

Structured Overlay without Consistent Hashing: Empirical Results

Thorsten Schütt, Florian Schintke, Alexander Reinefeld

Zuse Institute Berlin (ZIB)

E-mail: {schuett,schintke,reinefeld}@zib.de

Abstract

Consistent hashing is at the core of many P2P protocols. It evenly distributes the keys over the nodes, thereby enabling logarithmic routing effort ‘with high probability’. However, consistent hashing incurs unnecessary overhead as shown in this paper.

By removing consistent hashing from Chord, we derived a protocol that has the same favorable logarithmic routing performance but needs less network hops for updating its routing table. Additionally, our Chord[#] protocol supports range queries, which are not possible with Chord. Our empirical results indicate that Chord[#] outperforms Chord even under high churn, that is, when nodes frequently join and leave the system.

1. Introduction

Many lookup protocols in Peer-to-Peer (P2P) networks use consistent hashing [10] for assigning keys to nodes. Consistent hashing distributes the keys equally among all nodes, which allows to provide a lookup performance of $O(\log N)$ in networks of N nodes ‘with high probability’. Unfortunately, consistent hashing is not order-preserving: it randomly distributes lexicographically adjacent keys over all nodes. Hence, queries with partial keywords, wildcards or ranges cannot be handled by lookup protocols based on consistent hashing.

Taking Chord as a starting point, we devised an algorithm that makes the hashing superfluous, but has the same runtime overhead and is superior in a number of other aspects. Our algorithm, named Chord[#],

- needs $O(1)$ hops for updating an entry in the routing table instead of $O(\log N)$ as in Chord,
- has a proven logarithmic lookup performance rather than $O(\log N)$ just ‘with high probability’,
- supports more complex queries,
- does the routing in the node space rather than in the key space.

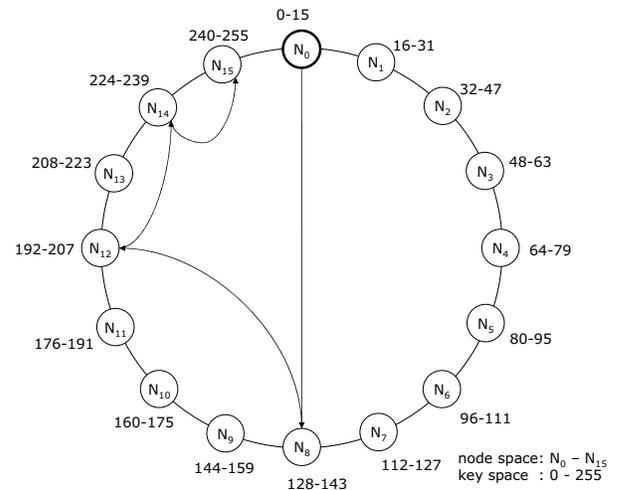


Figure 1. Chord routing hops from N_0 to N_{15}

2. Chord and Chord[#]

Fig. 1 illustrates a Chord ring with nodes N_0, \dots, N_{15} , each of them responsible for a subset of keys $0, \dots, 2^8 - 1$. A finger table in each node holds the addresses of the peers halfway, quarter-way, 1/8-way, 1/16-way, \dots , around the ring. When a node (e.g. N_0) receives a query, it forwards it to the node in its finger table with the highest identifier not exceeding $\text{hash}(\text{key})$. This halves the distance to the target in each step, resulting in $O(\log N)$ hops in networks with N nodes, because the DHT ensures a uniform distribution of the keys and nodes with high probability [17]. This allows Chord to compute the finger placement in the key space rather than the node space.

When substituting the hash function by a key-order preserving function, the keys are no longer uniformly distributed over the node space but they follow some unknown density function. To obtain the same logarithmic routing effort as in Chord, the fingers must be placed in such a way that they cross an exponentially increasing amount of nodes in the ring. The following recursive finger placement algorithm allows this, where the infix operator $x \cdot y$ means to

retrieve y from the routing table of a node x :

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

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 rather than in the key space allows us to remove the hash function and to arrange the keys in lexicographical order so that no node is overloaded. The following Figure illustrates how Chord[#] updates $finger_i$ in the finger table of node n :

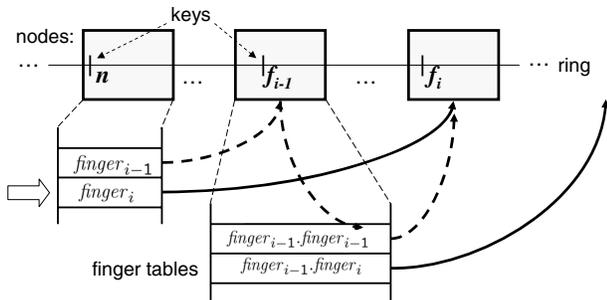


Figure 2. Finger update in Chord[#]

This finger placement has two advantages over Chord's algorithm: First, it works with any kind of key as long as a total ordering over the keys exists, and second, updating of fingers is cheaper than in Chord, because it needs just one hop instead of a full search. Chord[#] adjusts local finger table entries by utilizing the better informed remote finger table entries.

3. Proving the Logarithmic Routing Performance

Before we are going to 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 with¹

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

Hence, $O(\log N)$ hops are needed for updating a single entry in the routing table. The above equation can be rewritten

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

¹Let \oplus be the addition modulo 2^m .

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 [10]): 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).

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 :

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

3.1. Finger Placement in Chord

Both, Chord and Chord[#] use exponentially spaced 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 (see Sec. 4.1).

Chord places the i -th finger on the node that is responsible for f_i :

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

Using our integral approach from Lemma 1 and the density function $d(x)$ we develop an equivalent finger placement algorithm as follows. First, we look at the longest finger f_m . It points to $n + 2^{m-1}$ if the key space is of size 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 in clockwise direction 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 estimated by:

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

Other fingers to the $\frac{N}{4}$ -th, \dots , $\frac{N}{2^i}$ -th node are calculated accordingly and we can now formulate the following finger placement algorithm of Chord:

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

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

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

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

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

$$f_i = n \oplus 2^{i-1} \quad \blacksquare$$

We thereby derived two alternative methods to calculate the fingers in Chord. The equivalence of these two algorithms will be used in the following Section to prove the correctness of Chord[#]'s algorithm.

3.2. Finger Placement in Chord[#]

Theorem 3 (Chord[#] Finger Placement): *The following finger placement algorithm computes exponentially spaced fingers and therefore allows routing in $O(\log N)$.*

$$finger_i = \begin{cases} successor & : i = 0 \\ finger_{i-1} \cdot finger_{i-1} & : i \neq 0 \end{cases}$$

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

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

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

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

$$\int_X^{f_i} d(x) dx = \frac{2^{i-2}}{2^m} N \quad (3)$$

In Eq. 2 and Eq. 3, the only unknown is X . Comparing Eq. 2 to Theorem 2, we see that X is f_{i-1} . To calculate $finger_i$ we go to the node addressed by $finger_{i-1}$ in our finger table (Eq. 2), 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. 3. Eq. 1 is equivalent to

$$finger_i = finger_{i-1} \cdot finger_{i-1}$$

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 finger placement algorithm in Theorem 3. \blacksquare

Table 1. Parameters used in the experiments forming a 6-tuple.

Parameter	Description	Value
1. Base	Branching factor of finger table entries	2, 8, 16, 32
2. Successors	Number of direct successors stored in the successor list	4, 8, 16, 32
3. Succ. Stabilization Interval	Time spent between two successor list updates	30, 60, 90s
4. Finger Update Interval	Time spent between two finger table updates	30, 60, 300, 600, 900, 1200s
5. Latency Optimizer	Proximity routing methods for latency optimization	0, 1, 2
6. Piggybacking	piggy back routing information on queries	true, false

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[#].

4. Empirical Evaluation

In order to compare the performance of Chord[#] with that of Chord, we implemented a discrete event simulator that simulates dynamic P2P systems under churn. Previous performance studies focused mostly on static properties like the average number of routing hops or the size of the routing table. Li *et al.* [12, 13] went one step further and compared the bandwidth and latency of P2P protocols under churn, which gives a more realistic picture on the practical usefulness.

Following their approach as closely as possible, we run about 2000 experiments with various parameter sets. Both algorithms, Chord and Chord[#], were simulated with a ring of 1024 nodes. The latencies between the nodes are given by the King [8] dataset² which is based on real data observed in the Internet.

To simulate churn, each node joins and leaves once per hour. Each node issues a lookup for a random key every ten minutes. All intervals are distributed exponentially. Messages have a length of 20 bytes plus 4 bytes for each node address contained in the message. Each experiment runs for six hours of simulated time.

The parameter combinations listed in Tab. 1 gave a total of 1728 experiments for Chord[#]. We ran less Chord experiments because it does not support e.g. piggybacking. As a performance measure we used the average (resp. median) latency per key lookup and as a cost measure the bandwidth consumed by each node (bytes per node per second).

²We observed an inconsistency in the data given in Li *et al.* [12, 13], 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.

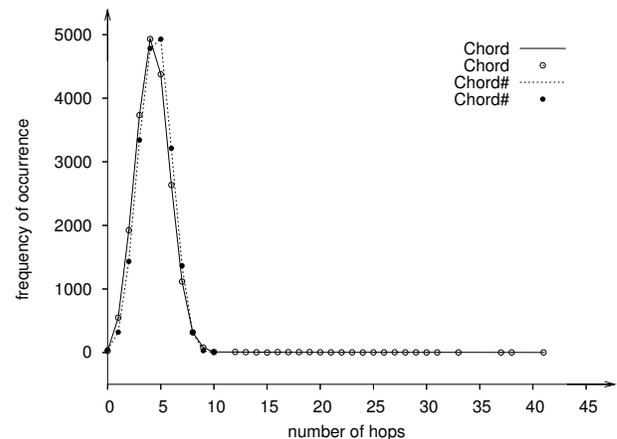


Figure 3. Hop frequency for Chord and Chord[#]. Note that Chord[#] needs at most 10 hops compared to 42 hops needed by Chord. Parameters: (2, 4, 30, 60, 0, false)

4.1. Median versus Average

In their first paper [12], Li *et al.* plotted the *average* of their results and in the second one [13], they took the *median*. Since it was not clear to us why they switched from average to median, we checked both cases and found that they differ substantially. Averaging over all lookups in an experiment gave larger latencies, because there are a few instances with a considerably higher latency than the large majority (ref. Fig. 3). The few extreme points – Chord needs at maximum 42 compared to 10 hops by Chord[#] – result in a higher *average* value. The median, in contrast, has a smaller latency because the extreme points do not change the result. In our opinion, the average should be taken for comparing P2P algorithms, because it represents *all* cases and not only the one in the middle. Nevertheless, to allow comparison with the results of Li *et al.*, we also plotted the median values in Fig. 4 as discussed in the following Section.

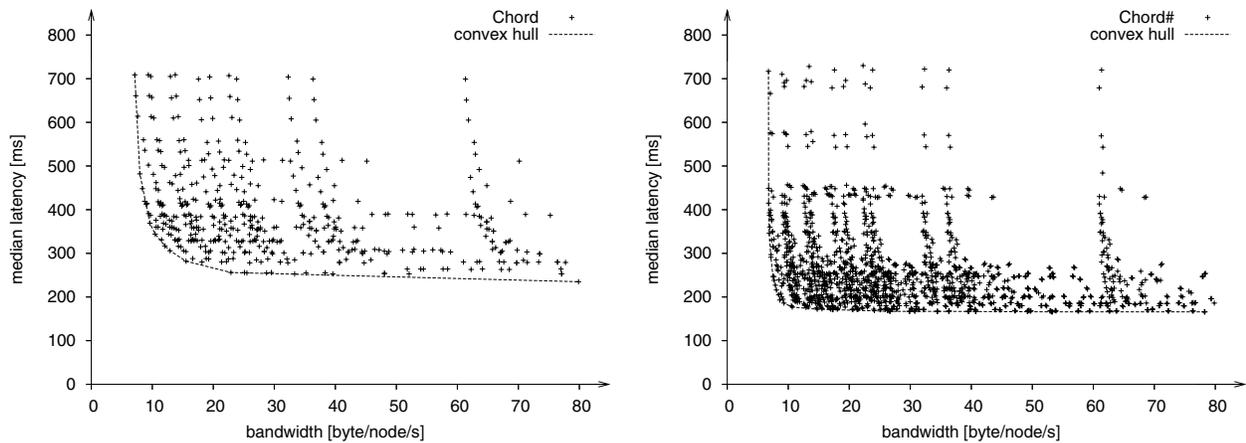


Figure 4. Chord and Chord# under churn (median latency). Each '+' represents one out of the 480 resp. 1728 experiments. The convex hull (bottom line) illustrates the best parameter combinations.

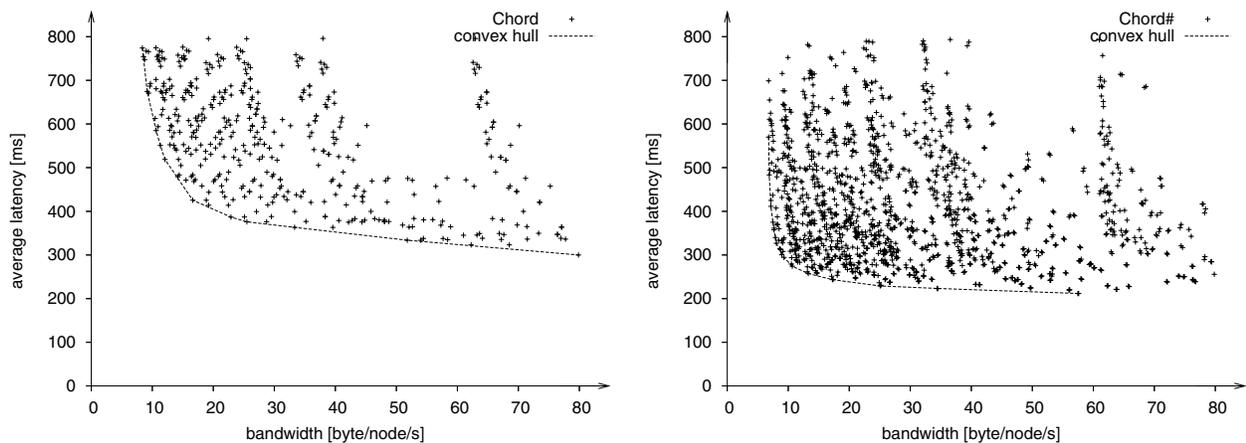


Figure 5. Chord and Chord# under churn (average latency). Each '+' represents one of the parameter sets. The convex hull (bottom line) illustrates the best parameter combinations.

4.2. Management Overhead

Certain management tasks must be done at regular time instances to keep the system operational. We distinguish the following activities:

Successor stabilization. Each node has a list of successors which are periodically checked whether they can still be reached. If a node has crashed, the successor list is updated accordingly.

Finger updates. To keep the routing table up to date, the fingers are periodically validated and updated.

Join, leave, and fail. Nodes may join or leave the system, that is, they register or unregister themselves to the

system. They may also fail, e.g. leave without unregistering.

Key search. The load caused by a key search is – compared to the other tasks – the only load triggered by the actual use of the system. All other tasks in this list are needed for maintaining the system.

4.3. Comparing Chord with Chord#

Fig. 4 shows for different parameter combinations of Chord and Chord# the *median* latencies versus bandwidth, while Fig. 5 gives the same data for the *average* latencies. Each '+' represents one parameter combination. The interesting combinations are those on the convex hull: They represent the favorable combinations with a low latency at

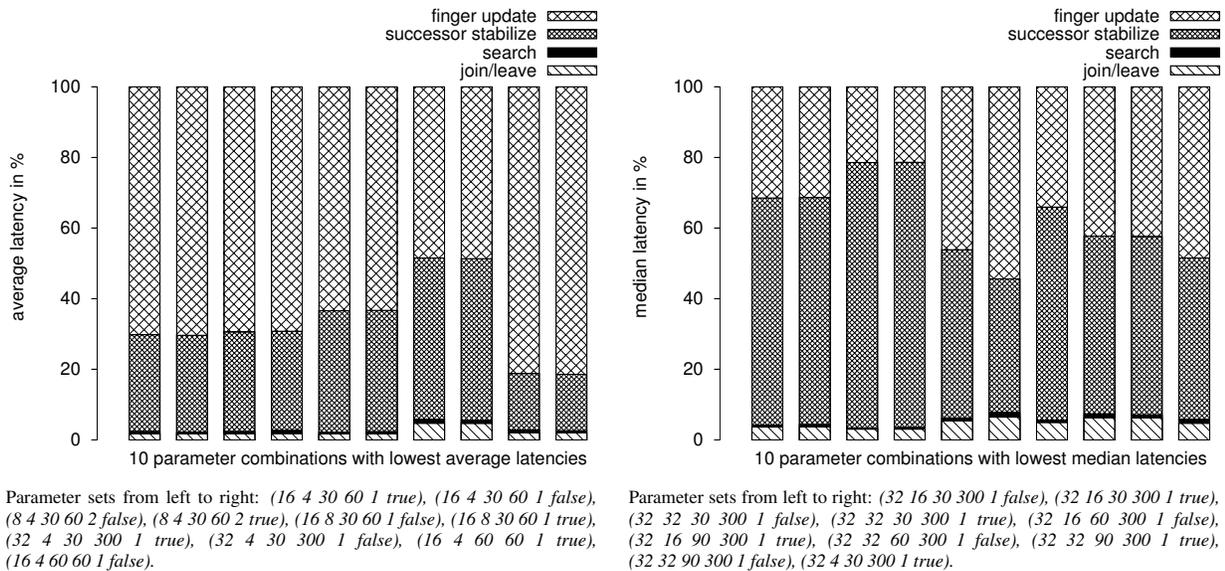


Figure 6. Chord# configurations with the lowest average (left) and lowest median latency (right).

low maintenance cost. All other data points are inferior in terms of performance and can be ignored. Nonetheless, we plotted them as well, because the configurations on the convex hull are more fragile, meaning that there is a risk for the ring to break under extremely high churn.

Comparing the convex hulls in either of Fig. 4 or 5, it is obvious that Chord# is superior to Chord in terms of latency. Most interesting is the lower left hand corner of the plots which contains parameter combinations with a low bandwidth and low latency.

As expected, the advantage of Chord# is more pronounced when comparing the median latencies rather than the average. In the latter case, latencies of all nodes are taken into account, not just the middle one. The results in Fig. 5 clearly demonstrate that Chord# requires less bandwidth and lower latencies than Chord.

4.3.1. Analyzing Parameters with Lowest Latency.

For Chord# we further analyzed the ten combinations with the lowest latency (Fig. 6) and split their bandwidth usage into four categories: finger updates, successor stabilization, search, and join and leave as described in Section 4.

Most of the bandwidth is used by the finger updates, as illustrated in the left part of the figure which shows the ten combinations with the lowest *average* latencies. This is a remarkable result, considering that Chord# needs only $O(1)$ hops per finger update as compared to $\log(N)$ hops in Chord. Even with this little update effort, the finger updates dominate the overall bandwidth of Chord#!

The right part of the figure shows the same data, but with the ten combinations having the lowest *median* latency results. Here more effort is spent for stabilizing the succes-

sors and less for updating the fingers³.

4.3.2. Proximity Routing. The standard finger placement algorithm in Chord and Chord# determines just one specific target node for each finger in the finger table. By checking the nodes nearby the goal and selecting the one with the lowest network latency, the average access latency to the target can be reduced. Common candidates for this so-called *proximity routing* are nodes from the successor list. In our experiments we found that proximity routing reduces the average latency by about 200 ms.

4.3.3. Finger Update Interval. Fig. 7 compares the performance of Chord and Chord# with the twenty best parameter sets. Each line connects the two results for one parameter set. For the majority of the cases Chord# performs better in terms of latency and bandwidth. The points in the upper left corner have long successor stabilizations and long finger update intervals. Here Chord#'s latency degrades because routing entries are forwarded between nodes and therefore stale entries need some time to propagate which causes more routing failures. The bandwidth usage is similar because it is no longer dominated by the finger updates.

5. Related Work

Since Chord [17] and CAN [15] were introduced in 2001, several P2P protocols with similar capabilities came

³For the lowest latency combinations the bandwidth tends to be higher than the mean, whereas for the low bandwidth combinations the latency tended to be higher.

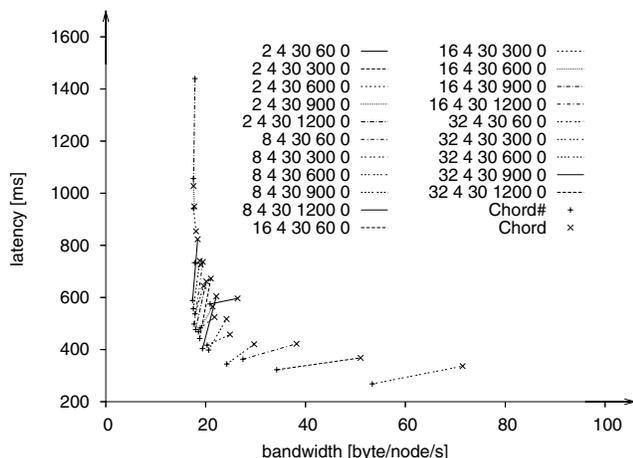


Figure 7. Comparison between Chord and Chord# for same parameter sets.

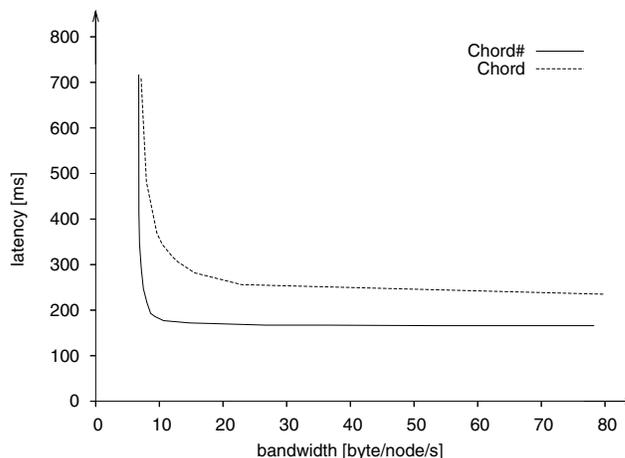


Figure 8. Chord and Chord# under Churn

up. Many publications focus on their improvement. Range queries belong to a group of challenges for which, to our knowledge, no satisfactory solutions are known yet [5, 14].

SkipNet and Skip Graph [9, 3] both support range queries, but as in Chord performance guarantees can only be given with high probability.

Mercury [4] is an attempt that tries to support range queries. Similar to Chord# it does not use consistent hashing and therefore has to deal with load imbalance. Mercury determines the density function with random walk sampling which generates much more communication traffic for maintaining the finger table. Chord#, in contrast, never computes the density function and therefore incurs less overhead. Multi-attribute range queries, which were also addressed by Mercury, can be introduced analogously to Chord#.

Other approaches [2, 16] use space-filling curves to map multi-dimensional keys to the nodes and to allow range queries. Space-filling curves are locality preserving, but they incur more maintenance overhead and larger routing costs.

6. Summary

We simulated Chord and Chord# under churn. The results confirm that Chord# outperforms Chord. Comparing the best parameter combinations of both protocols (represented by the convex hull in Fig. 8) shows that the queries in Chord# have a lower latency and incur less bandwidth for maintaining the system.

In practice, one would choose one of the ten best configurations listed in Fig. 6. In each of these cases, the finger

update dominates the overall node bandwidth. This is even more remarkable when considering that Chord#'s finger update needs only one hop versus $\log(N)$ hops in Chord – still the finger update dominates Chord#'s overall cost.

Our experimental setup, which follows that of Li *et al.* [12, 13] as closely as possible, simulates a dynamic system with nodes joining, leaving and crashing. However, it does not take into account new key insertions, multiple copies and traffic caused by load balancing [11, 6]. Especially the latter should be considered to obtain a more realistic picture on the practical usefulness, because Chord# must re-balance the keys from time to time to ensure an equal load over all nodes when non-randomly distributed keys are inserted. This is the price to be paid for supporting range queries. In the future we plan to extend our empirical analysis by examining more realistic usage patterns with key insertions and key duplications.

Many publications in the recent past focused on optimizing the bandwidth usage of P2P protocols. They mainly concentrated on the successor stabilization process [1, 7]. Our results support the importance of this research. But Fig. 6 shows that for low latency scenarios with high churn rates the finger stabilization dominates the bandwidth usage. Chord# reduces the finger update traffic significantly as shown in Fig. 6 and 7.

Acknowledgements

Thanks to Seif Haridi (SICS) for visiting us at the right time. Part of this work was financially supported by the EU project CoreGrid.

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. *Workshop on Global and P2P Computing, CCGRID 2003*, May 2003.
- [2] A. Andrzejak and Z. Xu. Scalable, efficient range queries for Grid information services. *P2P2002*, Sep. 2002.
- [3] J. Aspnes and G. Shah. Skip graphs. Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Jan. 2003.
- [4] A. Bhambe, M. Agrawal and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. *ACM SIGCOMM 2004*, Aug. 2004.
- [5] N. Daswani, H. Garcia-Molina and B. Yang. Open Problems in Data-sharing Peer-to-peer Systems. *ICDT 2003*, Jan. 2003.
- [6] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems. *VLDB 2004*.
- [7] A. Ghodsi, L. O. Alima and S. Haridi. Low-Bandwidth Topology Maintenance for Robustness in Structured Overlay Networks. In *38th International HICSS Conference*.
- [8] 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, Nov. 2002.
- [9] N. Harvey, M. Jones, M. Theimer and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS), Mar. 2003.
- [10] D. Karger, E. Lehman, T. Leighton, R. Panigraha, M. Levine and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *29th Annual ACM Sympos. Theory of Comp.*, 654-663, May 1997.
- [11] D. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. *IPTPS 2004*, Feb. 2004.
- [12] 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.
- [13] 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.
- [14] S. Ratnasamy, S. Shenker, and I. Stoica. Routing algorithms for DHTs: Some open questions. *IPTPS 2002*, Mar. 2002.
- [15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. *ACM SIGCOMM 2001*, Aug. 2001.
- [16] C. Schmidt and M. Parashar. Enabling flexible queries with guarantees in P2P systems. *IEEE Internet Computing*, 19-26, May/June 2004.
- [17] 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.