Diminished Chord: A Protocol for Heterogeneous Subgroup Formation in Peer-to-Peer Networks

David R. Karger[1] and Matthias Ruhl[2]

[1] MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139, USA
karger@csail.mit.edu
[2] IBM Almaden Research Center
San Jose, CA 95120, USA
ruhl@almaden.ibm.com

Abstract. In most of the P2P systems developed so far, all nodes play essentially the same role. In some applications, however, different machine capabilities or owner preferences may mean that only a subset of nodes in the system should participate in offering a particular service. Arranging for each service to be supported by a different peer to peer network is, we argue here, a wasteful solution.
Instead, we propose a version of the Chord peer-to-peer protocol that allows any subset of nodes in the network to jointly offer a service *without* forming their own Chord ring. Our variant supports the same efficient join/leave/insert/delete operations that the subgroup would get if they did form their own separate peer to peer network, but requires significantly less resources than the separate network would.
For each subgroup of k machines, our protocol uses $O(k)$ additional storage in the primal Chord ring. The insertion or deletion of a node in the subgroup and the lookup of the next node of a subgroup all require $O(\log n)$ hops.

1 Introduction

In most of the P2P systems developed, all nodes play essentially the same role in the system. In many applications, however, this might not be appropriate: nodes might be heterogeneous either in their capabilities or their approved uses. From the capability perspective, a bandwidth-intensive application might wish to limit itself to the subset of nodes with high connectivity, a storage-intensive one might focus on nodes with large disks, a compute intensive one on fast machines. Socially, in an environment where many different P2P services can be offered, some users might be comfortable running only certain "nice" services on their machines, steering clear of music-piracy services, music piracy detection services, or code-cracking endeavors by hackers or the NSA.

One way to support such endeavors would be to form a separate P2P overlay for each service, and let nodes join all the overlays in which they wish to participate. This is inefficient, however, as the same routing infrastructure will have to be replicated repeatedly—a machine may find itself pointing at the same successors and fingers in an unlimited number of distinct P2P networks. Even worse, it might need to maintain *different* successors and fingers in each of its networks.

In this paper, we give an efficient mechanism for an arbitrary set of nodes to form "subgroups" without increasing their overhead. We augment the Chord protocol to allow queries of the following form: find the first node *in subgroup* X whose address follows a given address on the Chord ring. This query can be answered in the same $O(\log n)$ time as standard Chord queries. To join a given group, a machine performs $O(\log n)$ routing hops and deposits a tiny constant amount of additional information. In other words, within the larger Chord ring, we are able to simulate a Chord ring on the nodes in group X. The maintenance traffic a node uses to maintain its groups is negligible (per group) compared to the amount of traffic the node spends maintaining the underlying Chord infrastructure.

Besides the precisely measurable improvements in efficiency, our use of subrings instead of independent rings provides a less quantifiable reduction in complexity by delegating some of the most complex parts of the ring maintenance protocol to be performed only once, by the primal ring, instead of once per subring. For example, our approach simplifies the rendezvous problem. To join a Chord ring, a node needs to identify, using some out-of-band mechanism, at least one node already in the ring. This remains true for joining the base Chord ring in our protocol. But once a node is in the base Chord ring, it can easily join any existing subgroup in the ring, without any new out-of-band communication. Thus, if the primal join problem is solved once (for example, by making all nodes in the world part of a primal Chord ring) it never needs to be addressed again. As a second example, consider the use of numerous redundant successor pointers by Chord to provide fault tolerant routing in the presence of node failures. Since the subrings live inside of a ring that provides such fault tolerance, the subrings themselves do not have to do so.

Our protocol works by embedding in the base Chord ring, for each subgroup, a directory tree that lets a node find its own successor in the subgroup. We arrange for the edges of the directory tree to be fingers of the Chord ring so that traversing the directory tree is cheap. Finding the group-X successor of a given key is accomplished by finding the primal-ring successor s of the key and then finding the group-X successor of node s (from a practical perspective, this suggests that such subring lookups will take roughly twice as long as standard ones).

1.1 Chord

Our discussions below are in the context of the Chord system [1] but our ideas seem applicable to a broader range of P2P solutions. Since we are only interested in routing among subgroups of nodes, we can ignore item assignment issues and concentrate on the routing properties of the system.

Chord defines a routing protocol in which each node maintains a set of $O(\log n)$ carefully chosen neighbors called *fingers* that it uses to route lookups in $O(\log n)$ hops. To do so, it maps the nodes onto a ring which we shall consider to be the interval $[0, 1]$ (where the numbers 0 and 1 are identified). Every node receives as address some random (fractional) number in the interval (e.g. by hashing the node's name or IP-address). A node with address a then maintains *finger pointers* to all nodes with addresses of the form $succ(a+2^{-b})$, where $b \geq 1$ is an integer. By $succ(x)$ we refer to the first node succeeding address x in the address space. It can be shown that with high probability, only $O(\log n)$ of the pointers are distinct. Thus, each node has $O(\log n)$ neighbors.

For the purposes of this work, we will ignore the insertion and deletion of nodes into the (underlying) Chord system itself.

1.2 Our Results

The formal problem considered in this work is the following. We have a subset of nodes in the network which are associated with some identifier X. We want to efficiently (in terms of storage and communication) perform the following operations:

Insert(q, X)**:** Inserts the node with address q into the subset with identifier X.

Lookup(q, X)**:** Returns the first node succeeding address q that is a member of the subset with identifier X.

Note that our protocol has to handle an arbitrary number of subgroups in the P2P system simultaneously. Using the **Lookup**-function as a primitive, each node in a subgroup can determine its successor in the subgroup. Also, as with the base Chord protocol, we assume that deletions are handled just like node failures: nodes repeatedly insert themselves to stay in the system while alive, and are eventually expunged once they depart and stop inserting themselves.

We will give algorithms for **Insert** and **Lookup** that require $O(\log n)$ hops to execute. The protocol does not have to know the size of the subgroups to operate. For each subgroup of size k, we have to store $O(k)$ additional data in the network (i.e. at nodes that are not necessarily in the subgroup themselves), but at most $O(\log n)$ data per subgroup at any single node.

Moreover, our algorithms are load-balanced in the following sense. If random nodes call **Insert** and **Lookup**, then the access load will be equally distributed among $\Omega(k)$ nodes. This precludes, for example, storing the entire group membership list at a single node, since this node would get swamped with requests. If $k = 1$ the claimed load balance does not offer much help. While in this case we are already expecting the single member of X to cope with all requests to X, our approach does have the drawback of swamping a single non-member of X with routing requests. Presumably caching techniques can be used to address this issue.

The simplest version of our protocol relies on a minor technical modification of the Chord protocol. It requires that with every finger pointer $succ(a + 2^{-b})$

we also maintain a *prefinger* pointer to the immediate predecessor $pred(a+2^{-b})$ of our finger. Here $pred(x)$ denotes the node preceding an address x. In Chord, fingers are actually found by finding prefingers and taking their immediate successors, so maintaining the additional prefinger information does not increase the processing requirements of the protocol.

For completeness we also give (more complicated) variants of our protocols that do without the prefinger pointers. These protocols require $O(\log n \log^* n)$ hops, however.

1.3 Related Work

An application where our subgroup protocol might be useful is a Usenet-caching application that runs on a P2P system [2]. The goal is to replace the current broadcast of Usenet news to all news servers with a more limited broadcast to an appropriate subset of servers from which news can be pulled by clients who want to read it. In this application, a given P2P node may wish to cache only certain newsgroups (e.g., those used by the node's owners). Our subgroup protocol can support this scheme by creating a single subgroup for each newsgroup.

Similar in spirit to our results, the OpenHash system [3] tries to separate the system layer in P2P system from the application layer. This allows several P2P applications to co-exist independently within the same P2P system.

In [4] subgroups within P2P systems are created, but the focus is on subgroups corresponding to regions of administrative control. To maintain this control, lookups are required to be resolved wholly within subgroups, not utilizing the rest of the network, as is done in our work.

2 The Subgroup Protocol

In this section, we state and analyze a Diminished Chord protocol for subgroup creation. For simplicity, we will just consider a single subgroup in the following discussion. We will refer to the nodes in the given subgroup as "green nodes." Later we will discuss the interactions between multiple groups.

2.1 A Tree-Based Solution

To provide some intuition, we outline a solution for the case where the nodes in our P2P network have somehow formed an ordered (by addresses) depth-$O(\log n)$ binary tree, such that each machine has a pointer to its parent machine in the tree. Some of the leaf nodes in the tree can become green, and we want any leaf node to be able to resolve a query of the form "what is the first green node following me in the tree?"

To support such queries, we augment the tree so that every node x stores the minimum address of a green node in the right subtree of x, if one exists. Note that this requires storing at most one value in any node. In fact, since each green node can be stored only in its ancestors, each green node will generate $O(\log n)$

storage in the tree. When a new node decides to become green, it takes $O(\log n)$ work for it to announce itself to its $O(\log n)$ ancestors.

Given such storage, we can easily answer a green-successor query. Let q be a leaf node and s the green successor of q. Consider the root-leaf paths to q and to s. These two paths diverge at some node a with the path to q going left and the path to s going right (since s is a successor of q). To be the green-successor of q, it must be that s is the first green node in the right subtree of a. It follows by the previous paragraph that a will hold s. This leads to the following algorithm for finding s in time $O(\log n)$. Walk up the path from q to the root. Each time we arrive at a node along a left-child pointer, inspect the contents of the node. This will ensure that we inspect node a and thus that we find s.

Since a green node may be stored in all of its ancestors, this scheme uses $O(\log n)$ space per green node. We can improve this bound by noticing that we only need to store s in the *highest* node in which it is stored in the scheme above. Since s is stored only at ancestors of s, any query that traverses *any* node storing s will necessarily, as it continues up to the root, traverse the highest node that stores s. This reduces the space usage to $O(1)$ per green node.

2.2 Embedding the Tree

We studied the tree because our approach using Chord is to embed just such a tree into the Chord ring. This can be explained most simply by assuming that all addresses in the Chord ring are occupied by nodes; once we have done so we will explain how to "simulate" the full-ring protocol on a ring with only a small number of nodes.

Our tree is actually built over a space of "address representations" in the Chord ring. For each subgroup, we have a *base address* a_0, which for example could be computed as a hash of the group's name X. Let $\langle a, b\rangle$ denote the address $(a_0 + b/2^a) \mod 1$ for $1 \leq a$ and $0 \leq b < 2^a$ (recall that we have defined the Chord ring to be the interval $[0, 1]$, so all addresses are fractions). Note that representations are not unique—in particular, $\langle a, b\rangle$ actually defines the same address as $\langle a + 1, 2b\rangle$—but we will treat these as two distinct nodes in the tree. The work for a particular tree node will be done by the machine at the address the node represents; one machine thus does work for at most one tree node on each level of the tree.

We make the $\langle a, b\rangle$ into a tree by letting $\langle a, b\rangle$ be the parent of the two nodes $\langle a + 1, 2b - 1\rangle$ and $\langle a + 1, 2b\rangle$ (where $\langle a, -1\rangle := \langle a, 2^a - 1\rangle$). Note that under this definition, one (the right) child of a node actually defines the same address as that node, while the left child is a (not immediate) predecessor of the node. Furthermore, given the full address space assumption, the address gap between a node and its parent is a (negative) power of two—meaning that there is a finger pointing from each node to its parent (see Figure 1).

Notice also that the tree thus defined is properly ordered with respect to the address space—that is, that the set of addresses represented in a node's subtree is a contiguous interval of the ring, and that the subtrees of the two children of a node divide the node's interval into two adjacent, equal-sized contiguous segments. It follows that our tree-based green-successor algorithm can be applied to this tree, and will return the minimal green tree-successor, which by the consistent ordering is also the minimal green address successor, of any node in the tree. The depth of the tree is equal to the number of bits in the address space, and this determines the time to query or update the data structure.

The actual tree-structure of the addresses (i.e., which finger is the parent pointer of a given address) depends on the base addresses a_0. It would be prohibitively expensive to record the tree structure; however, this is unnecessary as the correct parent pointer can easily be determined as a function of the current address and the base address.

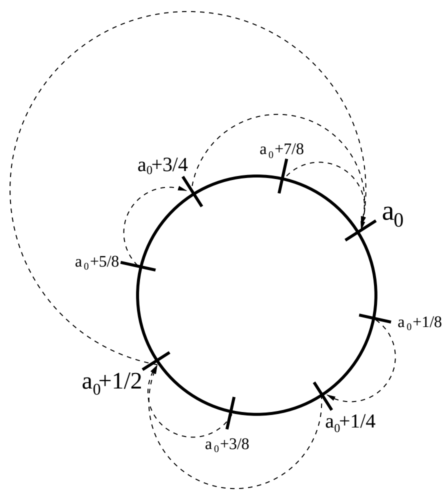

Fig. 1. The Chord address space shown as a loop. Nodes preceding the indicated addresses will be selected as nodes $p(a, b)$. The dashed lines show tree-pointers from left children to their parents.

2.3 Sparser Rings

It remains to extend this scheme to sparsely populated rings. We use the same "simulation" approach as was used to simulate a de Bruijn network over the full address space in the Koorde protocol [5]. The work a given address $\langle a, b\rangle$ needs to do is simulated by the machine immediately *preceding* that address on the Chord ring. We will call that machine $p(a, b)$.

To implement both **Insert** and **Lookup**, we have to traverse a path through the (logical) tree from a leaf address q towards the root a_0 until we find an answer. To simulate this traversal, we need to visit all the nodes immediately preceding addresses on this path. We give two different algorithms for this. The first, simpler one takes $O(\log n)$ hops, but requires that with each finger to a node $succ(a + 2^{-b})$ in Chord we also maintain a prefinger to $pred(a + 2^{-b})$, the node preceding the address $a + 2^{-b}$. As discussed above, prefingers are already found by Chord when it looks up fingers [1]. Thus, this additional information comes "for free."

For completeness, we sketch a second algorithm that computes a node-to-root path without requiring prefinger pointers. This algorithm takes $O(\log n \log^* n)$ hops for **Insert** and **Lookup**, and is considerably more complicated to implement.

With Prefingers. Given our embedding, there are two kinds of edges on a (logical) path through the tree. The edge from a right child to its parent is easy to follow in simulation since $p(a + 1, 2b) = p(a, b)$, i.e. the two tree nodes are mapped to the same physical machine. It therefore suffices to show how to get from a left child $p(a + 1, 2b - 1)$ to its parent $p(a, b)$.

So assume that we are at some machine $q_1 = p(a + 1, 2b - 1)$ responsible for address $\langle a + 1, 2b - 1\rangle$ and want to find the machine $q_2 = p(a, b)$ representing its parent $\langle a, b\rangle$. We know that q_1 precedes address $a_1 = a_0 + (2b - 1)/2^a$, while q_2 precedes address $a_2 = a_1 + 1/2^{a+1}$. First, we use the distance-$1/2^{a+1}$ prefinger from node q_1 to arrive at the node q preceding address $addr(q_1) + 1/2^{a+1}$. Then we repeatedly compute the successor of q until we pass the address a_2. This yields the last node before address a_2, i.e. node q_2.

To bound the running time, note that we perform one (prefinger) hop per move along the path. Since the tree has depth $O(\log n)$, this results in $O(\log n)$ hops. The prefinger from $p(a+1, 2b-1)$ may not point to exactly the node $p(a, b)$ that we want. For the node $p(a + 1, 2b - 1)$ is at an address slightly preceding $\langle a + 1, 2b - 1\rangle$, so its prefinger may be at an address slightly preceding $\langle a, b\rangle$ and some nodes might end up in the gap. So we may have to follow some successor pointers to reach $p(a, b)$. Nonetheless, it can be shown [1] that over the whole path the number of successor computations is only $O(\log n)$ with high probability.

As with Koorde, it would seem that our simulation must perform a number of hops equal to the number of address bits. However all but the first few hops of the simulation are actually in the purview of the same node, so take no time to simulate. The number of actual hops performed in the simulation is $O(\log n)$ with high probability.

Without Prefingers. In the previous algorithm we crucially needed the fact that we had access to prefinger pointers. Had we used fingers, the uneven distribution of nodes on the ring could have made us "overshoot" the addresses we actually needed to traverse, without any option of backtracking to them. The intuition in the following algorithm is to leave some "buffer" between the visited machines and the addresses on the path to absorb the overshoot.

In this discussion, we use the word "distance" to denote the amount of the ring's address space traversed by a finger; i.e. a finger reaching halfway around the circle is said to traverse distance 1/2.

In the previous algorithm, we simulated the traversal of a sequence of addresses on the ring by traversing the nodes immediately preceding those addresses, using prefinger pointers. The addresses we want to visit are separated by distances that are exact (negative) powers of two. Suppose that at each step, we instead traverse the finger corresponding to the desired power-of-two distance. This finger may traverse a slightly greater distance. But the random distribution of nodes on the ring means that the distance traversed by the ring is only $O(1/n)$ units greater than the intended power of two, and that over a sequence of $O(\log n)$ hops, the distance traversed is $O((\log n)/n)$ units greater than the sum of the intended powers of two (this analysis is similar to that used for Koorde [5]). In other words, even with the overshoots, we remain quite close to the intended path.

To cope with this overshoot, we arrange to begin the search at a node q' that is at distance $O((\log n)/n)$ *before* q (note that finding q' seems to require computing predecessors to move backward on the ring, which Chord does not support, but we will remove this technicality in a moment). From q' we use fingers to perform the same power-of-two hops that we would follow from q. By the previous paragraph, we will never overshoot the addresses we wanted to traverse from q. At the same time, those desired addresses will be only $O((\log n)/n)$ distance ahead of the nodes we visit; the random node distribution means that in such an interval there will be $O(\log n)$ nodes with high probability. To summarize, our finger-following path will traverse a sequence of $O(\log n)$ nodes, each only $O(\log n)$ nodes away from the address we actually want to traverse.

Chord actually proposes that each node keep pointers to its $\Theta(\log n)$ immediate successors for fault tolerance; these pointers let us reach the addresses we really want with one additional successor hop from each of the nodes we encounter on our path and thus accomplish the lookup in $O(\log n)$ time.

If we do not have the extra successor pointers, we can reach each desired address using $O(\log \log n)$ Chord routing hops from the addresses we actually traverse. By doing this separately for each address, we can find all the addresses on the leaf-to-root path of q in $O(\log n \log \log n)$ steps. This bound can be decreased to $O(\log n \log^* n)$ by computing not just one path starting $\Theta(\log n)$ nodes before q, but $\log^* n$ paths starting at distances $\log^{(k)} n$ before q. (Here $\log^{(k)}$ stands for the k-times iterated logarithm.) We omit the details in this paper, in particular since this algorithm is probably too complicated to be useful in practice.

It remains to explain how to get around the requirement of starting the search at a node q' which is $\Theta(\log n)$ nodes before q. Instead of going backward $\Theta(\log n)$ steps, we go forward $\Theta(\log n)$ steps using successor pointers. If we encounter a green node, we are done. If not, we end up at a node q'' with the same green successor as q. Since q is at distance $\Theta((\log n)/n)$ preceding q'', we can use q as the starting node to perform the green node lookup for q'', also providing the answer for q.

2.4 Load Balance

For a given subgroup, our protocol treats certain nodes (the ancestors of green nodes) as "special" nodes that carry information about the subgroup. These nodes attract query-answering work even though they are not part of the group, which may seem unfair. But much the same happens in the standard Chord protocol, where certain nodes "near" (immediately preceding) a given node become responsible for answering lookups of that node. And like the Chord protocol, our protocol exhibits a nice load balancing behavior when there are numerous subgroups. Recall that for a subgroup X, the "root" of the lookup tree for a subgroup named X is determined by a hash of the name, and is therefore effectively random. Thus, by symmetry, all addresses have the same probability of being on the lookup path for a given subgroup query. Since Chord distributes nodes almost-uniformly over the address space, we can conclude that the probability of any node being "hit" by a subgroup query is small. More precisely, since there are $O(\log n)$ steps per subgroup query, and each node is responsible for an $O(1/n)$ fraction of the address space in expectation, the probability a given node is hit by a subgroup query is $O((\log n)/n)$.

Of course, queries about the *same* subgroup tend to hit the same nodes. But suppose that many different subgroups are formed. The random (hashed) placement of query tree roots means that queries to *different* subgroups are *not* correlated to each other. This makes it very unlikely for any node to be involved in queries for many different subgroups. Space precludes fully formalizing this effect, but as one particular example, suppose that m different (possibly overlapping) subgroups are formed, and that one subgroup lookup is done for each group. Then with high probability, each node in the ring will be hit by $O(m(\log n)/n)$ subgroup queries.

3 Discussion

We stated and analyzed a protocol that allows for the creation of subgroups of nodes in the Chord P2P protocol. These subgroups are useful for efficiently carrying out computations or functions that do not require the involvement of all nodes. Our protocol utilizes the routing functionality of the existing Chord ring, so that subgroups can be implemented more efficiently than by creating a separate routing infrastructure for each subgroup.

Adding a node to the subgroup, or locating a node of the subgroup that follows a given address takes $O(\log n)$ hops. Although the algorithm is omitted for space reasons, the deletion of nodes from a subgroup can be performed in the same time bounds.

Our scheme requires only $O(k)$ storage per size-k subgroup, compared to the $O(k \log k)$ storage resulting from creating a new Chord ring for the subgroup. As opposed to the naive scheme, however, our protocol requires that information is stored at machines that are not part of the subgroup. We do not think that this is a significant problem however, as the protocol load is roughly equally distributed among at least $\Omega(k)$ machines in the network – the machines corresponding to the top k nodes in the embedded tree for a subgroup. A more complete analysis of the load distribution properties of our protocol will be in the full version of this paper.

Beyond the simple resource-usage metrics, our subring approach has an important complexity benefit over one using redundant, independent rings for each subgroup. The primal chord ring needs to handle complex correctness issues, keeping redundant successor pointers to preserve ring connectivity in the face of node failures, carefully maintaining successor pointers so as to avoid race conditions that would create artificial network partitions, and so on. Subrings can take all of this infrastructure for granted, using less robust but more efficient algorithms and relying on the primal chord ring to guarantee eventual correctness. Our approach thus parallels Chord's approach of layering efficient elements (such as proximity routing) atop a core that focuses on correctness issues (such as preserving connectivity).

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments.

References

1. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In: Proceedings ACM SIGCOMM. (2001) 149–160
2. Sit, E., Dabek, F., Robertson, J.: UsenetDHT: A Low Overhead Usenet Server. In: Proceedings IPTPS. (2004)
3. Karp, B., Ratnasamy, S., Rhea, S., Shenker, S.: Spurring Adoption of DHTs with OpenHash, a Public DHT Service. In: Proceedings IPTPS. (2004)
4. Mislove, A., Druschel, P.: Providing Administrative Control and Autonomy in Peer-to-Peer Overlays. In: Proceedings IPTPS. (2004)
5. Kaashoek, F., Karger, D.R.: Koorde: A Simple Degree-optimal Hash Table. In: Proceedings IPTPS. (2003)