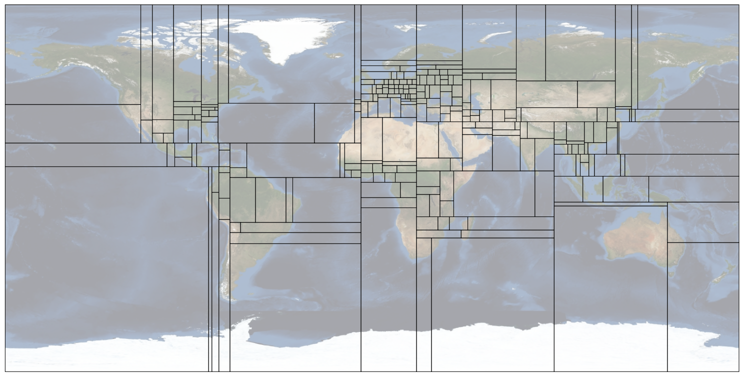


Fig. 7: 1,904,711 cities split evenly into 256 rectangular nodes.

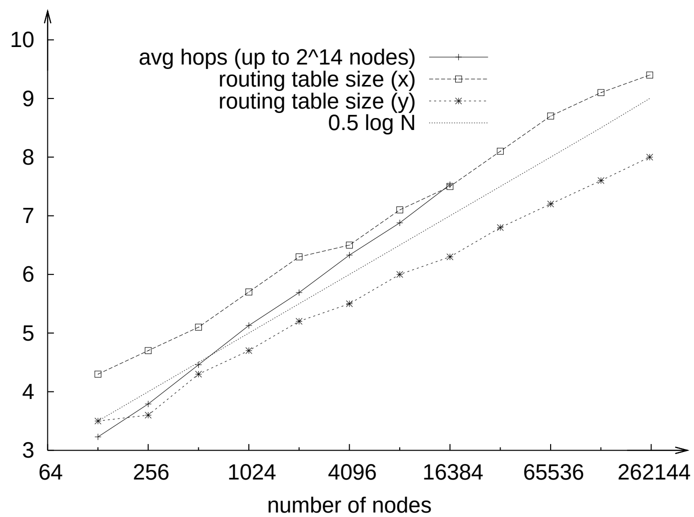


Fig. 8: SONAR results for increasing system sizes (2-dimensional).

cause a routing bottleneck at the poles but would also result in different ring directions for the western and eastern hemisphere (southwards vs. northwards).

Figure 8 shows the results for various all-to-all searches in networks of different sizes. The routing performance, depicted by the '+' ticks, almost perfectly matches the expected $0.5 \log_2 N$ hops. Only in the larger networks the expected value slightly deviates.

We also checked whether the number of fingers in the routing tables, which are calculated without global information (Fig. 4), meets our expectations. The '□' ticks give the routing table sizes in horizontal direction, and the '$*$' ticks represent the sizes in vertical direction. As expected, both graphs have the same

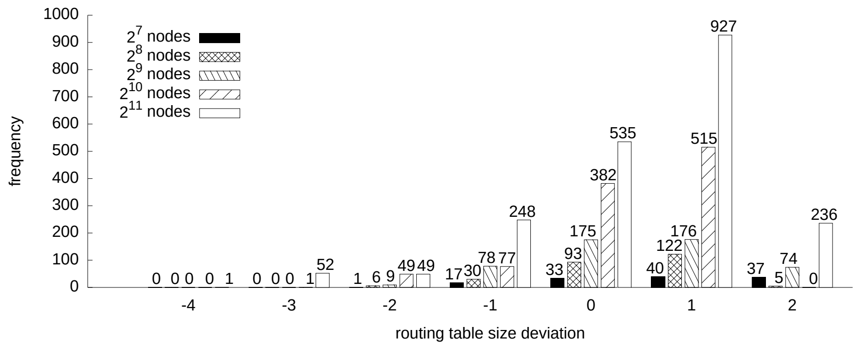


Fig. 9: Routing table size deviation from the expected value.

slope of $0.5 \log_2 N$: One lies consistently above, the other below. This is attributed to the different domain sizes of the coordinate system (360 versus 180 degrees) and to the uneven number of splitting planes.

Figure 9 gives further insight into the characteristics of SONAR's routing tables. It shows—again for various network sizes—the deviation of the table sizes from their expected size $\log_2 N$ (denoted by '0'). As can be seen, the same pattern applies for all network sizes: About 50% of the tables contain one extra entry, about 25% meet the expected size of $\log_2 N$, while there is a decreasing number of tables with fewer entries. These deviations are caused by the uneven key distribution and by SONAR's finger update algorithm which has a tendency to insert in some cases an extra finger that is more than halfway around the ring (but still 'left' of the own node).

5 Conclusion

SONAR efficiently supports range-queries on multi-dimensional data in structured overlay networks. It needs $O(\log N)$ routing steps for processing range-queries of arbitrary shapes and an arbitrary number of attribute domains. The finger calculation needs just one hop for updating an entry in the routing table.

We presented empirical results from a Zipf distributed data set with approximately two million keys. The results confirm that SONAR does its routing with a logarithmic number of hops—even in skewed data distributions. Additional tests with other practical and uniform distributions (not shown here) gave the same logarithmic routing performance. Furthermore, we observed that the sizes of the distributed routing tables are always $O(\log N)$ although they are autonomously maintained by the nodes with local information only.

Acknowledgements

Thanks to the anonymous reviewers for their valuable comments. The topographic images were taken from the 'Blue Marble next generation' project of NASA's Earth Observatory. Thanks to Slaven Rezić for the street map of Berlin.

References

1. K. Aberer, L. Onana Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, and M. Hauswirth. The essence of P2P: A reference architecture for overlay networks. P2P 2005, 2005.
2. A. Andrzejak and Z. Xu. Scalable, efficient range queries for Grid information services. P2P 2002, 2002.
3. J. Aspnes and G. Shah. Skip graphs. *SODA*, Jan. 2003.
4. F. Banaei-Kashani and C. Shahabi. SWAM: A family of access methods for similarity-search in peer-to-peer data networks. *CIKM*, Nov. 2004.
5. J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, Vol. 18, No. 9, 1975.
6. A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. *ACM SIGCOMM 2004*, Aug. 2004.
7. Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein. A Case Study in building layered DHT applications. SIGCOMM'05, Aug. 2005.
8. V. Gaede and O. Günther. Multidimensional access methods. ACM Computing Surveys, 30 (2), 1998.
9. P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: Multi-dimensional queries in P2P systems. *WebDB 2004*.
10. D. Karger, E. Lehman, T. Leighton, R. Panigrah, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. *ACM Sympos. Theory of Comp.*, May 1997.
11. D. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. *IPTPS 2004*, Feb. 2004.
12. D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and implementation tradeoffs for wide-area resource discovery. *14th IEEE Symposium on High Performance Distributed Computing (HPDC-14)*, Jul. 2005.
13. W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, June 1990.
14. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. *ACM SIGCOMM 2001*, Aug. 2001.
15. S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. *Proceedings of the USENIX Annual Technical Conference*, Jun. 2004.
16. C. Schmidt and M. Parashar. Enabling flexible queries with guarantees in P2P systems. *IEEE Internet Computing*, 19-26, May/June 2004.
17. T. Schütt, F. Schintke, and A. Reinefeld. Structured overlay without consistent hashing: Empirical results. *GP2PC'06*, May. 2006.
18. I. Stoica, R. Morris, M.F. Kaashoek D. Karger, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet application. *ACM SIGCOMM 2001*, Aug. 2001.
19. G. Zipf. Relative frequency as a determinant of phonetic change. *Harvard Studies in Classical Philiology*, 1929.