Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Simulation-based evaluation of a scalable publish/subscribe system

D3.2.3

Due date of deliverable: November 30^{th}, 2007
Actual submission date: May 31^{th}, 2008

Start date of project: June 1^{st} 2006

Type: Deliverable
WP number: WP3.2
Task number: T3.2.2

Responsible institution: ZIB
Editor & and editor's address: Thorsten Schütt
Zuse Institute Berlin
Takustrasse 7
14195 Berlin
Germany

Version 2.0 / Last edited by Thorsten Schütt / May 26^{th}, 2008

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	$\surd$
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	2007/10/11	Thorsten Schütt	ZIB	first draft
0.2	2007/10/17	Thorsten Schütt	ZIB	extended different pub/sub styles
0.3	2007/10/31	Thorsten Schütt	ZIB	future work
0.9	2007/11/27	Thorsten Schütt	ZIB	included the reviewers' suggestions
1.0	2007/11/30	Thorsten Schütt	ZIB	final changes
1.1	2008/05/20	Thorsten Schütt, Alexander Reinefeld	ZIB	rewrite after rejection at review
2.0	2008/05/26	Thorsten Schütt, Alexander Reinefeld	ZIB	updated after internal review

Reviewers:
Thilo Kielmann (VU), Björn Kolbeck (ZIB)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T3.2.2	Design and implementation of a scalable publish/subscribe system	ZIB*

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Executive summary

This deliverable presents the state of the design and evaluation of the XtreemOS software component "Publish/Subscribe System". This system is a key component of the highly available and scalable infrastructure as described in deliverable D3.2.1 under the responsibility of WP 3.2.

Distributed hash tables and structured overlay networks are inherently scalable systems which are usually used for key-value stores. The main design principle is that each node knows only a very small part of the system and delegates tasks to its neighboring nodes if he can not perform the task himself. Maintenance happens in a similar way, that each node only tries to repair its neighborhood. This results a simple algorithms. We present in great detail the inner workings of Chord$^{\#}$and SONARand simulations showing their scalability.

Chord$^{\#}$ is a DHT which allows to efficiently execute range queries. It eliminates the hash function, which is the center piece of traditional DHTs, and instead stores the items in lexicographical order. The lexicographical order ensures that similar keys are stored on the same or on neighboring nodes. The support of range queries is an important pre-requisition for implementing publish/subscriber systems because they allow to build group based notification systems with hierarchically structured topics. The simpler model of a topic-based publish/subscribe system can be seen as a special case of hierarchical topics with a flat hierarchy.

SONAR is generalization of Chord$^{\#}$ to a multi-dimensional key space. Where Chord$^{\#}$ supports a one-dimensional key space with total order, like e.g. strings, keys in SONAR are multi-dimensional vectors. Again as in Chord$^{\#}$ similar items are stored on neighboring nodes, which allows to efficiently perform range-queries of almost arbitrary shapes. A 2-dimensional instance of SONAR could be used as a geographic database, supporting queries like, give me all hotels north-west of my current position less than 1000m away. For publish/subscribe, the high-dimensional key space can be used to support content-based models.

As a basis for our publish/subscribe system we designed, implemented and evaluated the Chord$^{\#}$ and SONAR algorithms. We extended them towards a full-fledged publish/subscribe system and proved their scalability by means of simulation. The system is shown to scale up to some thousands of nodes (i.e. publishers and subscribers).

Contents

1 Introduction

One important goal of Workpackage 3.2 of the XtreemOS project is to provide a *highly scalable/publish subscribe service* (pub/sub). This service will be used by XtreemOS services to notify other services and users about important, possibly time-critical events within the XtreemOS operating system. A typical event could be an unexpected termination of a job, an update in the file system, or the availability of new resources and services. Hence, the herewith described pub/sub system is at the core of many services in XtreemOS.

Typical *publishers* will be services like the file system, the monitoring component, or the scheduler. *Subscribers*, on the other hand, may be services as well as users, e.g. on mobile devices.

There exists a wealth of literature on pub/sub systems (for an overview see [10]). Most of them are based on some kind of group communication. Only few pub/sub systems provide scalability up to thousands of publishers or subscribers. Pub/sub systems comprise two components:

1. a component used to register those entities that must be notified at the arrival of a given event, and

2. a component used to notify the entities.

Structured overlays with distributed hash tables (DHTs) are a natural candidate for implementing services that are truly scalable. However, it became only recently possible to support complex queries (such as range queries) on DHTs. One of these systems is Chord$^{\#}$ which we use as a basis for our implementation. With Chord$^{\#}$, complex queries can be used to map the database part on a DHT. With increasing complexity of the supported queries, the pub/sub system is able to provide powerful subscription styles [10] like the following:

DHT Capability	**Pub/Sub Style**
simple lookup	topic-based
1D range queries	hierarchical topics
multi-dimensional range queries	content based pub/sub

This deliverable documents the design, implementation and evaluation of the proposed pub/sub architecture based on simulations.

The next two sections introduce Chord$^{\#}$ resp. SONAR and show their scalability based on simulations. Sections 4.1, 4.2 and 4.3 introduce the architecture of the pub/sub service. Section 5 gives a short summary of the simulation results. An outlook on the next steps is given in Sect. 5.2, including a short summary of the papers [31, 32, 33] which resulted from this deliverable.

2 A Pub/Sub Backend for Simple Queries: Chord$^{\#}$

Chord$^{\#}$ is a DHT that supports range queries. It is more efficient than the well-known Chord DHT, and it does so with the same logarithmic number of entries in the routing table. In the following, we introduce Chord$^{\#}$ by deriving it step by step from Chord as illustrated in the four parts of Fig. 1.

Fig. 1a: Chord organizes the nodes $N0, \ldots, N15$ in a logical ring, each of them being responsible for a subset of the keys $0, \ldots, 2^8 - 1$. In each node a finger table holds the addresses of the peers halfway, quarter-way, $1/8$-way, $1/16$-way, ..., around the ring. When a node (e.g. N0) 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 [36].

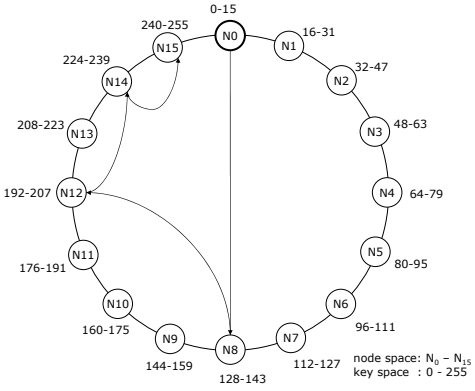
Fig. 1b: Because this scheme does not support range queries, we eliminated the hashing of keys as shown in Fig. 1b. All keys are now sorted in lexicographic order, but unfortunately nodes responsible for popular keys (e.g. 'E') will become overloaded—both in terms of storage space and query load. Hence,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

Fig. 1c: When also eliminating the hashing of *nodes*, we need to introduce a scheme for explicit load balancing [17]. Part (c) shows that nodes can now be placed at any suitable place in the ring to achieve an almost even key distribution. However, without adjusting the fingers in the finger table, much more hops are needed to retrieve a given key. The lookup started in node N13 for key 'R', for example, needs six instead of four hops. In the extreme case, routing could degrade to $O(n)$.

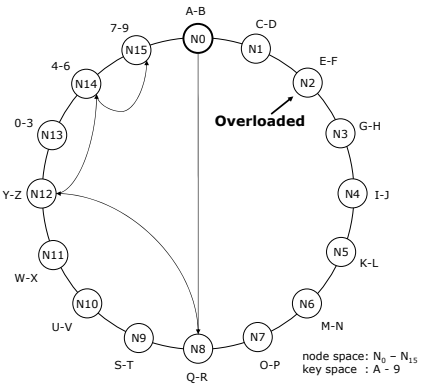
Fig. 1d: Here, we introduce a scheme that dynamically adjusts the fingers in the finger table. The lookup performance is now again $O(\log n)$ – just as in Chord. But in contrast to Chord this new variant does the routing in the node space rather than the key space, and it supports complex queries –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

The most important step is the substitution of Chord's hash function by a key order preserving function. When doing so, the keys are no longer equally distributed over the nodes but they follow some unknown density function. To still obtain a logarithmic routing effort, the fingers must be computed in such a way that they cross a logarithmic number of *nodes* in the ring. The following finger placement algorithm does this (the infix operator $x \,.\, y$ retrieves y from the routing table of a node x):

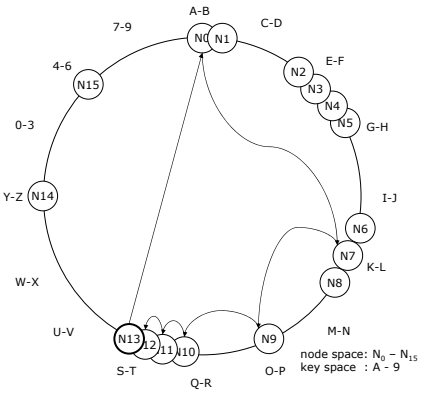
$$\mathit{finger}_i = \begin{cases} \mathit{successor} & : i = 0 \\ \mathit{finger}_{i-1} \,.\, \mathit{finger}_{i-1} & : i \neq 0 \end{cases}$$



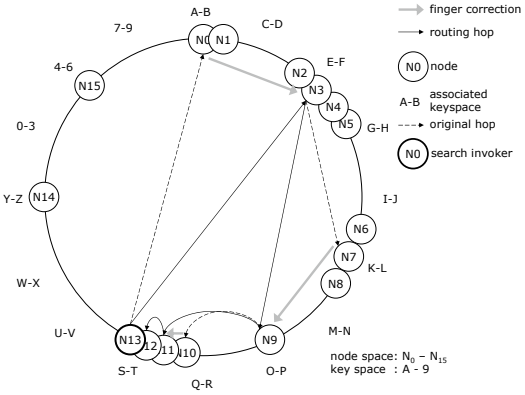
(a) Chord



(b) Chord without hashing



(c) Chord without hashing but with load balancing



(d) Chord$^{\#}$

Figure 1: Transformation steps from Chord to Chord$^{\#}$

To calculate the i^{th} 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. In general, the fingers in level i are set to the fingers' neighbors in the next lower level $i-1$. At the lowest level, the fingers reference to the direct successors.

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 better informed remote nodes for adjusting the fingers in its finger table by recursive finger references.

2.1 Logarithmic Routing Performance

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

$$p_i = (n \oplus 2^{i-1}) \quad \text{for} \quad 1 \leq i \leq m$$

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

$$p_i = (n \oplus 2^{i-2}) \oplus 2^{i-2}$$

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 only 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:*

1. *Each node is responsible for at most $(1+\epsilon)\frac{K}{N}$ keys.*
2. *When 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).*

[1] We assume calculations to be done in a ring using (mod 2^m).

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:*

$$\int_{keyspace} d(x)\,dx = N.$$

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.

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

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. 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:

$$\int_{keyspace} d(x)\,dx = \sum_{i=0}^{N-1} \int_{a_i}^{a_{i+1}} d(x)\,dx = N$$

■

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 $\tilde{d}(x)$.

2.1.1 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 f_i in a node n with the following scheme:

$$p_i = (n \oplus 2^{i-1}), 1 \leq i \leq m \tag{1}$$

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 f_m. It points to $n + 2^{m-1}$ 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, which is stored on the $\frac{N}{2}$-th node to the right, can be predicted.

$$\int_{n}^{p_m} d(x)dx = \frac{N}{2}$$

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:*

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

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

$$\int_n^{p_i} d(x)\, dx = \frac{2^{i-1}}{2^m} N$$

$$\int_n^{p_i} \frac{N}{2^m}\, dx = \frac{2^{i-1}}{2^m} N$$

$$\frac{N}{2^m}(p_i \ominus n) = \frac{2^{i-1}}{2^m} N$$

$$p_i = n \oplus 2^{i-1}$$

■

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

2.1.2 Finger Placement in Chord$^{\#}$

Theorem 3 (Chord$^{\#}$ Finger Placement):

$$\mathit{finger}_i = \begin{cases} \mathit{successor} & : i = 0 \\ \mathit{finger}_{i-1} \,.\, \mathit{finger}_{i-1} & : i \neq 0 \end{cases}$$

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

$$\int_n^{p_i} d(x)\,dx = \frac{2^{i-1}}{2^m} N,\ 1 \leq i \leq m \tag{2}$$

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

$$\int_n^{X} d(x)\,dx = \frac{2^{i-2}}{2^m} N \tag{3}$$

$$\int_X^{p_i} d(x)\,dx = \frac{2^{i-2}}{2^m} N \tag{4}$$

In Eq. 3 and Eq. 4, the only unknown is X. Comparing Eq. 3 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. 3), 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. 4. So, Eq. 2 is equivalent to

$$\mathit{finger}_i = \mathit{finger}_{i-1} \,.\, \mathit{finger}_{i-1}$$

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. ∎

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$^{\#}$.

2.2 k-ary Search

The protocol described so far has a routing performance of $O(\log_2 N)$ hops. Following the lines of DKS [3], this can be improved to $O(\log_k N)$ hops for arbitrary

k by simply adding fingers to the routing table. We first review k-ary search in Chord and them apply it to Chord$^{\#}$.

In Chord the longest finger (f_{m-1}, where m is the number of bits in the identifiers) and the key of the current node (f) split the ring into two halves ($(f, f_{m-1}]$ and $(f_{m-1}, f]$). The next shorter pointer f_{m-2} splits the first half again into two halves ($(f, f_{m-2}]$ and $(f_{m-2}, f_{m-1}]$). This scheme is recursively applied until the subsets contain only one key. It can be easily seen, that this scheme cuts the distance to the goal in half with each hop, resulting in $O(\log_2 n)$ hops.

By dividing into k equally sized subsets at each level, each hop reduces the distance to $\frac{1}{k}$-th and the overall number of hops per search to $O(\log_k n)$. This extension is implemented by adding $k-2$ columns to the routing table and calculating the fingers as follows: $f \oplus (i+1)\frac{2^m}{k^l}$, where $1 \leq i \leq k$ and $1 \leq l \leq \log_k(2^m)$.

```
1  # ask node n to find the successor
2  # of id
3  n.find_successor(id)
4    if (id ∈ (n, successor])
5      return successor;
6    else
7      n'=closest_preceding_node(id);
8      return n'.find_successor(id)
9
10 # search the local table for the
11 # highest predecessor of id
12 n.closest_preceding_node(id)
13   for i = m downto 1
14     if (finger[i] ∈ (n, id))
15       return finger[i];
16   return n;
```

(a) Chord's original routing algorithm [36].

```
1  # ask node n to find the
2  # predecessor of id
3  n.find_predecessor(id)
4    if (id ∈ (n, successor])
5      return n;
6    else
7      n'=closest_preceding_node(id);
8      return n'.find_predecessor(id)
9
10 # search the local table for the
11 # highest predecessor of id
12 n.closest_preceding_node(id)
13   for i = m downto 1
14     if (finger[i] ∈ (n, id])
15       return finger[i];
16   return n;
```

(b) Improved routing algorithm for Chord.

Figure 2: Improving the finger checks to $O(1)$

Chord$^{\#}$ can be improved analogously by k-ary search, but here we need to calculate the fingers differently. A closer look at the above algorithm reveals that the i-th finger in the l-th row points to the $i * b^{l-1}$-th neighbor of the current node. This can be rewritten to make the recursive structure more obvious. Depending on i, two cases must be distinguished:

$i = 1$: In this case, $i * b^{l-1}$ can be rewritten as $t * b^{l-2} + u * b^{l-2}$, where $t + u = b$. Note, that t and u can be chosen arbitrarily as long as their sum equals b.

As explained in Sect. 2.1, the $a + b$-th neighbor can be found if the a-th neighbor is known and the a-th neighbor knows its b-th neighbor. When both facts are locally known, the target can be found in one hop. This property was used in the 2-ary Chord$^{\#}$ and is also the base for the k-ary implementation.

All nodes can always find their $t*b^{l-2}$-th as well as their $u*b^{l-2}$-th neighbor in the local routing table by going one row up from the current entry without any accesses to the network. So, in this case the entries can be updated with one network hop.

$i \neq 1$: Here we rewrite $i*b^{l-1}$ as $t*b^{l-1}+u*b^{l-1}$, where $t+u=i$. Again, t and u can be chosen arbitrarily. As in the former case, the necessary information can be obtained by looking at the entries to the left of the current entry.

In summary, the finger updates are as efficient as the ones for the 2-ary Chord$^{\#}$. But with a slight twist, the whole system is dynamic and therefore routing table entries might be incorrect or missing. To adapt, Chord$^{\#}$ can exploit that t and u may be chosen within certain limits and thereby avoiding stale entries. This flexibility can also be used to find inconsistencies in routing tables.

Theorem 4 (Chord$^{\#}$ Finger Placement with k-ary Search): *The following algorithm for computing $finger_{i,j}$ with $1 \leq j \leq \lceil \log_b N \rceil, 1 \leq i < k$ routes in $O(\log_k N)$ hops.*

$$finger_{i,j} = \begin{cases} successor & : i = 1, j = 1 \\ finger_{l,j} \,.\, finger_{m,j} & : i > 1, l + m = i \\ finger_{l,j-1} \,.\, finger_{m,j-1} & : i = 1, j > 1, \\ & \quad l + m = b \end{cases}$$

The update process may be further improved by piggypacking information with search operations. This on-thy-fly correction has almost no traffic overhead and allows more frequent updates. Unrolling the equation would increase the number of hops, but it would also increase the adoption rate under churn.

2.3 Fast Finger Check

In Chord, all keys in the interval $(n, n.successor]$ are stored at the *successor* of n [36]. We propose to store the keys at node n instead. For lookups, Chord searches for the predecessor of a key first, and then returns the succesor of this key. In our approach we search for the key and return the node itself. Thereby most operations on keys need one hop less than in normal Chord.

Furthermore the check in closest_preceding_node (line 12 in Fig. 2 can then search for the closed instead of the open interval (line 14). This makes no difference for most routing requests, but in case of updating a finger that is already up-to-date, it reduces the costs for this from O(log n) to O(1).

In a Chord-search for a node that is already in the finger table, it is not able to simply follow the finger itself, because this actually points to the successor of

the wanted key. But all routing requests have first to visit the predecessor of the wanted key. A routing to the predecessor of a finger represents the worst case for routing, inducing O(log n) hops.

With our modifications routing operations do not have to traverse anylonger through the predecessor, but routes directly to the wanted node. In case of routing table entries, a finger points directly to the wanted node and a check of this finger can be performed in O(1).

All our simulations (Chord and Chord$^{\#}$) were done using this improvement (see Sect. 2.4). In case of Chord$^{\#}$ this improvement has no effect whereas Chord's bandwidth consumption benefits from it.

2.4 Simulations

To evaluate Chord$^{\#}$ under practical conditions, we implemented an event-based simulator following the approach of Li *et al.* [19, 20] who suggested to compare P2P protocols under churn, i.e., with nodes joining and leaving with an exponentially distributed lifetime of nodes. We simulated an overlay network with 1024 nodes running for a total of 6 hours. Each participating node issues every 10 minutes a lookup 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 dataset [15] which is based on real data observed with Internet DNS servers[2].

As the simulation does not account for user data, but only for protocol overhead, it measures the worst case behavior. Following the approach of Li *et al.*, we varied the same parameters for testing the algorithm's performance, resulting in a total of 480 resp. 1728 simulation runs[3] (the parameters form a 6-tuple):

1. **Base** is the branching factor of each finger table entry. Each finger table contains a total of $(base - 1)\log_{base}(n)$ fingers. Values: 2, 8, 16, 32.

2. **Successors** is the number of direct successors stored in each nodes' successor list. Values: 4, 8, 16, 32.

3. **Successor Stabilization Interval** denotes the time spent between two updates of the nodes' successor lists. Values: 30s, 60s, 90s.

[2]We observed an inconsistency in the data given in Li *et al.* [19, 20], who seem to have also used the first 1024 node entries of the King dataset, which actually have an average round-trip latency of 197 ms. They claim, however, a latency of 178 ms which is only true when taking the whole set of 1740 nodes.

[3]For Chord, we ran less simulations because e.g. it is not possible to implement piggybacking in Chord.

```
1  #number of nodes in the ring
2  n.estimate_size_of_ring()
3    return n.get_distance(n.Key - 1)
4
5  #number of nodes between n.Key and key
6  n.get_distance(key)
7    lower = 0;
8    upper = 1 << m;
9    middle = n._find_successor_limited(key, upper);
10   if(middle > 0)
11     return n.search_ring_size(key, lower, middle);
12   return middle; //-1: ring is larger than 1<<m
13
14
15 #binary search for distance
16 n.search_distance (key, lower, upper)
17   if(lower == upper)
18     return lower;
19   middle = n._find_successor_limited(
20                key, (lower + upper)/2);
21   if(middle > 0)
22     return n.search_distance(key, lower, middle)
23   return n.search_distance(key, middle, upper);
24
25 n.closest_preceding_node_limited(key, limit)
26   for i = m downto 1
27     if(finger[i] in (n.Key, key]
28          && (1 << i) <= limit)
29       return (finger[i], i);
30   return (null, -1);
31
32 #find key without crossing more than limit nodes
33 n.find_successor_limited(key, limit)
34   if(key \in (n.Key, n.successor.Key])
35     return 1;
36   else
37     (next, index)=closest_preceding_node_limited(
38                        key, limit);
39     if(next == null)
40       return -1;
41     distance = next.find_successor_limited(
42                        key, limit - 1 << index);
43     if(distance == -1)
44       return -1;
45     return 1 << index + distance;
```

Figure 3: Calculating the number of nodes in the ring.

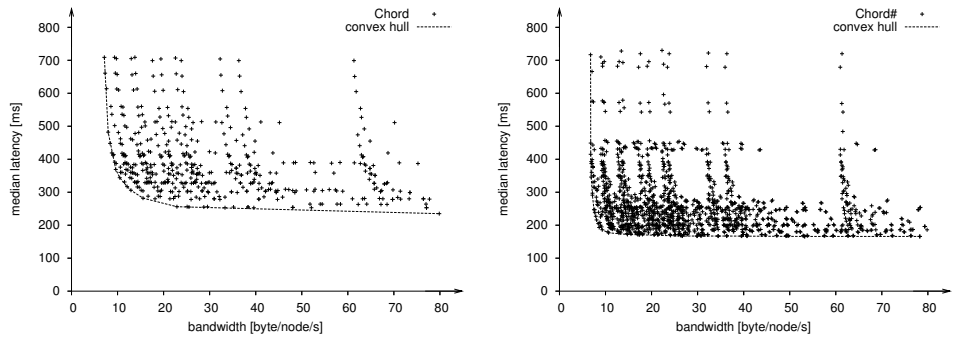


Figure 4: Chord and Chord$^{\#}$ under churn (median latency). Each '+' represents one out of the 480 resp. 1728 experiments. The convex hull (bottom line) illustrates the best parameter combinations.

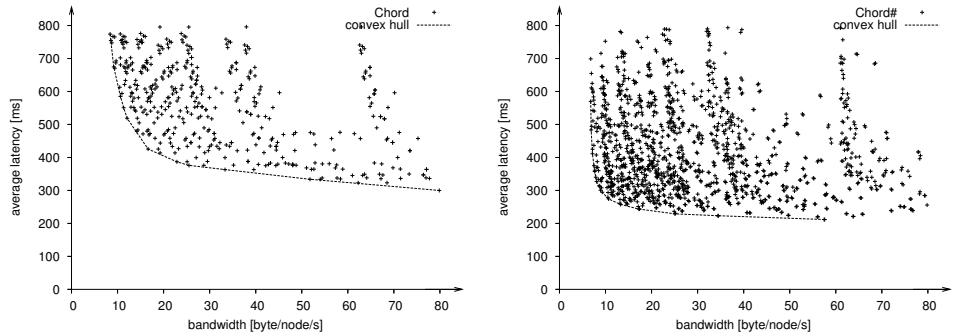


Figure 5: Chord and Chord$^{\#}$ under churn (average latency).

4. **Finger Update Interval** is the time spent between two finger table updates. Values: 30, 60, 300, 600, 900, 1200 s.

5. **Latency Optimizer** tells whether proximity routing was used for improving the latency. Values: 0, 1, 2.

6. **Piggybacking** tells whether routing information is piggy backed on data lookups. Values: true, false.

The performance is measured in terms of latency per key lookup and the cost in terms of bandwidth per node (bytes per node per second).

Figures 4 and 5 show the performance results of Chord (left figure) and Chord$^{\#}$ (right) in terms of lookup latency versus maintenance bandwidth used for keeping the routing table up-to-date. Note that we plotted both, the average latency data (Fig. 4) and the median latency (Fig. 5) because it was unclear to us, why Li *et al.* changed from average to median in their two papers.

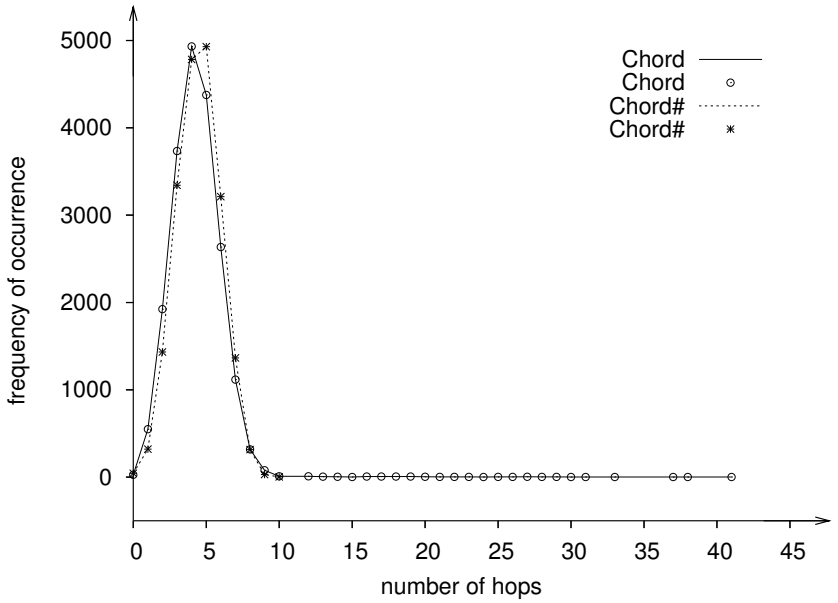


Figure 6: Number of hops for finding random keys in 1024 nodes for Chord$^{\#}$ and Chord. Note that Chord$^{\#}$ needs at maximum $\log(1024) = 10$ hops whereas Chord guarantees the logarithmic routing performance only with 'high probability', that is, there exist cases that need considerably more hops.

Each '+' in the figures represents a complete simulation for one specific parameter combination. The best parameter combinations are those on the convex hull: They represent combinations with low latency at low maintenance cost. All other data points are inferior and can safely be ignored.

Comparing the convex hull of both protocols shows that data lookups of Chord$^{\#}$ are much faster. The data points in the lower left hand corner of the plots are most interesting: They depict the favorable parameter combinations with both, low bandwidth and low latency.

2.4.1 Latency

Chord$^{\#}$ clearly outperforms Chord (Fig. 4, 5). Additionally, we plotted the hop number in Fig. 6. As can be seen, Chord sometimes needs an excessive number of 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. Even though a Chord search may need $O(\log N)$ network hops on the average, this is not so in every single case as can be seen in Fig. 6 Chord$^{\#}$, in contrast, is more predictable in this respect. In the worst case it needs just 10 hops compared to 42 hops of Chord.

2.4.2 Checking the Propagation of Misplaced Fingers

In contrast to Chord which relies on searches for its finger updates, Chord$^\#$ computes its routing fingers recursively. As a consequence, Chord$^\#$'s finger update is faster, but it may propagate outdated fingers. However, our simulations indicate (Sec. 2.4) that there is no measurable negative effect.

This may be attributed to the fact that the finger to the direct successor is mostly correct due to the successor stabilization process. In general, short fingers are correct with higher probability than long fingers, because the information they rely on is fresher. Both, the probability of an erroneous finger and the absolute error increases with the length of the finger, but wrongly placed long fingers are easier tolerated than misplacements of the short ones (which are more likely to be correct). Hence, on a means/cost basis the recursive finger placement is beneficial.

Moreover, k-ary search (Sec. 2.2) provides an elegant way to check the accuracy of fingers by calculating them in several different and independent ways. The results can be compared to find inconsistencies in routing tables.

Fig. 3 presents an alternative method to check whether the i-th finger is indeed pointing to the 2^i-th neighbor on the ring.

2.5 Summary

This chapter shows that Chord$^\#$ is a scalable, distributed key-value store. It efficiently supports lookups for individual keys as well as range queries. In Chap. 1, we introduced the two major components (registry and delivery system) of a pub-/sub system. Given the simulations in this chapter, Chord$^\#$ will be used to efficiently implement the registry for a pub/sub with simple topics as well as with hierarchical topics. Chap. 4 will show how this is implemented in detail.

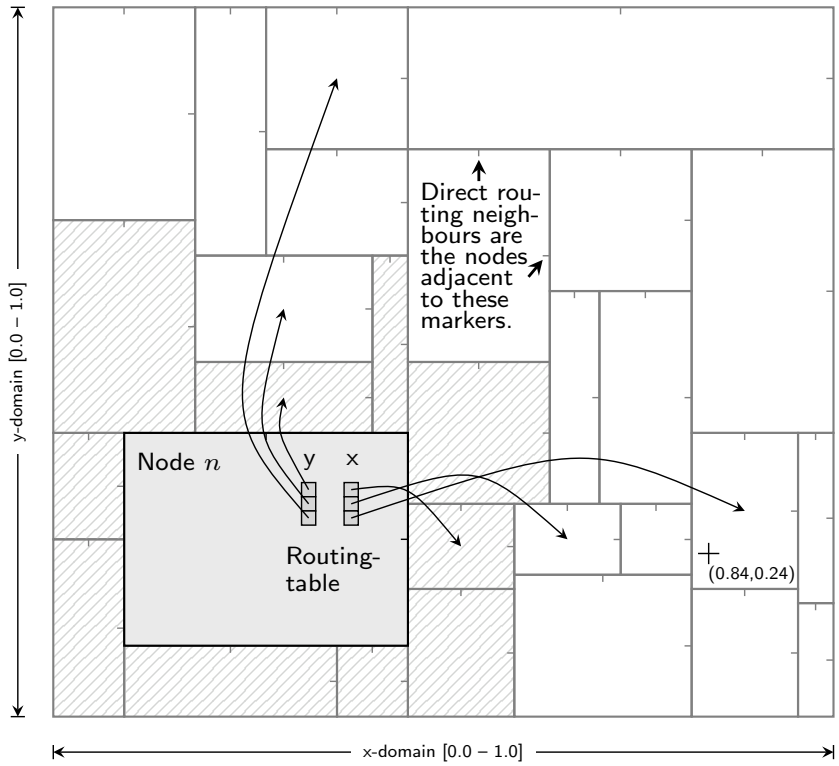


Figure 7: Routing table for $d = 2$

3 Backend for Content-based Pub/Sub: SONAR

To allow for storing multi-dimensional key/value pairs, we extended the Chord$^{\#}$ rings to a d-dimensional torus. In the following, we describe the resulting SONAR system, which we used as the basis for our more advanced publish/subscriber system (Section 4.1) which also supports hierarchical topics.

As before, we first present SONAR in its simplest form and then introduce additional features that make the system robust under churn.

SONAR operates on a d-dimensional Cartesian coordinate space as illustrated in Fig. 7. Each dimension represents the domain of one attribute of the keys. Fig. 7 shows a two-dimensional data space with x- and y-intervals of $[0, 1]$. The coordinate space is built on an overlay network which in turn is mapped onto arbitrarily located storage nodes.

The key space is dynamically partitioned among the nodes such that each node is responsible for roughly the same amount of objects. Load-balancing [17] allows to add and remove nodes when the number of objects increases or shrinks or when additional storage space becomes available. This process, called *node join* and

```
1  // checks whether the resp. coordinate of b lies between a and c
2  bool IsBetween(Dim dim, Key start, Key end, Key pos);
3
4  void updateRoutingTable(int dim) {
5    int  i = 1;
6    bool done = false;
7
8    rt[dim][0] = this.Successor[dim];
9
10   while (!done) {
11     Node candidate = rt[dim][i - 1].getPointer(dim, i - 1);
12     if (IsBetween(dim, rt[dim][i - 1].Key, candidate.Key, this.Key)){
13       rt[dim][i] = candidate;
14       i++;
15     } else
16     done = true;
17   }
18 }
```

Figure 8: Calculation of routing pointers for one dimension **`dim`**.

node leave is detailed in Sect. 3.4. New attributes may be added at any time by adding a new dimension to the overlay.

3.1 Overlay

Multi-attributed data is stored in a virtual d-dimensional torus topology. Fig. 7 gives an example of geospatial objects arranged in a 2D data space. One data object, marked '+', is located at coordinate (0.84,0.24). Each box is handled by one node of the overlay, and all together they cover the complete key space. Since the keys are usually not uniformly distributed, the boxes have different sizes to balance the load.

Because of the different sizes, a box may have more than one direct neighbor in each direction. The pointers to all neighbors of a node are stored in a *neighbor list*. SONAR uses the neighbor list only for accessing direct neighbors; the routing to distant nodes is done with *routing tables* (Sect. 3.2).

In the one-dimensional case the successors form a ring topology. Following one successor after the other, the ring can be traversed similar as in Chord [36]. Doing the same in the d-dimensional case does not guarantee to end in the starting node, because the rings are skewed due to different node sizes. There is no need to care, however, because queries never need a full round.

3.2 Routing in the Node Space

The neighbor list is—in principle—sufficient for routing, as long as its links are up-to-date. But due to the lack of far-reaching routing information, the routing

performance of such a system would be $O(\sqrt[d]{N})$ on average and $O(N)$ in the worst case.

For improved routing, SONAR maintains additional *routing tables* in each node which contain pointers to nodes at exponentially increasing distances in each dimension. With a total of $\log N$ routing pointers per node, the average number of hops is reduced to $O(\log N)$. For a proof see [30].

Compared to Chord, which uses a DHT, SONAR does not compute the routing pointers in the key space, but does this in the node space: When Chord jumps over half of the *key entries* in the ring, SONAR jumps over half of the *nodes* in the ring, which is actually the goal of Chord's behavior. To calculate its i^{th} pointer in the routing table, a node looks at its $(i-1)^{th}$ pointer and asks the remote node listed there, for its $(i-1)^{th}$ pointer. In general, the pointers at level i are set to the pointers' neighbors in the next lower level $i-1$. At the lowest level, the pointers refer to the direct successor [30]:

$$pointer_i = \begin{cases} successor & : i = 0 \\ pointer_{i-1}.getPointer(i-1) & : i \neq 0 \end{cases}$$

The shortest pointer leads to the direct successor and each following pointer doubles the distance by combining two shorter pointers. Such a routing table exists for each dimension. Fig. 8 sketches the pointer update algorithm in pseudo code.

Since there may be several neighbors in each direction (ref. Sect. 3.1), we define the one at the center of the respective side as the neighbor which is used for the first (shortest) routing table entry. It is marked by a small tick on the edges in Fig. 7.

Note that our pointer placement algorithm calculates each pointer in constant time $O(1)$, whereas Chord needs $O(\log N)$ for the same operation because Chord does a key-lookup for calculating a pointer, which needs $\log N$ hops.

3.3 Size of the Routing Table

Given that the total number of nodes is not known locally, how do we limit the number of routing table entries to $\log N$? We could estimate the size of the system (like Mercury) with a density function, but that would cause unnecessary effort while still being imprecise. In SONAR, the maximum number $\log N$ of routing table entries is given implicitly by successively filling the routing table with pointers of exponentially increasing length:

> Starting with the neighbor that is adjacent to the center of node n in routing direction s, we insert an additional $pointer_i$ into the table as

```
1       double getDistance(Node a, Node b);
2
3       Node findNextHop(Point target)
4       {
5           Node candidate = this;
6           double distance = getDistance(this, target);
7
8           if (distance == 0.0)
9               return this;  // found target
10
11          for (int d = 0; d < dimensions; d++) {
12              for (int i = 0; i < rt[d].Size; i++) {
13                  double dist = getDistance(rt[d][i], target);
14                  if (dist < distance) {
15                      candidate = rt[d][i];  // new candidate
16                      distance = dist;
17                  }
18              }
19          }
20          Assert(candidate != this);
21          return candidate;
22      }
23
24      Node find(Point target)
25      {
26          Node nextHop = findNextHop(target);
27          if (nextHop == this)
28              return this;
29          else
30              return nextHop.find(target);
31      }
```

Figure 9: Routing to a target.

long as the following equations holds true:

$$pointer_{i-1}[d] < pointer_i[d] < n[d]$$

When this condition fails, it is clear that the pointer would overspan the current node n (i.e. making more than one round) and hence would not be included in the routing table. Fig. 8 shows pseudo-code for the pointer update algorithm and Sect. 3.6 shows its accuracy.

When all routing tables are filled with the appropriate routing information using the code in Fig. 8, the tables can be used to route queries to their target node(s). Just as in other DHTs, we use a greedy routing strategy. In each node the pointer is taken which maximally reduces the euclidean distance to the target (see Fig. 9).

3.4 Handling Churn

When a node wants to join the network, the area covered by an existing node has to be divided into two parts and the responsibilities split over the two nodes. Two things have to be done:

1. Find a suitable target node with a high load in terms of query- or item-load: This can be done by randomly choosing some nodes (as in Cyclon [35]) and selecting a candidate from the list of resulting nodes.

2. Split the data of the selected node and transfer one portion to the new node: Splits are performed parallel to one of the coordinate system axes. Care must be taken to select the 'right' axis: If a dimension is always favored over the others, the number of nodes contacted in a range query may become disproportionately high and leave operations may become more expensive.

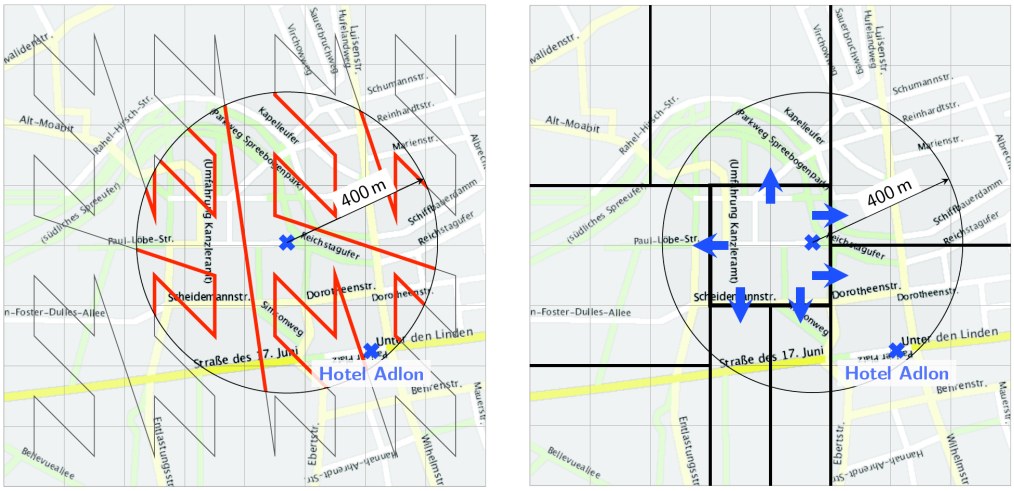
The nodes and their splitting planes form a kd-tree [7]. In contrast to traditional databases, the kd-tree is here not used as an indexing structure, but only for maintaining the topology—similar as in MURK [13]. When a node leaves the system, the occupied space must not be merged with an arbitrary neighbor, but only with 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.

3.5 Range Queries

SONAR supports multi-attributed range queries. Given d intervals over d attribute domains, the range query will return all values between the lower and upper bounds for each domain. Because of their shape, such range queries are named d-dimensional *rectangular* range queries.

In practice, circles or polygons often better fit user requests. Fig. 10 illustrates a two dimensional circular range query where the query is defined by a center and a radius. For this example, we assume a person located in the governmental district of Berlin searching for a hotel. The center is the location of the person and the radius is the acceptable 'walking distance'. In a first step (Fig. 10b) the query is routed to the node responsible for the center of the circle. Thereafter the query is forwarded to all neighbors that partially cover the circle. The query is checked against their local data and the results are returned to the requesting node. Fig. 11 shows the pseudo code for this range query algorithm. To avoid redundant operations, the query is only forwarded to nodes that have not been asked before.

In contrast to overlays with space-filling curves, SONAR just needs to route to the node in the center of the circle. When space-filling curves are used (Fig. 10a), several line segments of the curve may be responsible for the range and for each



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

Figure 10: Circular range query.

segment a complete routing (each in $O(\log N)$) must be performed to retrieve all results.

In SONAR the neighborship is given by the key space and queries may take arbitrary shapes. The query is sent to the center node and from there forwarded to all neighbors which possibly may have a subset of the query range. These nodes are then responsible for the cut-off. A more sophisticated implementation could route to several points on the outline of the shape and start the flooding towards the center.

3.6 Simulation

We used four different data sets to evaluate the performance of SONAR: two with real geographic coordinates and two with synthetically generated objects. For each data set, we did several experiments with varying data sizes. Each experiment was started by partitioning the two-dimensional space into non overlapping rectangular patches so that each patch contains about the same amount of data items. This was done by recursively splitting the patches at alternating sides until the number of data items in the area dropped below a given threshold. Fig. 12 illustrates the four data sets:

a) ***USA*** contains the locations of the 13,509 cities in the United States with a population of at least 500 inhabitants. This data set is taken from the TSPLIB, see http://www.tsp.gatech.edu/.

```
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);
}

void queryRange(Range r, Operation op)
{
    Node center = find(r.Center);
    center.doRangeQuery(r, op, newId());
}
```

Figure 11: Range query algorithm.

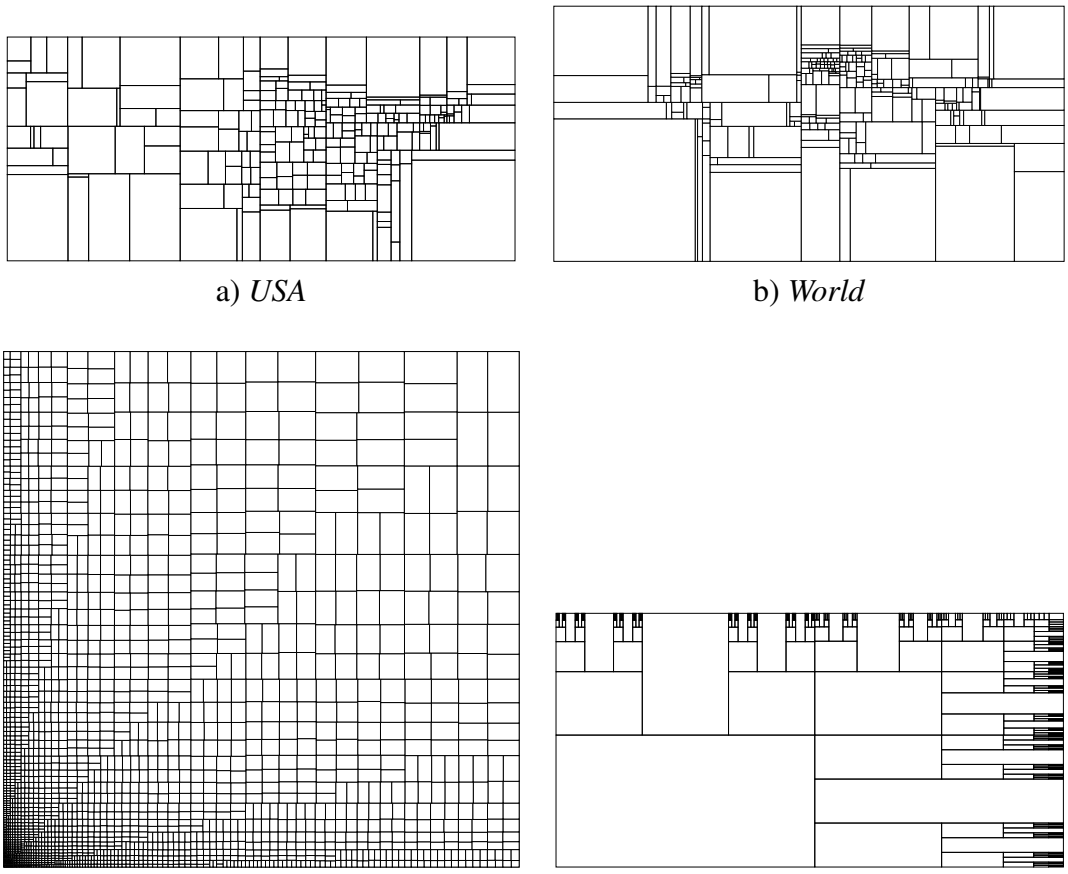
b) ***World*** is also taken from the TSPLIB. It contains the 1,904,711 largest cities in the world. Each city location (longitude and latitude) was mapped onto a doughnut-shaped torus rather than a sphere, because the poles of a sphere would become a bottleneck and the routing direction in the western hemisphere would interfere with that of the eastern hemisphere (southwards vs. northwards).

c) ***Exponential*** was generated by running a random number generator with an exponential distribution and placing the data points on the 2D plane.

d) ***Worst*** is a synthetically constructed worst case pattern that bends the direct routing neighbors to nodes with few successors in the same routing direction. This makes it difficult for SONAR to find a short routing path, because for each dimension i, the routing table holds less than $\log N_i$ pointers (with N_i being the number of keys in this direction only).

   The network was constructed by recursively splitting the rectangular area into four patches: south-west (SW), south-east (SE), north-west (NW), and north-east (NE).

   - The SW part is slightly larger in both directions than the other parts and it is not further subdivided.
   - The NW part is split into three equally-sized vertical slices: The middle one fills the whole space, while the outer ones fill just a bit more



a) *USA* b) *World*

c) *Exponential* d) *Worst*

Figure 12: The four data sets used in the performance evaluation.

than half of the available space, and the remainder is recursively split as before—up to the specified recursion depth.

- The SE part follows the same pattern as NW, but is splitted horizontally.
- For the NE part, the described global pattern with the splitting into four parts is applied recursively until the recursion depth is reached. If the recursion depth is reached, the remaining part is added as a whole. We used a recursion depth of 5 iterations, which lead to 348 rectangles.

The two TSPLIB data sets have been selected because their Zipf-like distribution [38] is typical for many applications. *Exponential* was used to check the routing behavior with a skewed distribution where the boxed have very different sizes and *Worst* is an artificially constructed data pattern that causes a worst case routing behavior.

We did not include results of a uniform data distribution, because it partitions the data space into a regular grid. Each node then has a routing table with exactly $\log N$ entries, that is, $\frac{logN}{i}$ pointers for each dimension $0 < i \leq d$. In this ideal data distribution, SONAR's pointer placement algorithm calculates precisely exponentially spaced pointers, which results obviously in an optimal logarithmic routing performance.

3.6.1 Average Routing Performance

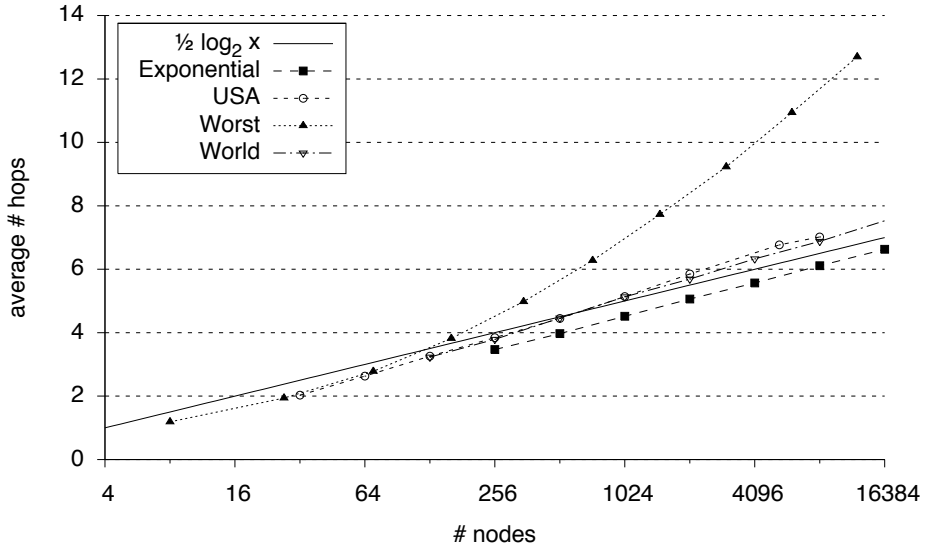


Figure 13: SONAR routing performance on networks of various sizes for 4 data sets.

To evaluate SONAR's routing performance we conducted an all-to-all search by issuing a query from each node to the center of each other node, resulting in a total of N^2 queries.

Fig. 13 shows the average number of routing hops for various network sizes. As can be seen, the curves of *USA* and *World* lie within the expected average of $0.5 \log N$ hops depicted by the straight line, which indicates that the scheme works well with Zipf-distributed [38] real-world data.

The results for the *Exponential* data set further support the claimed $\log N$ routing performance, despite the greater variation in the box sizes which makes the pointer placement more difficult. Only the *Worst* data set shows a degradation to $\log^2 N$ for the larger instances. In Sect. 3.6.4 we present a monitoring scheme that detects and eliminates such anomalies without using global information.

3.6.2 Routing Load per Node

In another set of experiments, we checked the distribution of the routing load over all participating nodes. The larger the spectrum of node sizes, so our conjecture, the higher the routing load at those nodes that are responsible for a greater part of the data space, because they are likely to have more incoming routing pointers.

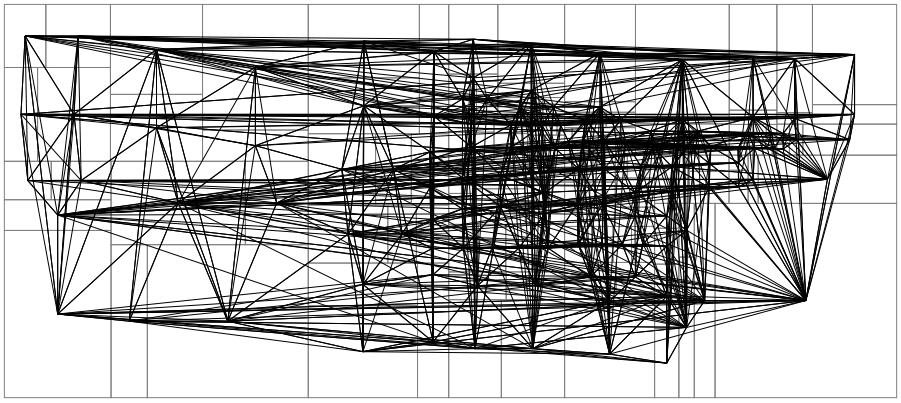


Figure 14: All routing hops of an all-to-all search (*USA*, $N = 128$)

Fig. 14 depicts the *USA* data set with a network of 128 nodes. Each box is handled by one node and contains about the same number of cities. The lines show all routing hops (including wrap-around hops) performed in an all-to-all search. We counted a total of 50,824 hops, resulting in an average of $50,824/16,256 = 3.1$ hops per query, which is slightly better than the expected $1/2 \log 128 = 3.5$ hops.

A closer inspection of the results in Fig. 14 reveals that some nodes have a much higher indegree than others, see for example the 'Florida Node' at the right bottom. This may result in a higher CPU load, because nodes with many incoming routing pointers will receive more forwarding requests from other nodes. To further investigate this aspect, we plotted the distribution of the node indegrees in Fig. 15. It can be seen that the indegree of most nodes is near the expected value of $\log 128 = 7$. Only seven nodes have an indegree > 14. With an indegree of 29, the node covering Florida is the largest.

Another interesting aspect is the frequency of the use of long versus short pointers. Ideally, a query would start with a long-distance hop and then successively reduce the hop distance until the target is found. Hops to direct neighbors should only occur at the very end of the lookup. A closer look at our experimental results reveals that 78% of all hops were indeed long-distance hops and only 22% were direct neighbor hops. More precisely: From the 3.13 hops, that were on the

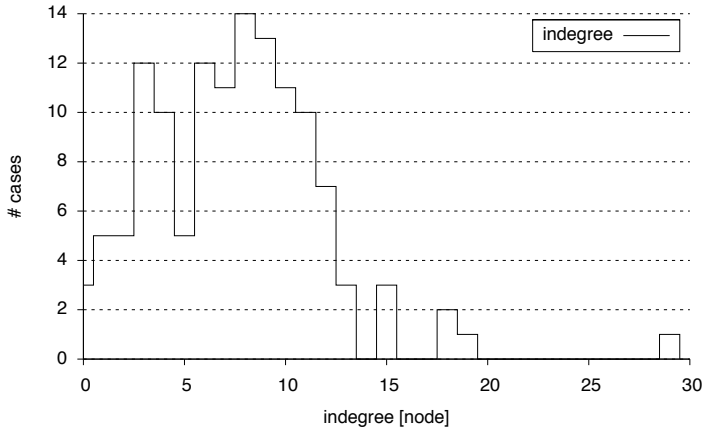


Figure 15: Node indegree distribution for Fig. 14 (*USA*, $N = 128$)

average required to find the target in our all-to-all search, only 0.68 hops were direct neighbor hops, while the other 2.45 hops were long-distance hops. This is another indication of SONAR's lookup efficiency.

3.6.3 Accuracy of the Routing Table Information

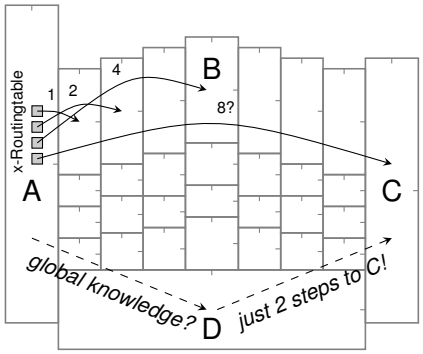


Figure 16: Erroneous pointer lengths due to lack of global information: Node A believes that its routing pointers span distances of 1, 2, 4, and 8 hops respectively. From a global perspective, however, A's longest pointer spans only 2 hops, see the arc below.

SONAR has only local information for building the routing tables. Each node computes its pointers by recursively asking a remote node for its pointer infor-

mation. This could lead to inaccuracies in the long pointers, because they are recursively constructed using local information of other shorter pointers.

Fig. 16 illustrates a situation, where local information misleads the pointer update algorithm to include pointers in the routing table which are too short. Each of the shown pointers is built by recursively concatenating the next shorter local pointers. This causes pointers of a believed length of 1, 2, 4, and 8 to be inserted into the routing table of node A. In reality, however, the longest pointer spans only two hops instead of eight. A global observer with an optimal routing strategy would just make two hops via node D to arrive at node C. This information is, unfortunately, not available in node A.

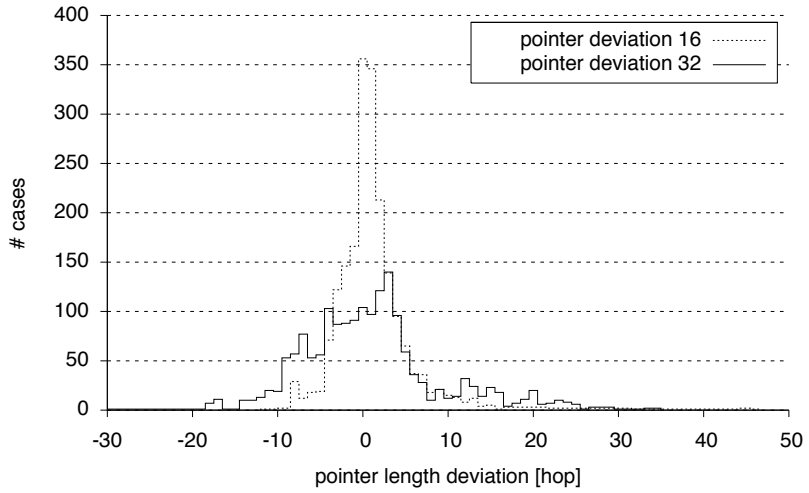


Figure 17: Deviation of pointer lengths due to local information (*World*, $N = 1024$). The measured lengths are centered around their expected length of 16 resp. 32.

To check whether this situation occurs in practice, we plotted the actual pointer lengths versus their expected values. Fig. 17 and 18 show the deviation of the pointer lengths for the *World* and *Worst* data sets. The expected pointer length is always a power of two and can be derived from the pointer update algorithm. For the actual length we measured the number of hops an algorithm with global information would need to get from one node to the other by only moving east- or northwards as done by our greedy routing.

In both Figures, the large majority of the pointers is set correctly at their expected values, while only a small number of routing entries deviates. Since no direction is preferred, the deviations in both directions compensate each other. Interestingly, this also applies to the *Worst* data set with a slight tendency towards

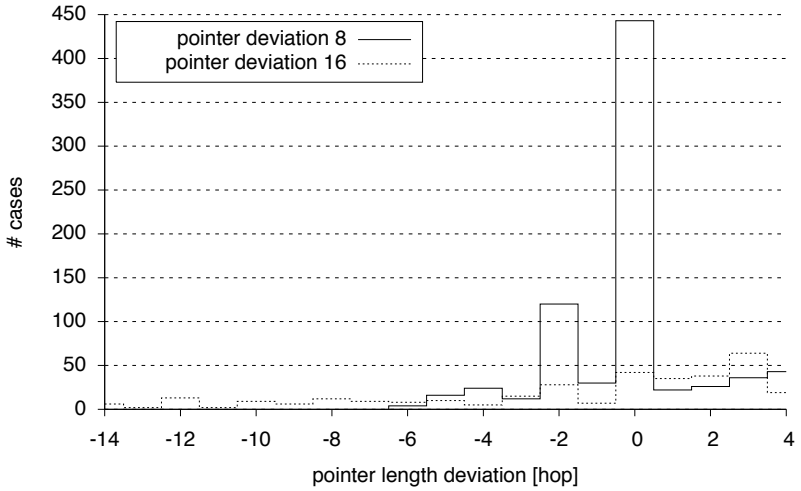


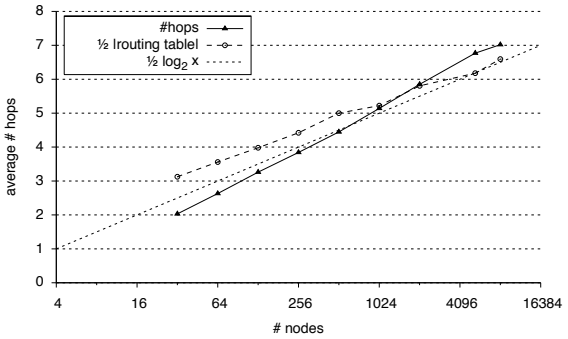
Figure 18: Deviation of pointer lengths due to local information (*Worst*, $N = 1490$). The measured lengths are centered around their expected length of 8 resp. 16.

shorter pointers. This indicates that the $\log^2 N$ routing behavior of *Worst* is not caused by incorrect pointer lengths.

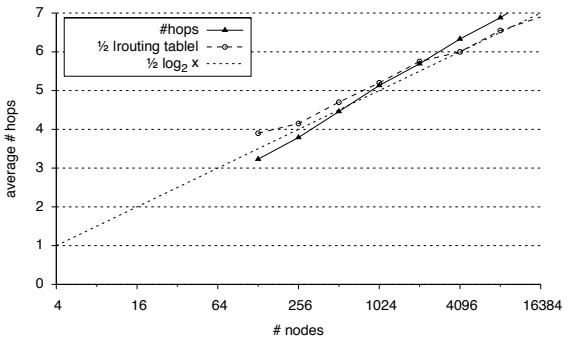
3.6.4 Worst Case Behavior

SONAR exhibits a logarithmic routing performance in all but one of the four analyzed data sets. Only *Worst* needs $O(\log^2 N)$ hops on the average. Even though this artificially constructed data pattern is very unlikely to occur in practice, it must be dealt with. We conjecture that the degradation is caused by missing information in the routing table. This could happen if the table does not cover the whole key space with exponentially spaced pointers or if the table is not filled with $\log N$ pointers.

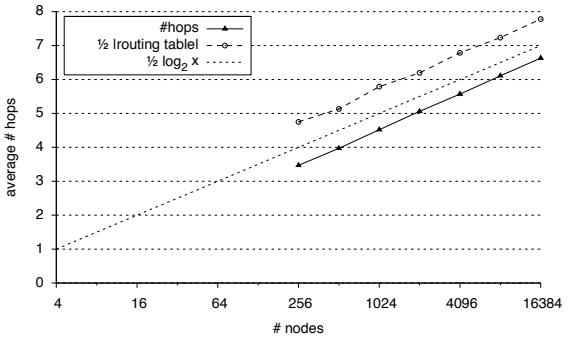
a) USA



b) World



c) Exponential



d) Worst

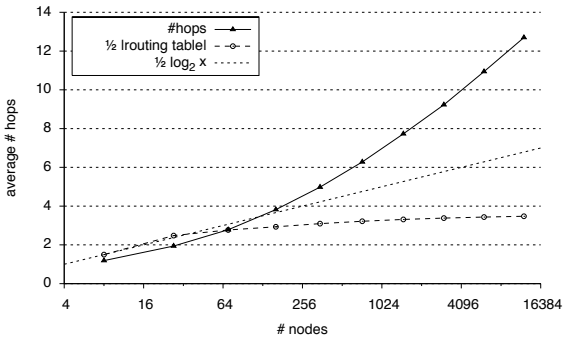


Figure 19: SONAR performance on four data sets for various network sizes. The straight line shows the measured number of routing hops and the dotted line shows the number of hops predicted from the size of the routing table.

Fig. 19 shows the measured number of routing hops (straight lines) and the expected hops (dotted lines) for all four data sets. The expected hop number was computed by counting for all nodes the sizes of the routing table and dividing it by two. As can be seen, SONAR needs fewer hops than expected in the smaller networks of *USA* and *World*, while it needs slightly more for the larger networks. More pronounced is the gain in the *Exponential* data set, where SONAR saves one hop at all network sizes.

This situation is quite different for the *Worst* data set. Here, the average hop number (top curve) increases as the networks get larger. The degrade in the routing performance is paralleled by a reduction in the routing table size (bottom curve). The routing table does not grow with increasing network size because we constructed the *Worst* data set in such a way, that SONAR is not able to find enough pointers to fill its routing table with $\log N$ exponentially spaced targets.

Self-improving routing tables. Two steps are necessary to fix the problem: First, a monitoring scheme must be employed to detect any degradation in the routing performance from the expected $\log N$-routing, and then measures for improvements must be taken. Since global information is not available in P2P systems, both steps must be done with local information only.

The monitoring is done by piggybacking the hop count on top of queries. Several hop counts are collected and averaged. From the construction of the routing table with exponentially spaced pointers (Sect. 3.2), we expect it to have $\log N$ entries. Comparing this to the observed hop number, which should be $0.5 \log N$ on average, we are able to assess the quality of the routing—based on local information.

When a routing degradation has been detected, the mapping of the key space to the nodes must be changed. This can be done, for example, by augmenting the load metric with a factor that takes the routing load (resp. the node indegree) into account. Nodes with a large indegree should split their key space to achieve a better workload balance and to improve the routing table.

3.7 Summary

This chapter shows that SONAR is a scalable, distributed key-value store, which efficiently supports supports lookups for individual keys as well as range queries for multi-dimensional keys. Similar to Chord$^{\#}$, SONAR can be used to efficiently implement the registry for a content-based pub/sub. Chap. 4 will show how this is implemented in detail.

4 Publish/Subscribe System Architecture

For the delivery of notifications, we will rely on standard multi-cast algorithms, e.g. [14]. The overlay network could be used to deliver the notifications to nodes which are nearby the subscribers and these nodes would then deliver the messages. By incorporating more overlay nodes into the delivery, the load can be spread more evenly and in some instances firewall problems can be avoided.

The range queries available in Chord$^{\#}$ and SONAR provide functions which are missing in most structured overlay networks. In the context of pub/sub systems, they allow to implement the backend in a truly scalable way without sacrificing the power of the supported topics.

4.1 Topic-Based Publish/Subscribe System

Many pub/sub systems support only topic-based subscriptions, where nodes can publish events and subscribe to individual topics identified by keywords. Topic-based pub/sub can be easily mapped to DHTs, by using the keywords/topics as keys and storing for each keyword the list of subscribers. The following table shows an example:

Key	Value
NodeFailures	[node1, node4, node5]
NewNodes	[node3, node6, node7]

For the XtreemOS project, the topic-based pub/sub service is implemented on Chord$^{\#}$ [30] and standard multicast algorithms [14]. To improve the event delivery [14] the overlay structure can be exploited. We demonstrated the scalability of this approach in [30, 29, 33].

4.2 Publish/Subscribe System for Hierarchical Topics

In the last section, we introduced topic-based pub/sub systems. They can be extended to hierarchical topics, where the topics form a tree (see Fig.20). Here, users may subscribe to individual topics or sub-trees.

We again start from range queries supporting DHTs, which usually store data points and perform lookups for all items in a given range. E.g., Fig. 21 shows several data points and two range queries in a two-dimensional data space. Notice that it is also possible to support applications which work the other way round: Storing ranges and looking for all ranges which include a given point. In the following we will show that this can be exploited to build hierarchical databases on top of the DHT.

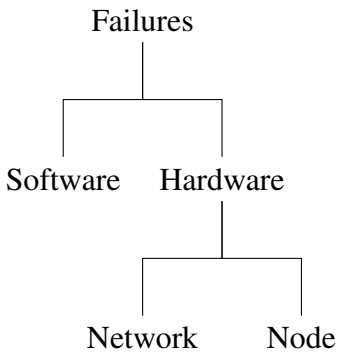


Figure 20: Example Hierarchy of Topics

4.2.1 Trees vs. Range Queries

We assume that all nodes in the tree have a non-unique name and $\leq$ denotes some lexicographical order function on the name space. Furthermore, two special letters **/** and ω are needed which are smaller resp. larger than all other letters in the alphabet. Each node can be uniquely identified by the path from the root to the node with individual path elements separated by /, e.g. /Failures/Hardware/Networkω.

Subscriptions to individual topics are handled in the same way as described in the last section. Subscription to sub-trees are stored as ranges, e.g. "all hardware failures" is equivalent to the range [/Failures/Hardware, /Failures/Hardwareω]. The definition of **/** and ω in this case guarantees that all hardware failure topics lie in the given range.

This encoding enables a direct mapping to Chord$^\#$'s namespace because both order keys using lexicographical order and queries like [/Failures/Hardware, /Failures/Hardwareω] (all hardware failures) can be executed efficiently.

The step from topic-based to hierarchical subscriptions seems to be rather small when explaining the differences with Chord$^\#$ as a backend. However, the differences are tremendous. Users can organize related topics into groups and browse through the topics to find the relevant ones.

4.3 Content-Based Pub/Sub

Finally, hierarchical topics can be extended to a content-based pub/sub system, where individual events are described by a list of key-value pairs, like, e.g.:

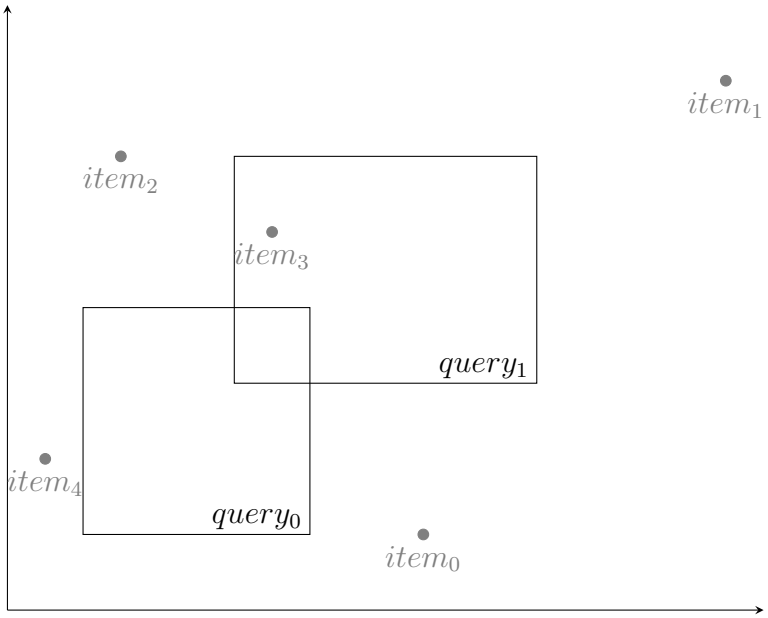


Figure 21: Range Queries

Property	Value
Date	2007-10-17
Temperature	17 degrees celsius
Height	50m
Latitude	52.516222
Longitude	13.377417

From the point of view of the DHT, content-based pub/sub systems can be seen as a generalization of hierarchical topics. The subscriptions are again ranges and the events are points. Both have to be mapped to a multi-dimensional data space. So the subscriptions are hypercuboids in this high-dimensional space.

5 Conclusion

5.1 Simulation Results

Chord$^{\#}$ is a DHT with support for one-dimensional range queries and explicit load-balancing. Load-balancing is based on [17] and supports arbitrary load metrics, like memory consumption, cpu load, network bandwidth, etc.

SONAR is a generalization of Chord$^{\#}$ and additionally supports multi-dimensional range queries. The simulations were conducted on a variety of different data distributions. In all cases except one the lookup performance was in $O(\log N)$. The counter example is specially designed artificial case, whereas the other examples were derived from real world data.

For both system, we showed that lookups can be performed with logarithmic effort – for the former even in face of high churn situations. The capacity of the whole system scales therefore with $\frac{N}{\log N}$.

5.2 Ongoing work

A prototype of the described system was already tested on PlanetLab, a testbed for distributed applications. For the next deliverable (D3.2.7: Reproducible evaluation of a scalable publish/subscribe system (M30)) we will extend the prototype to a full-fledged service building on the results of the simulations and experiments on PlanetLab.

The outward facing interface will be developed in tight collaboration with WP3.1 and WP3.4. WP3.1 will be the lead partner as it will include the application interface to other applications. The collaboration with WP3.4 will be focused on performance, scalability, and functionality. During the development phase WP3.4 will be the first user of the service.

5.3 Dissemination

The following three papers have been published as part of this deliverable:

A Structured Overlay for Multi-dimensional Range Queries.
Thorsten Schütt, Florian Schintke, and Alexander Reinefeld.
In *Proceedings of the 13th International Euro-Par Conference on Parallel and Distributed Computing*

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 pe rform 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.

P2P Routing of Range Queries in Skewed Multidimensional Data.
Alexander Reinefeld, Florian Schintke, and Thorsten Schütt.
Zuse Institute Berlin Technical Report

Abstract: We present a middleware to store multidimensional data sets on Internet-scale distributed systems and to efficiently perform range queries on them. Our structured overlay network *SONAR (Structured Overlay Network with Arbitrary Range queries)* puts keys which are adjacent in the key space on logically adjacent nodes in the overlay and is thereby able to process multidimensional range queries with a single logarithmic data lookup and local forwarding. The specified ranges may have arbitrary shapes like rectangles, circles, spheres or polygons.

Empirical results demonstrate the routing performance of SONAR on several data sets, ranging from real-world data to artificially constructed worst case distributions. We study the quality of SONAR's routing structure which is based on local knowledge only and measure the indegree of the overlay nodes to find potential hot spots in the overlay. We show that SONAR's routing table is self-adjusting, even under extreme situations, keeping always a maximum of $\lceil \log N \rceil$ routing entries.

Range Queries on Structured Overlay Networks.
Thorsten Schütt, Florian Schintke, and Alexander Reinefeld.
In *Computer Communications: Foundations of P2P*.

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.

References

[1] K. Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. *CoopIS*, Oct. 2001.

[2] 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.

[3] L. Alima, S. El-Ansary, P. Brand and S. Haridi. DKS(N,k,f): A family of Low-Communication, Scalable and Fault-tolerant Infrastructures for P2P applications. *Workshop on Global and P2P Computing*, CCGRID 2003, May 2003.

[4] A. Andrzejak, and Z. Xu. Scalable, Efficient Range Queries for Grid Information Services. P2P 2002, 2002.

[5] J. Aspnes and G. Shah. Skip graphs. *SODA*, Jan. 2003.

[6] F. Banaei-Kashani and C. Shahabi. SWAM: a family of access methods for similarity-search in peer-to-peer data networks. *CIKM*, Nov. 2004.

[7] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, Vol. 18, No. 9, 1975.

[8] A. Bharambe, M. Agrawal and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. *ACM SIGCOMM 2004*, Aug. 2004.

[9] 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.

[10] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many Faces of Publish/Subscribe. ACM Computing Surveys, Vol. 35, No. 2, pp. 114-131, June 2003.

[11] V. Gaede, and O. Günther. Multidimensional access methods. ACM Computing Surveys, 30 (2), 1998.

[12] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online Balancing of Range-Partitioned Data with Applications to Peer-t o-Peer Systems. *VLDB 2004.*

[13] P. Ganesan, B. Yang, and H. Garcia-Molina. One Torus to Rule Them All: Multi-dimensional Queries in P2P Systems. *WebDB 2004.*

[14] A. Ghodsi. Distributed k-ary System: Algorithms for Distributed Hash Tables. PhD Thesis, Oct. 2006.

[15] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating Latency between Arbitrary Internet End Hosts. Proceedings of the 2nd Usenix/ACM SIGCOMM Internet Measurement Workshop (IMW) 2002, Marseille, France, November 2002.

[16] 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. *29th Annual ACM Sympos. Theory of Comp.*, May 1997.

[17] D. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. *IPTPS 2004*, Feb. 2004.

[18] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. Proc. 32nd ACM Symposium on Theory of Computing, 2000.

[19] J. Li, J. Stribling, T. M. Gil, R. Morris, and M.F. Kaashoek. Comparing the performance of distributed hash tables under churn. *IPTPS 2004*, Feb. 2004.

[20] J. Li, J. Stribling, R. Morris, M.F. Kaashoek, and T. M. Gil. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. *Infocom*, May. 2005.

[21] P. Maymounkov and D. Maziéres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. *IPTPS*, March 2002.

[22] M. Naor and U. Wieder. Novel Architectures for P2P Applications: the Continuous-Discrete Approach. *SPAA 2003.*

[23] 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.

[24] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, June 1990.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. *ACM SIGCOMM 2001*, Aug. 2001.

[26] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. *Proceedings of the USENIX Annual Technical Conference*, Jun. 2004.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *Middleware*, Nov. 2001.

[28] C. Schmidt and M. Parashar. Enabling flexible queries with guarantees in P2P systems. *IEEE Internet Computing*, 19-26, May/June 2004.

[29] T. Schütt, F. Schintke and A. Reinefeld. Chord$^{\#}$: Structured Overlay Network for Non-Uniform Load-Distribution. Zuse Institute Berlin, Technical Report, August 2005.

[30] T. Schütt, F. Schintke and A. Reinefeld. Structured Overlay without Consistent Hashing: Empirical Results. *GP2PC'06*, May 2006.

[31] T. Schütt, F. Schintke and A. Reinefeld. A Structured Overlay for Multi-Dimensional Range Queries. *Europar*, Aug. 2007.

[32] A. Reinefeld, F. Schintke and T. Schütt. P2P Routing of Range Queries in Skewed Multidimensional Data Sets. Zuse Institute Berlin Technical Report, Aug. 2007.

[33] T. Schütt, Florian Schintke, and Alexander Reinefeld. Range Queries on Structured Overlay Networks. Computer Communications: Foundations of P2P.

[34] Y. Shu, B. Chin Ooi, K. Tan, A. Zhou. Supporting Multi-dimensional Range queries in Peer-to-Peer Systems. P2P'05, Sep. 2005.

[35] S. Voulgaris, D. Gavidia and M. van Steen: CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management*, 13:197 217(21), Jun. 2005.

[36] 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.

[37] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, Vol. 22, No. 1, Jan. 2004.

[38] G. Zipf. Relative Frequency as a Determinant of Phonetic Change. Reprinted from *Harvard Studies in Classical Philiology*, 1929.