Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

THORSTEN SCHÜTT, FLORIAN SCHINTKE,
ALEXANDER REINEFELD

Chord$^{\#}$: Structured Overlay Network for Non-Uniform Load-Distribution

ZIB-Report 05-40 (August 2005)

Chord$^{\#}$: Structured Overlay Network for Non-Uniform Load-Distribution

Thorsten Schütt, Florian Schintke, Alexander Reinefeld
Zuse Institute Berlin (ZIB)
{schuett,schintke,reinefeld}@zib.de

August 31, 2005

Abstract

Data lookup is a fundamental problem in peer-to-peer systems: Given a key, find the node that stores the associated object. Chord and other P2P algorithms use distributed hash tables (DHTs) to distribute the keys and nodes evenly across a logical ring. Using an efficient routing strategy, DHTs provide a routing performance of $O(\log N)$ in networks of N nodes.

While the routing performance has been shown to be optimal, the uniform key distribution makes it impossible for DHTs to support range queries. For range queries, consecutive keys must be stored on logically neighboring nodes.

In this paper, we present an enhancement of Chord that eliminates the hash function while keeping the same routing performance. The resulting algorithm, named Chord$^{\#}$, provides a richer functionality while maintaining the same complexity. In addition to Chord, Chord$^{\#}$ adapts to load imbalance.

1 Introduction

Peer-to-peer (P2P) systems have properties that make them ideally suited for building large distributed systems: They provide scalability, fault tolerance and, to a certain extent, local self-organization.

One of the most interesting research problems in P2P systems is the *data lookup problem*: How to find objects in a large P2P system in an efficient and scalable manner? Many schemes for routing lookup queries have been proposed [2]: Network flooding (Gnutella), superpeer networks (KaZaA, FastTrack), unstructured routing with caching for anonymous data management (Freenet), skip lists

(Chord, DKS), tree-like data structures (Kademlia, Pastry, Tapestry), multi-dimensional coordinate spaces (CAN), and many more.

State-of-the-art systems employ a two-stage data lookup: First a hash function is used to convert the search key into a numeric value and then one of the above mentioned routing methods is applied to retrieve the object or its metadata. The first step was considered necessary, because the DHT evenly distributes the key space over all nodes, thereby avoiding load imbalances, but preventing range queries.

We present a lookup method that does the routing without DHTs. It has the same favorable properties and performance as Chord [12] but allows more expressive queries. We describe our new method along the Chord protocol, but it can analogously be applied to other P2P systems that are based on DHTs, like DKS(N,k,f) [1], without influencing their additional properties.

In the remainder of this paper, we briefly recall the purpose of lookup services and review the Chord protocol. Thereafter, we present our Chord$^{\#}$ algorithm, which is simple to implement, and discuss its performance, both analytically and empirically.

2 Lookup Services

One major challenge in building large distributed systems is to devise a *lookup service* that scales over millions of nodes and allows to add or remove nodes at any time without disrupting or compromising the service. We distinguish three kinds of lookup services: name, directory, and discovery services.

Name services provide a mapping between names and attributes. They are used by clients to obtain attributes of resources, services or objects when given their name. The named entities can be of many types: They may hold the addresses and other details of users, computers, networks, or objects. The Internet DNS (domain name service) is an example of a distributed name service. It maps domain names to IP numbers. It does so by looking up the domain names in a hierarchy of tables stored in routers or computers. If one DNS server does not know how to translate a particular domain name, it asks the server in the next higher layer of the DNS hierarchy, and so on, until the correct IP address is returned. Thereby, the DNS system creates an overlay network for its own metadata management. Caching schemes have been introduced for faster access with less network traffic, but nonetheless the principle behind DNS is a simple hierarchical scheme.

Directory services are inverse to name services: Given a list of attributes, a directory search returns name(s) or address(es) of that item(s). With a directory search, it is possible to answer queries like “Give me all PDF articles stored on

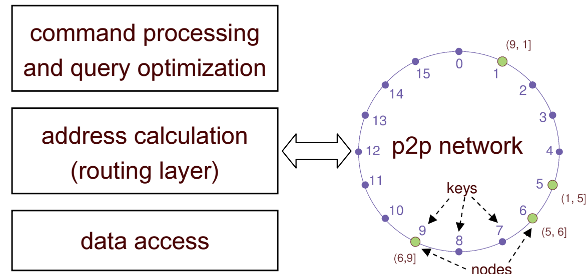


Figure 1: Architectural layout of a lookup service

this computer that have been published by Donald Knuth before 1995". Directory services resemble the yellow pages in the telephone directory. Microsoft's Active Directory Services, the X.500 and its lean variant LDAP are typical examples of directory services.

Discovery services go one step further. They allow services to register themselves in spontaneous networks for later lookup. The Java-based Jini, the UDDI web service, and the Globus MDS-2 are typical examples of discovery services that provide standard interfaces and schemas for registering and finding services in distributed systems.

General Layout of a Lookup Service. Fig. 1 depicts the general structure of a lookup service [5]. The *command processing and query optimization* shown at the top accepts user commands. Basic file access commands like 'ls /usr/etc' or 'cat file' are directly forwarded to the next lower level, while more difficult commands like a query for Knuth's papers published before 1995 require some pre-processing in the query optimizer.

The *address calculation* or *routing layer* between the command processing and the data access is needed in distributed systems for finding the nodes holding the required data. In DNS, for example, the routing is done by a lookup in a network router table and in Globus by a service call to MDS. In P2P systems – the topic of our paper – the routing can be done with Chord$^{\#}$ on a ring-like overlay network as shown in Fig. 1.

The *data access* is responsible for the actual file access. It accesses the node specified by the router and returns its contents to the user.

Lookup in P2P Systems. P2P systems blur the distinction between the client and server roles. When a client node enters the P2P system, it also contributes

server capacities to the network. This makes the system scalable even in very large environments with a dynamically changing number of nodes.

Many current P2P systems use *distributed hash tables (DHTs)*, which are a variant of consistent hashing [6]. A DHT maps the keys to a ring in a load-balanced way. Any publisher that wants to publish a named object must first apply a hash function to convert the name to a numeric value and then calls `lookup(hash (key))` which yields the network location of the node currently responsible for the key. Finding objects works in the same way, a consumer who wants to retrieve that object also applies the hash function to the key of that object, then calls `lookup(hash(key))` and thereafter retrieves the object from the resulting node.

Nodes that join the system are also placed in the logical ring. Their position is determined by applying the hash function to a unique id of the node. The uniform distribution of nodes and keys is determined by the hash function and cannot be influenced at runtime.

Performance Metrics. We distinguish four kinds of performance metrics.

The *routing performance* is the average number of network hops traversed for finding objects.

The *entry load* denotes the number of key-value pairs each node is responsible for. In Chord, the DHT distributes the entry load uniformly over all participating nodes.

The *query load* denotes the average number of queries that are processed by a node in a given time. It depends on the entry load and the distribution of the queries. In practice, the query load is not uniformly distributed, but some popular keys are retrieved more often than the large majority. With Chord, all queries for a popular item [11, 13] are served by a single node only. Our approach allows to distribute the nodes according to any load demand.

The *routing load* denotes the amount of routing operations performed by a node in a given time interval. It is affected by the popularity of some well-known servers which are used by many users as access points for submitting their queries. Imbalances in the routing load can be reduced in Chord and Chord$^{\#}$ likewise by introducing an additional overlay network.

3 Traditional Chord

In Chord [2, 12], each peer is responsible for the management of a certain fraction of the data. The peers are organized in a logical ring topology. Each retrieval operation is forwarded to a node that is nearer to the location until the location is

found. As shown for node 0 in the left part of Fig. 2, each node holds a *finger table* containing the addresses of the node halfway, quarter-way, $\frac{1}{8}$-way, $\frac{1}{16}$-way, $\ldots$, around the ring. When a node receives a query, it forwards it to the node in its finger table with the highest ID not exceeding `hash(key)`. This way, the distance is reduced in each step by a power of two, resulting in $\log(N)$ hops in networks of N nodes.

In a dynamic network, the finger tables may become outdated whenever nodes enter or leave the system. Still, the system is able to locate all keys. A newly entering node n inserts itself into the ring by asking an arbitrary node to look up n's key. It then sets its own successor pointer and updates the predecessors successor list. The rest of the finger table entries can be either copied from its predecessor or updated later on during the routing. It is easy to see that even when the finger list is out-of-date or corrupt, Chord still makes progress – just a little bit slower. If all finger entries should fail, each Chord node has additional r pointers to its r immediate successors that can be used for routing.

The pointers are calculated in the key-space rather than in the node-space. The pointers refer to the nearest node with a greater id in the ring (see Fig. 2). The distinction between key- and node-space is not necessary for Chord, because keys and nodes are randomly distributed across the ring by the hash function [6].

4 Chord$^{\#}$

The key idea of Chord$^{\#}$ is to substitute Chord's hash function by a key-order-preserving function and adjusting the finger tables accordingly.

In Chord, the DHT ensures that the keys *and* nodes are evenly distributed over the ring. This allows Chord to compute the placement of its pointers in the key space rather than the node space. The entries in the finger table cross $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ \ldots, \frac{1}{2^m}$ of the keys (Fig. 2), thereby guaranteeing a routing performance of $O(\log N)$.

In Chord$^{\#}$, we eliminated the hash function to allow range queries and active load-balancing for various criteria, like changing the query or entry load. As a result, the keys are not uniformly distributed over the key space but follow some density function. In order to maintain the same logarithmic routing effort as Chord, we need to compute the pointers in such a way that they cross $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ \ldots, \frac{1}{2^m}$ of the *nodes* in the ring.

Fig. 2 shows what would happens when the standard Chord pointer placement algorithm would be used in a ring without evenly distributed nodes. As Chord calculates the pointees independently of the node distribution, several pointers would point to the same node. This reduces the effective size of the routing table and

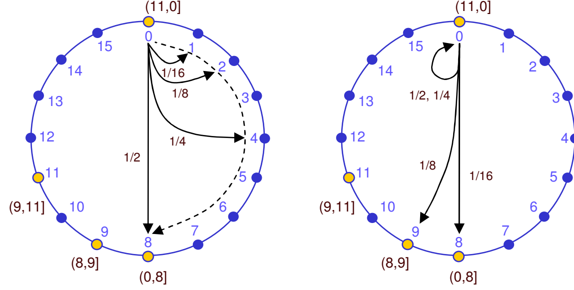


Figure 2: Pointers in Chord (left) vs. pointers in Chord$^\#$ (right)

thereby the routing performance. Chord$^\#$ on the other hand guarantees almost perfect usage of the routing table. The $\frac{1}{16}$ and $\frac{1}{8}$ pointers point to the correct nodes, the $\frac{1}{4}$th and $\frac{1}{2}$nd pointers are useless in this scenario as there are only 2^2 nodes in the system, which needs only 2 pointers. (Note: the names of the pointers are based on Chords scheme.)

4.1 Handling Non-Uniform Distributions

Assuming a ring with equally loaded nodes which are not evenly distributed across the ring, because the keys are no longer evenly distributed, we now investigate the distribution of nodes across the ring. To describe the node distribution we define the density function $d(x)$ over the key space. It gives for each point x in the key space the reciprocal of the width of the corresponding interval.

As Chord is based on consistent hashing the following theorem holds true [6].

Theorem 1 (Consistent Hashing): *For any set of N nodes and K keys, with high probability [12]:*

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).*

For Chord 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 $K = 2^m$).

Lemma 1 *The integral over $d(x)$ equals the number of nodes in a range a to b. 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 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 one 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 one, as expected.

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

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

∎

Lemma 1 can be used, e.g., to predict the number of nodes in the system $\tilde{N}$ having an approximation of $d(x)$, $\tilde{d}(x)$. Nodes can compare $\frac{1}{2}\log(\tilde{N})$ with their observed average routing performance to evaluate the quality of their $\tilde{d}(x)$ and take actions to improve it. For dynamic systems it is unreasonable to assume exact knowledge of d.

4.2 Pointer Placement Algorithm

Both, Chord and Chord$^{\#}$ use logarithmically placed pointers, so that searching is done in $O(\log N)$. Chord, in contrast to our scheme, computes the placement of its pointers 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 takes with *high probability* $O(\log N)$ steps. In regions with less than average sized intervals ($d(x) \gg \frac{N}{K}$) the routing performance degrades.

In a node n, Chord places the pointers p_k according to the following scheme:

$$p_k = (n + 2^{k-1}) \bmod 2^m, 1 \leq k \leq m \qquad (1)$$

Using our integral approach from Lemma 1 and the density function $d(x)$, we develop an equivalent pointer placement algorithm as follows. First, we take a look at the farthest pointer p_m. It points to $n + 2^{m-1}$ when the complete 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 pointer links to the $\frac{N}{2}$-th node to the right with *high probability* due to the consistent hashing theorem.

With equation 1 it is now possible to *exactly* predict one key p_m which is stored on the $\frac{N}{2}$-th node to the right.

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

Other pointers to the $\frac{N}{4}$-th, ..., $\frac{N}{2^i}$-th node can be calculated accordingly. Pointers that wrap around must be handled separately.

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

Theorem 2 (Chord Pointer Placement Algorithm): *For Chord the following two pointer placement algorithms are equivalent.*

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

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

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

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

$$p_i = n + 2^{i-1}$$

■

Chord balances the entry load with its hash function which uniformly distributes the key-value pairs over the nodes. In Chord$^{\#}$ we lost this ability by substituting the hash function with the identity. Therefore, we need to explicitly implement load-balancing between nodes.

So overall we gained range queries and kept most of the features of the traditional Chord. But the load-balancing gets a little bit more complicated and we still have to find a method to predict $d(x)$. For the former we refer to existing mechanisms like [3] and the latter will be described in the next section.

Pointer Placement in Chord$^{\#}$. In the following, we will analyze the pointer placement algorithm (ref. Theorem 2) in more detail.

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

First we split the integral into two by introducing an arbitrary point X between n and p_i so that the value of both integrals is the same:

$$\int_{n}^{X} d(x)\,dx + \int_{X}^{p_i} d(x)\,dx = \frac{2^{i-2}}{2^m}N + \frac{2^{i-2}}{2^m}N$$

We further split this equation into two and assume that the value of both integrals is the same:

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

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

In Eq. 2, the only unknown is X but comparing it to Theorem 2, we see that $X = p_{i-1}$. Applying Theorem 2 to Eq. 3 again we get

$$\int_{p_{i-1}}^{p_i} d(x)\,dx = 2^{i-2-m+k}, 1 \leq i \leq m$$

which is equivalent to

$$p_i = p_{{i-1}_{i-1}} \tag{4}$$

So, instead of approximating $d(x)$ for the whole range between n and p_i, we split the integral into two parts and treat them separately. The integral from n to p_{i-1} is equivalent to the calculation of the pointer p_{i-1}. The remaining formula is equivalent to the calculation of the $i-1$-th pointer of the node at p_{i-1}.

We thereby derived a recursive formula for placing the pointers where the recursion ends with the successor of the current node, which makes the density function $d(x)$ unnecessary in practice.

Theorem 3 (Chord$^{\#}$ Pointer Placement Algorithm): *A pointer placement algorithm, that allows range queries and gets rid of the density function.*

$$pointer_i = \begin{cases} direct_successor & : \quad i = 0 \\ pointer_{i-1}.pointer_{i-1} & : \quad i \neq 0 \end{cases}$$

So, to calculate the 2nd (i^{th}) pointer, a node simply asks his neighboring node ($i - 1^{th}$ pointer) for his 1st ($i - 1^{th}$) pointer. In general, the pointer of one level is set to the neighbor's neighbor on the next lower level. On the lowest level, the pointer references to the direct neighbor.

Note that the density function is completely eliminated in this pointer placement algorithm – it is never computed! The density function is only used in our theoretical analysis to prove the correctness of the pointer placement algorithm.

For dynamic systems the pointers have to be updated from time to time. However in Sect. 5 we show that the routing performance degrades gracefully. Taking into account that the update of one pointer needs only one network-hop, the maintenance overhead for dynamic systems is comparable to other Chord-based systems. Node join and leave operations are done exactly the same way as in Chord [12].

5 Empirical Results

The following analysis supports the following two issues empirically: (a) that we can still route in $O(\log N)$ and (b) that the pointers adapt to d fast. For the simulation we used a corpus of 629,228 English words [7]. Apart from words, that are used in daily life, the corpus also contains short sequences of symbols/digits like '1876-1909'. This corpus fits our needs particularly well because it contains word frequencies as well. We used the frequencies to simulate common user behavior where a few words are searched for very often whereas the rest is seldomly searched for (Zipf distribution [13]). We also simulated the system with evenly distributed query- and entry-load with almost the same results as with the Zipf distribution.

The load-balancing algorithm used in the experiments is not intended for real-world P2P-systems. We just implemented a simple algorithm to evenly distribute the entry load across the nodes. For scalable load-balancing algorithms we refer the reader to other work like e.g. [3], which can be combined with our approach.

Routing Performance. For the results presented in Tab. 1 we created rings of different sizes with all words already inserted (ie. evenly distributed entry load). The pointer placement algorithm was run for approx. $2 \log N$ rounds and then

# nodes	avg. hops theory	avg. hops practice	# nodes	avg. hops theory	avg. hops practice
2	0.50	0.50	512	4.50	4.50
4	1.00	1.00	1,024	5.00	*4.99*
8	1.50	1.50	2,048	5.50	5.50
16	2.00	2.00	4,096	6.00	6.00
32	2.50	2.50	8,192	6.50	6.50
64	3.00	3.00	16,384	7.00	7.00
128	3.50	3.50	32,768	7.50	7.50
256	4.00	4.00	65,536	8.00	8.00

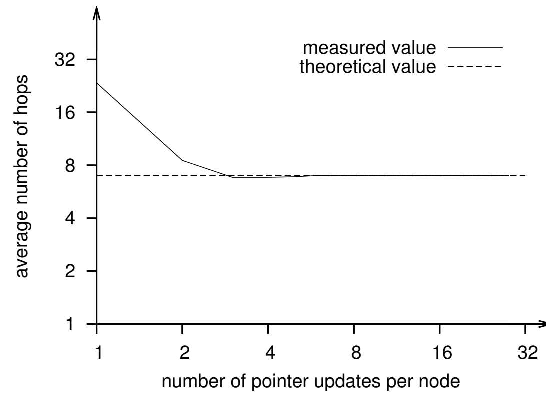
Table 1: Empirical analysis of the routing performance of Chord$^{\#}$: Just one entry differs by only 0.5%

20,000 queries were executed. The word frequencies were used to search for common words more often than for other words—the query load was Zipf distributed. The average routing performance for the queries is shown in Tab. 1. The measured performance matches the theoretical results almost perfectly.

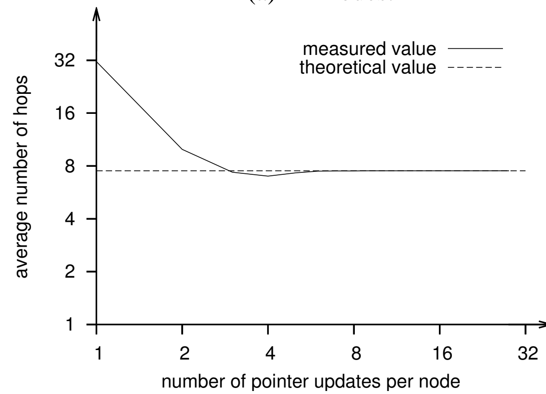
This experiment confirms our analytical results: it is indeed possible to route with $O(\log N)$ in absence of hash functions.

Adaptation Speed. In Fig. 5 the adaptation speed is depicted. The x-axis shows the number of pointer updates per node whereas the y-axis shows the average routing performance for a set of 20,000 queries. For this test we also created load-balanced rings with all words inserted and then we flushed the routing tables. For each round we ran the pointer placement algorithm once on each node (the node order was randomized) and afterwards the 20,000 queries were issued. After $\log N$ rounds the expected performance was reached.

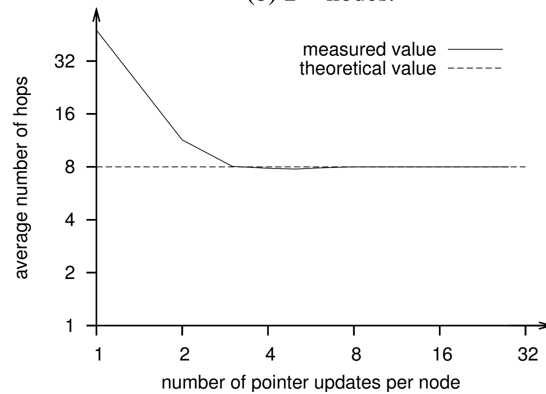
The routing performance should have converged after $\log N$ updates because with each update one entry in the routing table gets the correct value. After the first update only the neighbor is exactly known, all other entries are either empty or contain random values. For each following run at least one further entry gets the correct value because of the recursive definition of the pointer placement algorithm. So for a static system $\log N$ updates are needed, for dynamic systems more runs may be necessary, but the routing performance converges to $\log N$ after few updates.



(a) 2^{14} nodes.



(b) 2^{15} nodes.



(c) 2^{16} nodes.

Figure 3: Convergence of the routing performance for 16k, 32k and 64k nodes

0	$n_i + 1$
1	$n_i + 2$
2	$n_i + 4$
3	$n_i + 8$
4	$n_i + 16$
$\vdots$	$\vdots$
l	$n_i + 2^{l-1}$

a) Routing table for Chord

0	$n_i + 1$	$n_i + 2$	$n_i + 3$
1	$n_i + 4$	$n_i + 8$	$n_i + 12$
2	$n_i + 16$	$n_i + 32$	$n_i + 48$
3	$n_i + 64$	$n_i + 128$	$n_i + 192$
4	$n_i + 256$	$n_i + 512$	$n_i + 768$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
l	$n_i + 4^{l-1}$	$n_i + 2 \cdot 4^{l-1}$	$n_i + 3 \cdot 4^{l-1}$

b) Routing table for DKS with $k = 4$

Figure 4: Routing tables in DKS with l entries for $k \in \{2, 4\}$

6 Chord$^{\#}$ Extensions

All extensions of Chord can be analogously applied to Chord$^{\#}$; we show this using DKS [1] as an example. In comparison to Chord DKS provides improvements of the maintenance overhead and of the routing performance. Arbitrary keys can be found in $O(\log_k N)$ instead of $O(\log_2 N)$ where k can be chosen arbitrarily at startup time. Chord$^{\#}$ is orthogonal to the DKS extensions therefore we can combine them and get the optimal of both approaches.

Fig. 4 shows routing tables generated by Chord$^{\#}$ equivalent to the DKS ones for $k \in \{2, 4\}$ ($n_i + x$ denotes the xth neighbor of the i-th node), where DKS for $k = 2$ is equivalent to Chord. The recursive definition of pointer placement algorithm for $k > 2$ is ambiguous. We can (a) use different numbers of hops which may improve the convergence (e.g. $n_i + 64 + 64 = n_i + 32 + 32 + 32 + 32$) and (b) split the covered area in different ways ($n_i + 32 + 16 = n_i + 16 + 32$). Nonetheless it is possible to implement $O(\log_k N)$-routing in Chord$^{\#}$.

7 Related Work

After Chord [12] and CAN [9] have been published in the year 2001 several other systems with similar capabilities came up. Many recent publications focus on improvements to these existing systems. Range queries belong to a group of problems for which no satisfactory solutions are known yet [4, 8].

Perhaps the most interesting approach is *Mercury* [3]. Similar to our work it does not use a hash function and therefore has to cope with an uneven node distribution. In Mercury, the individual nodes use random walk sampling to determine the density function d which causes a lot of traffic for maintaining the

network. Chord$^{\#}$, in contrast, does not explicitly compute the density function and therefore has less CPU-time and communication overhead. Multi-attribute range queries, which we did not discuss in this paper, can be implemented analogously to Mercury in our system.

8 Conclusion

DHTs are state-of-the-art in P2P lookup, because they uniformly distribute the key space over all nodes. However, they do not allow to balance the query load and they do not support range queries. We presented an enhancement to the Chord algorithm that performs a data lookup without using DHTs. The resulting scheme, named Chord$^{\#}$, provides a richer functionality while maintaining the same complexity as Chord. In contrast to Chord, Chord$^{\#}$ is able to adapt to imbalances in the query load and it supports range queries.

These new features can be used to improve lookup services in distributed systems like name, directory, and discovery services. Moreover, they provide the means for handling more complex queries, like relational or attributed queries. In attributed databases it is not feasible to build indexes. Rather, we store each object descriptor in k nodes, one for each of its k keys. To answer the query "give me all PDF files written by Donald Knuth before 1995", for example, the query optimizer (ref. Fig. 1) first asks the node that is responsible for 'author=Knuth', intersects the results with the range query '< 1995' and finally with all resulting documents of 'type = PDF'. Such queries require to pass a second parameter `lookup(key,query())` to the backend. For query optimization purposes, it would be helpful to maintain the cardinality of the data objects in the metadata.

Acknowledgements

This work was funded by the Zuse Institute Berlin as part of a long-term research project [10] on the design of an attribute-based distributed data management system.

References

[1] 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. In the 3rd International workshop on Global and P2P Computing

on Large Scale Distributed Systems, (CCGRID 2003) (Tokyo, Japan), May 2003.

[2] H. Balakrishnan, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Lookup up data in P2P systems. *Communications of the ACM*, 46(2), February 2003.

[3] A. Bharambe, M. Agrawal and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *SIGCOMM 2004*, August 2004.

[4] N. Daswani, H. Garcia-Molina and B. Yang. Open Problems in Data-sharing Peer-to-peer Systems. In *ICDT 2003*, January 2003.

[5] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In *Proceedings of IPTPS02*, March 2002.

[6] 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. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654 – 663. ACM Press, May 1997.

[7] G. Leech, P. Rayson and A. Wilson. Word Frequencies in Written and Spoken English: based on the British National Corpus. Longman, London, 2001.

[8] S. Ratnasamy, S. Shenker, and I. Stoica. Routing algorithms for DHTs: Some open questions. 2002.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. ACM SIGCOMM 01, San Diego, CA, USA, 2001.

[10] A. Reinefeld, F. Schintke and T. Schütt. Scalable and Self-Optimizing Data Grids. Chapter 2 (pp. 30 - 60) in: Yuen Chung Kwong (ed.), Annual Review of Scalable Computing, vol. 6, June 2004.

[11] M. Ripeanu, I. Foster and A. Iamnitchi. Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design. In *IEEE Internet Computing Journal*, 2002.

[12] I. Stoica, R. Morris, M.F. Kaashoek D. Karger, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet application. In *Proceedings of ACM SIGCOMM*, August 2001.

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