Koorde: A simple degree-optimal distributed hash table

M. Frans Kaashoek and David R. Karger
MIT Laboratory for Computer Science
{kaashoek,karger}@lcs.mit.edu

Abstract

Koorde[1] is a new distributed hash table (DHT) based on Chord [15] and the de Bruijn graphs [2]. While inheriting the simplicity of Chord, Koorde meets various lower bounds, such as $O(\log n)$ hops per lookup request with only 2 neighbors per node (where n is the number of nodes in the DHT), and $O(\log n/\log\log n)$ hops per lookup request with $O(\log n)$ neighbors per node.

1 Introduction

A number of different performance measures exist for DHTs; optimizing one tends to put pressure on the others. These measures include:

1. **degree**: the number of *neighbors* with which a node must maintain continuous contact;
2. **hop count**: the number of hops needed to get a message from any source to any destination;
3. The degree of **fault tolerance**: what fraction of the nodes can fail without eliminating data or preventing successful routing;
4. The **maintenance overhead**: how often messages are passed between nodes and neighbors to maintain coherence as nodes join and depart;
5. The degree of **load balance**: how evenly keys are distributed among the nodes, and how much load each node experiences as an intermediate node for other routes.

There are other measures for DHTs, such as delay (i.e., proximity routing) and resilience against malicious nodes, but because of the page limit we mostly ignore them in this paper.

A quick survey of existing systems shows some common trends. Degree tends to be logarithmic, or at worst polylogarithmic. Hop count is generally logarithmic as well. These bounds turn out to be close to optimal, but not optimal.

We point out that for any constant degree k, $\Theta(\log n)$ hops is optimal. We also show that to provide a high degree of fault tolerance, a node must maintain $O(\log n)$ neighbors; in that case, however, an $O(\log n/\log\log n)$ hop count may be achieved.

Koorde is a simple DHT that exploits de Bruijn graphs to achieve these lower bounds. Koorde may be important in practice, because it has low maintenance overhead.

2 Bounds and tradeoffs

In this section, we discuss lower-bounds and tradeoffs between some of the DHT measures.

2.1 Degree and hop count

Our first observation relates the degree and routing hops in any system:

Lemma 2.1. *An n-node system with maximum degree d requires at least $\log_d n - 1$ routing hops in the worst case and $\log_d n - O(1)$ on average.*

Proof. Since the maximum degree is d, the number of nodes within distance h is, by induction, at most $d^{h+1}/(d-1)$. Since $d^{\log_d n} = n$, it follows that some node is at distance at least $\log_d n - 1$. The average-case claim follows from the corollary that in fact almost all nodes are at distance $\log_d n - O(1)$. □

Many protocols (Chord, Kademlia [9], Pastry [11], Tapestry [5]) offer $O(\log n)$ degree and hop count. CAN [10] uses degree d to achieve $O(dn^{1/d})$ hops. These are near-optimal bounds but the lower bound

This research was conducted as part of the IRIS project (http://project-iris.net/), supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660.

[1] A mathematical chord is a koorde in the Dutch language.

allows for (i) $O(\log n)$ hops using only constant degree and (ii) a degree of $O(\log n)$ achieving $O((\log n)/\log\log n)$ hops.

The Viceroy DHT [8] provides constant *expected* degree. However, its *high probability* bound is $O(\log n)$—in fact, it is likely that a few unlucky nodes will have $\Omega(\log n)$ degree. Viceroy is also relatively complex. For example, it involves estimating the size of the network to select various "levels" for nodes in the system. Furthermore, fault tolerance is not discussed in the Viceroy paper.

2.2 Fault tolerance and maintenance

A strong notion of fault tolerance is one that requires all live nodes to remain connected in the presence of node failures. Connectivity is a necessary (but not sufficient) condition for efficient routing.

Lemma 2.2. *In order for a network to stay connected with constant probability when all nodes fail with probability* $1/2$*, some nodes must have degree* $\Omega(\log n)$.

Proof. Suppose that the maximum degree $d < \log n - 2\log\log n - 1$. Then the probability that a particular node is *isolated*, staying up but losing all of its neighbors, is at least $(1/2)^{d+1} \geq 2^{2\log\log n}/2^{\log n} = (\log^2 n)/n$. Since there are n nodes, we expect at least $\log^2 n$ nodes to become isolated. This almost gives what we want, but we must deal technically with the fact that node isolations are not independent: if one node is not isolated then it has a living neighbor, which decreases the odds of other nodes being isolated.

However, since the maximum degree is d, each node has at most d^2 neighbors at distance 2. It follows that there is a set S of n/d^2 nodes such that no two share any neighbors (the set can be found by a greedy algorithm: take a node, include it in S, delete its distance-2 neighbors, repeat). Since none of the nodes in S share any neighbors, their "isolation events" are independent; thus the probability that no node in S gets isolated is at most $(1 - (\log^2 n)/n)^{n/d^2} < 1/e$ (using the inequality $(1-x)^{1/x} < 1/e$ for any x). In other words, such an event happens with constant probability. □

A star graph has only one node with degree exceeding 1 and still manages to stay connected with constant probability in the above model. However. in a P2P system we want no node to be of substantially above-average degree. Under such a restriction the lemma can be strengthened: the *average* degree must be $\Omega(\log n)$. Space precludes a proof.

As a particular case, a network partition can also be thought of as a collection of failures of the nodes on the "other side" of the partition; tolerating a failure rate of 1/2 means that the larger half of the system will stay connected after the partition. This argument generalizes to failure probabilities $p \neq 1/2$; roughly speaking, d must be such that the expected number of *surviving* neighbors $pd = \Omega(\log n)$.

A similar argument can be applied to the maintenance traffic. Liben-Nowell, Balakrishnan, and Karger [7] introduce the notion of "half-life" as the time it takes for a peer-to-peer network to replace half its nodes through departures and new arrivals, and prove that every node must be notified about $\Omega(\log n)$ other nodes per half-life if the network is to remain connected.

Most DHTs support some mechanism for handling nonbyzantine failures, but few provide analytical results. The Chord DHT uses "successor lists" of $O(\log n)$ neighbors of each node and proves that with these successor lists, the network remains connected (and continues to route efficiently) with high probability even if half the nodes fail simultaneously. Building on the successor lists, Liben-Nowell, Balakrishnan and Karger show how to limit maintenance traffic to $O(\log^2 n)$ per node per half-life (compared to the lower bound of $\Omega(\log n)$.

Saia, Fiat, Gribble, Karlin, and Saroiu [12] provide a DHT with analytical results in the presence of malicious nodes. Their DHT is "adversarially fault tolerant" in that an adversary killing *any* half of the nodes (not necessarily at random) is only able to disconnect an ϵ fraction of the surviving nodes. To achieve this high level of fault tolerance, however, some performance is sacrificed. Each node maintains $O(\log^3 n)$ state. Lookups take $O(\log n)$ time but require $O(\log^3 n)$ messages. Every data item must be replicated $\log N$ times. Metadata about various items must be distributed to $O(\log^3 N)$ nodes.

2.3 Load balance

All the DHTs discussed above offer some load balance, in both the amount of data stored and the amount of routing traffic carried. In general, any one node-load is within an $O(\log n)$ factor of the average load over the system with high probability. The Chord DHT shows how, by replicating each node into $O(\log n)$ "virtual nodes," it is possible to improve the maximum-to-average load ratio to a constant (arbitrarily close to 1). A similar technique can be applied to many of the DHTs mentioned above, including Koorde. However, such replication does increase the state needed at a node, and the maintenance overhead, by a logarithmic factor. Thus, the schemes with optimal degree and hop count (Koorde and Viceroy) must give up their optimality if constant-factor load balance is to be achieved. It is an open question to find a system that is both degree optimal and load balanced.

3 Koorde: A constant-degree DHT

Koorde combines Chord with de Bruijn graphs. It looks up a key by contacting $O(\log n)$ nodes with $O(1)$ state per node.

Like Chord, Koorde uses consistent hashing [6] to map keys to nodes. A node and a key have identifiers that are uniformly distributed in a 2^b identifier space. A key k is stored at its *successor*, the first node n that follows k on the identifier circle, where node $2^b - 1$ is followed by node 0. The successor of key k is identified as *successor*(k).

3.1 De Bruijn graphs and routing

Koorde embeds a de Bruijn graph on the identifier circle for forwarding lookup requests. A de Bruijn graph has a node for each binary number of b bits. A node has two outgoing edges: node m has an edge to node $2m \bmod 2^b$ and an edge to node $2m + 1 \bmod 2^b$ (see Figure 1). In other words, a node m points at the nodes identified by shifting a new low order bit into m and dropping the high order bit. We represent these nodes using *concatenation* $\bmod 2^b$, writing $m \circ 0 = 2m \bmod 2^b$ and $m \circ 1 = 2m + 1 \bmod 2^b$.

If we assume a P2P system in which every number corresponds to a node (i.e., $2^b = n$), de Bruijn routing works as follows. With 2^b nodes in the system, consistent hashing will map key k to node k, since *successor*(k) is k.

Routing a message from node m to node k is accomplished by taking the number m and shifting in the bits of k one at a time until the number has been replaced by k (see Figure 2). Each shift corresponds to a routing hop to the next intermediate address; the hop is valid because each node's neighbors are the two possible outcomes of shifting a 0 or 1 onto its own address. Because of the structure of de Bruijn graphs, when the last bit of k has been shifted, the query will be at node k. Node k responds whether key k exists.

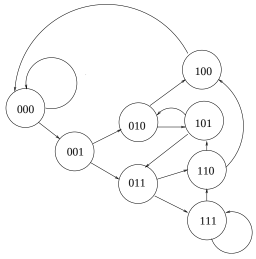

Figure 1: A de Bruijn graph for $b = 3$.

```
procedure $m$.LOOKUP($k$, kshift)
 if $k = m$  then return ($m$) /*$m$ owns $k$*/
 else {
      $t = m \circ$ topBit(kshift);
      return ($t.lookup(k,$ kshift $\langle\langle\ 1)$);
  }
```

Figure 2: Lookup of key k at node m in a de Bruijn graph. *kshift* is the key k as shifted by previous iterations. On the first call *kshift* $= k$.

This lookup algorithm will contact $b = O(\log n)$ nodes, since after b left shifts the query is at the destination node. To support the forwarding step, each node maintains information only about its two de Bruijn neighbors.

3.2 Koorde routing

Most P2P systems contain only a few of the possible 2^b nodes, because only a subset of the nodes will have joined at any given point in time and b is large for other reasons (e.g., to avoid collisions its size is determined by the output of a cryptographic hash function). Thus, some points on the identifier circle correspond to nodes that have have joined the system, while many points on the ring correspond to "imaginary" nodes.

To embed a de Bruijn graph on a sparsely populated identifier ring, each joined node m maintains knowledge about two other nodes: the address of the node that succeeds it on the ring (its successor) and the first node, d, that precedes $2m$ (m's first de Bruijn node). Since the de Bruijn nodes follow each other directly on the ring, there is no reason to keep a variable for the second de Bruijn node ($2m + 1$); it is likely that d is also the predecessor for $2m + 1$.

To look up a key k, the lookup algorithm must find *successor*(k) by walking down the de Bruijn graph. Since the de Bruijn graph is "incomplete," Koorde *simulates* the path taken through the complete de Bruijn graph, passing through the immediate real predecessor, *predecessor*(i), of each imaginary node i on the de Bruijn path.

Figure 3 shows Koorde routing as an extension of the de Bruijn routing of Figure 2. Koorde passes the current imaginary node i as an argument to the routing function. In a single routing step Koorde simulates the hop from imaginary node i to imaginary node $i \circ$ *topBit*(k), shifting in k. Koorde does so by hopping to $m.d$, which will have value near $2m$ and hopefully be equal to *predecessor*$(i \circ$ *topBit*$(k))$. If so, Koorde iterates the next routing step.

If at every hop, d is indeed the predecessor of $i \circ$ *topBit*(k), then Koorde contacts b nodes, where b is the number of bits in identifiers, because the algorithm shifts i left 1 bit at each hop.

Unfortunately, although $m.d$ is by definition the closest predecessor of $2m$, it may not be the closest predecessor of $i \circ$ *topBit*(k)—because the nodes' random distribution around the ring is not perfectly even, some other node might interpose land in between $m.d$ and $2i$. Koorde checks for this case, and corrects. When the node d receives the query, it checks whether it is indeed the predecessor of $i \circ$ *topBit*(k) by examining its own successor pointer. If it is, Koorde takes its next de Bruijn hop. If not, it forwards the query forward along the ring, following *successor* pointers, until the predecessor of $i \circ$ *topBit*(k) is encountered.

As in Section 3.1, the algorithm makes b calls to $d.lookup()$. To bound the overall work, we must bound the number of *successor* lookups.

Lemma 3.1. *In the course of doing a lookup, with high probability the number of routing hops is at most* $3b$*.*

```
procedure $m$.LOOKUP($k$, $kshift$, $i$)
  if $k \in (m, successor]$  then return ($successor$)
  else if $i \in (m, successor]$  then return (
    $d.lookup(k,$
              $kshift \langle\langle 1,$
              $i \circ topBit(kshift)))$
  else return ($successor.lookup(k, kshift, i)$)
```

Figure 3: The Koorde lookup algorithm at node m. k is the key. i is the imaginary de Bruijn node. d contains the predecessor of $2m$, and *successor* contains the successor of m.

Proof. To conserve space we analyze the expected number of hops; a high probability extension is standard. In a single step simulating the advance from imaginary node i to imaginary node $i \circ$ *topBit*(k), we first move from node $m =$ *predecessor*(i) to node $d = m.d$ and then advance from d to *predecessor*$(i \circ$ *topBit*$(k))$ using successor pointers. The nodes we traverse this way are precisely the ones located in identifier space between $2m$ and $2i + 1$. Conditioned on the values m and i, the fact that nodes are randomly distributed around the ring means that the the odds of each node landing in between these two values is $(2i - 2m + 1)/2^b$, so the expected number of nodes in this region of identifier space is $u = n(2i - 2m + 1)/2^b$. To remove the conditioning on m and i, notice that regardless of i, again because n nodes are inserted randomly, the expected value of $i - m$ (distance between i and m in identifier space) is $2^b/n$, so the expected value of u is $n(2 * 2^b/n)/2^b = 2$.

In other words, each imaginary hop involves, in expectation, following two successor pointers. Thus, in total, we expect to follow b de Bruijn pointers and $2b$ successor pointers. □

By maintaining *predecessor*$(2m)$ *and* its successor (for a total of 3 pointers per node), we reduce the expected number of successor hops per shift to 1, reducing the expected routing cost to $2b$.

3.3 Lookup in $\log n$ hops

The lookup algorithm described so far contacts $O(b)$ nodes, where b is the (large) number of bits in identifiers. However, we can reduce the number of hops to $O(\log n)$ with high probability by carefully selecting an appropriate imaginary starting node.

In Section 3.2, we started $lookup$ with the node m on which the query originated. But since m is responsible for all imaginary nodes between itself and its successor, we can choose (without cost) to simulate starting at any imaginary de Bruijn node i that is between m and its successor. If the ring contains few real nodes, then only i's top bits are significant; we can set i's bottom bits to any value we chose without leaving m's region. If we choose i's bottom bits to be the top bits of the key k, then as soon as the lookup algorithm has shifted out the top bits of i, it will have reached the node responsible for k. With high probability, the distance in identifier space from m to its successor exceeds $2^b/n^2$, which means that m's region contains imaginary nodes with all possible values of the lowest $\lg(2^b/n^2) = b - 2\lg n$ significant bits; this means we can set this many bits to equal the high order bits of k and be left to shift out only the $2\lg n$ most significant bits of the current address, which requires $O(\log n)$ hops.

3.4 Maintenance and concurrency

Just like finger pointers in Chord, Koorde's de Bruijn pointer is merely an important performance optimization; a query can always reach its destination slowly by following successors. Because of this property, Koorde can use Chord's join algorithm. Similarly, to keep the ring connected in the presence of nodes that leave, Koorde can use Chord's successor list and stabilization algorithm.

Chord has a nice "self stabilizing" property in which a ring consisting only of successor pointers can quickly construct all its fingers by pointer jumping; it is unclear whether the Koorde can similarly self-stabilize.

4 Extensions

To allow users to trade-off degree for hop count, we extend Koorde to degree-k de Bruijn graphs. When choosing $k = \log n$, Koorde can also be made fault tolerant.

4.1 Degree-k de Bruijn graphs

Koorde can be generalized to provide a simple optimal trade-off between routing table size and routing hop count. In a traditional de Bruijn graph, a node m has edges to nodes $2m$ and $2m + 1$. This graph allows us to shift in one new address bit with a single edge traversal. The same idea can be generalized to a different, non-binary base. For any k, a *base-k de Bruijn graph* connects node m to the k nodes labeled $km, km + 1, \ldots, km + (k - 1)$. The resulting graph has out degree k but, since we are shifting by a factor of k each time, has diameter $\log_k n$.

This idea can be carried over to Koorde. Instead of letting node m point at *predecessor*$(2m)$, each Koorde node points at *predecessor*(km) and the k nodes immediately following. It can be shown that under this scheme, we expect to use only a constant number of hops through real nodes to simulate a single imaginary-node hop (correcting a single base-k digit). We thus expect to complete the routing in $O(\log_k n)$ hops, matching the optimum lower bound for degree k networks.

4.2 Fault tolerance

Base Koorde has constant degree and thus, by the Lemma 2.2 earlier, cannot be fault tolerant against nodes failing with constant probability. To achieve such fault tolerance, we need to increase to minimum degree $\log n$. The approach is straightforward. To provide fault tolerance for immediate successors, we use the "successor list" maintenance protocol developed for Chord: rather than m maintaining only its immediate successor, it maintains the $O(\log n)$ nodes immediately following it. Even if nodes fail with probability 1/2, at least one of the nodes in each successor list will stay alive with high probability. In such case, routing is always possible, at worst by following live successor pointers to the correct node.

Koorde must provide a similar "backup" in case the "distant" node that m points at, *predecessor*$(2m)$, fails. If that node fails, its immediate predecessor on the ring becomes the new "correct" node for m to point at. Therefore, Koorde proactively points, not at *predecessor*$(2m)$, but at $O(\log n)$ nodes on the ring immediately preceding $2m$. One might think that the easiest way to provide such a set is to use a "predecessor list" construction similar to the successor list. However, this construction would violate several of the key invariants used to prove correctness of the Chord protocol. In particular, unlike successor pointers, predecessor pointers may point "off the ring" at nodes that have initiated the Chord join protocol but have not yet completed it; pointing at such new nodes could make lookup operations incorrect.

Fortunately, a slightly different approach does work. To set up its pointers, node m uses a lookup to find not the immediate predecessor of $2m$, but the immediate predecessor p of $2m - x$, where $x = O(\log n/n)$ is chosen so that, with high probability, $\Theta(\log n)$ nodes occupy the interval between $2m - x$ and $2m$. Node m can retrieve the successor list of p, which gives it a set of $O(\log n)$ nodes reaching to point $2m$ on the interval. These nodes provide the necessary redundancy: even if half the nodes fail, with high probability m will have a pointer to the immediate predecessor of address $2m$. This scheme requires an estimate of n, which is easy to achieve in practice by considering the distribution of a nodes' successors.

This attempt to gain fault tolerance has eliminated the constant degree attraction of Koorde. But Koorde can make good use of the extra degree: instead of working with a base-2 de Bruijn graph, Koorde can work with a base-$O(\log n)$ de Bruijn graph. With such a graph, Koorde has fault tolerance *and* the number of routing hops is $O((\log n)/\log\log n)$, which is optimal.

5 Related work

We are not the first to use de Bruijn graphs in routing [1, 3, 13, 14], and concurrent with our work others have noted their application to DHTs [4]. Compared to the related work, our primary contribution is how to simulate a lookup using a de Bruijn graph in a sparsely-populated identifier space.

Koorde's approach of using de Bruijn graphs is different than D2B's [4]. The D2B DHT attempts to organize its nodes such that the nodes form a de Bruijn graph, but cannot guarantee that the graph constructed is a de Bruijn graph. As a result, D2B can guarantee only with high probability that the out degree is $O(1)$. D2B also modifies node identifiers to create a de Bruijn graph. Koorde puts no restrictions on applications in how they choose node identifiers. Finally, Koorde inherits Chord's algorithms for handling concurrent joins; the D2B technical report doesn't discuss this topic.

6 Summary

Koorde allows its users to tune the out-degree from 2 to $O(\log n)$ to achieve hop counts ranging from $O(\log n)$ to $O(\log n/\log\log n)$. This lets users trade maintenance overhead against hop count, which may be important in practice for systems in flux. An implementation of Koorde is available as part of the Chord software distribution (`http://www.pdos.lcs.mit.edu/chord`).

References

[1] BERMOND, J.-C., AND FRAIGNIAUD, P. Broadcasting and gossiping in de Bruijn networks. *SIAM Journal on Computing 23*, 1 (1994), 212–225.

[2] DE BRUIJN, N. A combinatorial problem. In *Proc. Koninklijke Nederlandse Akademie van Wetenschappen* (1946), vol. 49, pp. 758–764.

[3] ESFAHANIAN, A., AND HAKIMI, S. Fault-tolerant routing in de bruijn communication networks. *IEEE Trans. on Computers 34*, 9 (1985), 777–788.

[4] FRAIGNIAUD, P., AND GAURON, P. The content-addressable network D2B. Tech. Rep. 1349, CNRS Universié de Paris Sud, January 2003.

[5] HILDRUM, K., KUBATOWICZ, J. D., RAO, S., AND ZHAO, B. Y. Distributed Object Location in a Dynamic Network. In *Proc. 14th ACM Symp. on Parallel Algorithms and Architectures* (Aug. 2002).

[6] KARGER, D., LEHMAN, E., LEIGHTON, F., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. 29th Annual ACM Symposium on Theory of Computing* (El Paso, TX, May 1997), pp. 654–663.

[7] LIBEN-NOWELL, D., BALAKRISHNAN, H., AND KARGER, D. R. Analysis of the evolution of peer-to-peer systems. In *Proc. PODC 2002* (Aug. 2002).

[8] MALKHI, D., NAOR, M., AND RATAJCZAK, D. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of Principles of Distributed Computing (PODC 2002)* (July 2002).

[9] MAYMOUNKOV, P., AND MAZIERES, D. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. 1st International Workshop on Peer-to-Peer Systems* (Mar. 2002).

[10] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *Proc. ACM SIGCOMM* (San Diego, CA, August 2001), pp. 161–172.

[11] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-s cale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)* (Nov. 2001).

[12] SAIA, J., FIAT, A., GRIBBLE, S., KARLIN, A., AND SAROIU, S. Dynamically fault-tolerant content addressable networks. In *Proc. 1st International Workshop on Peer-to-Peer systems* (Mar. 2002).

[13] SAMATHAM, M., AND PRADHAM, D. The de bruijn multiprocessor network: a versatile parallel processing and sorting network for VLSI. *IEEE Trans. on Computers 38*, 4 (1989), 567–581.

[14] SIVARAJAN, K., AND RAMASWAMI, R. Multihop lightwave networks based on de bruijn graphs. In *INFOCOM'92*, pp. 1001–1011.

[15] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM* (San Diego, Aug. 2001).