2002-08-27

Scalable Peer-to-Peer Indexing with Constant State

Scalable Peer-to-Peer Indexing with Constant State

Jeffrey Considine
jconsidi@cs.bu.edu

Thomas A. Florio
tflorio@cs.bu.edu

August 27, 2002

Abstract

We present a distributed indexing scheme for peer to peer networks. Past work on distributed indexing traded off fast search times with non-constant degree topologies or network-unfriendly behavior such as flooding. In contrast, the scheme we present optimizes all three of these performance measures. That is, we provide logarithmic round searches while maintaining connections to a fixed number of peers and avoiding network flooding. In comparison to the well known scheme Chord, we provide competitive constant factors. Finally, we observe that arbitrary linear speedups are possible and discuss both a general brute force approach and specific economical optimizations.

1 Introduction

Peer-to-peer networking applications such as "file sharing communities" have seen explosive growth. Applications like Sharman Network's *Kazaa* file sharing program boast of over one hundred million downloads of their free client, with over two and a half million downloads occurring each week [2]. Recent work has noted that the amount of network traffic generated by this increasingly popular class of application now rivals that of the world wide web [9].

As the percentage of Internet users utilizing these applications and the volume of traffic they generate has exploded, the demands on such systems have intensified. The basic paradigm of these burgeoning systems can best be described by the phrase "pay to play" - users wishing to join the community and get and receive desired files must make a commitment of their own resources, often both computational resources and data. To avoid centralized points of failure or overload, it is increasingly important that work be distributed across all parties without any of them shouldering a disproportionate amount of responsibility or accountability. Simultaneously, users of these services demand the high performance of a single dedicated server serving them as quickly as possible, i.e. not limited by the requests of other users. Thus, in order to be viable, such services must be fast, reliable, network friendly and well distributed.

To these ends, much work in the realm of peer-to-peer networking has focused on the problems of topology management and indexing. The indexing problem is to map a set of keys such as file names to their locations, nodes with copies of the file. Generally, each participant in the indexing network assumes a zone of responsibility, and thus provides a mapping of keys that fall within this zone to actual locations. There is a variety of means by which zones of responsibility may be assigned. For example, until recently in the Gnutella network, hosts were responsible for keys that reside on their system. Yet other systems utilize a more centralized approach of *super nodes*: a set of system participants with sufficient resources to act as servers and batch indexing information for other clients. Kazaa and recent versions of Gnutella employ such a model.

A survey of recent work demonstrates a tradeoff between the degree of connectivity and the speed and network friendliness of indexing. Before our work, only two of these performance measures were optimized at a time at the expense of the third. For example, the Gnutella network uses a small number of connections to peers and has a bounded number of hops for searches but is forced to perform network flooding since there is no concept of a distributed indexing. More recent approaches such as Chord and CAN avoid flooding but tradeoff between scalability in terms of search speed and the number of connections to neighbors that must be maintained. It was posed as an open question whether such tradeoffs were, in fact, necessary [15]; that is, whether it is possible to provide fast, scalable searches using only connections to a constant number of peers and not use network flooding. This paper presents such a solution.

2 Related Work

Much work has been done on the topics of peering networks and caching or indexing. Indexing schemas fall into two (admittedly overly) broad categories: those that use variable sized node degrees [17] (growing with the size of the network) and those that employ flooding [11, 12, 1].

Gnutella [1] represents one of the most widely used peer-to-peer file sharing applications, and further embodies a low degree, ad-hoc network. Hosts in the network are connected to a constant number of peers, thus maintaining small state and minimizing the traffic that the host must carry. However, there is no concept of a distributed index in Gnutella so searches are based on flooding the network out to some fixed range. This makes searches both network-unfriendly and subject to horizon problems.

Consistent hashing [5] forms the basis for the indexing schemes Chord [17], CAN [14] and the scheme which we introduce. The basic idea is to randomly map both hosts and keys onto the unit circle and assign keys to the closest host on the circle. This has numerous desirable properties such as natural load balancing and minimal work as hosts arrive and depart.

Chord [17] builds on consistent hashing by organizing hosts in a ring and providing a logarithmic sized table of finger pointers allowing the ring to be quickly traversed. Chord has very fast search times (logarithmic with a small constant) and is very to robust to node failures. Because of this, we will use Chord as our standard for comparison since it is the only scheme we know of with both logarithmic search times and network friendliness.

Another scheme based on consistent is CAN, or content addressable networks [14]. Instead of using a logarithmic number of pointers, a

CAN is organized into a d-dimensional hypercube with $2d$ pointers per node. CAN search times are $O(dn^{1/d})$ so they do not scale quite as well as Chord since d is typically fixed.

Pandurangan et al. describe a protocol for building, with high probability, peer-to-peer networks of logarithmic diameter, with fixed node degree [11] [12]. Their scheme requires the use of a centralized server, and thus introduces a single point of failure. The authors do not provide an indexing schema, but such a technique is designed to give a low diameter peering network to optimize the effect of flooding.

Law and Siu describe a distributed technique of creating random regular graphs [7], and as a peering network, provide mechanisms for distributed broadcast and search [6]. Central to their technique is the use of graphs constructed of Hamiltonian cycles. Thus, repairing the graph in the case of node failures is costly - a departure or failure of a node yields a non-Hamiltonian cycle subpart. The methods of search and broadcast require flooding, and thus are not network friendly.

Finally, we must note that this work is not strictly the first to optimize all three performance measures. A similar scheme was developed independently in [8] with many similarities. This scheme also has many similarities to butterfly networks but uses a much more randomized approach and focuses on the minimal number of edges for indexing (as in our presentation, robustness is mostly orthogonal anyway). The key differences are as follows - our scheme is more deterministic, requires explicit counting information (their scheme needs only loosely approximations), naturally has fixed *incoming* degree bounds (they lose some elegance avoiding a logarithmic factor), and changes more links as nodes join and leave. With regards to this last point, their scheme only updates an expected $O(1)$ neighbors ($O(\log n)$ with high probability), In comparison, individual changes in our scheme result in many more updates but we note that only paths actually containing nodes that left are broken during this period and alternate paths usually exist.

3 Our Topology

We describe our new network topology incrementally in terms of what edges are necessary for various features. Section 3.1 beings by considering the minimal edge requirements for distributed indexing. Section 3.2 continues adding edges to allow easy maintenance of the indexing topology. Section 3.3 discusses how to build in robustness since the low number of edges present is insufficient for most guarantees. Finally, Section 3.4 discusses various optimizations to improve constant factors in searching.

3.1 Minimal Topologies

From examples such as butterfly and banyan networks, it is well known that a network with out-degrees of two suffices for routing within a logarithmic number of steps. These networks typically route from a set of n input ports to a set of n output ports in $\lg n$ layers using a total of $(\lg n + 1)n$ nodes. In comparison, we scale our network down to only n nodes and want to be able to route between any pair of them. In addition to the fast routing requirement, we also want natural load balancing and adaptability to changes in the network size. Despite these differences, we still use many ideas from these networks to develop the core of our topology.

To form the basis of our load balancing, we use the consistent hashing approach of [5] in the same fashion as Chord. Each node is mapped onto a circle by hashing their IP address into the range $[0, 1)$. Keys are assigned to nodes by hashing them onto the circle too and finding the closest node on the circle. Unfortunately, this hashing scheme will assign $\Theta(\log n)$ load to some individual nodes with high probability and the virtual node scheme suggested is not applicable since it requires an additional factor of $\Omega(\log n)$ connectivity. To remedy this, schemes such as load balancing by allowing two or more hashed locations can bring the maximum load down to $\Theta(\log \log n)$ with high probability [10].

The first outgoing edge of each node is to its closest neighbor on the circle when moving clock-wise along the circle (increasing hashes in our implementation). These edges form a ring of all the nodes of the network. One should note that one cannot do any better using only one outgoing edge - such topologies are limited to disjoint rings, rooted trees with edges pointing towards the root, and combinations thereof (i.e. the tree roots are on rings).

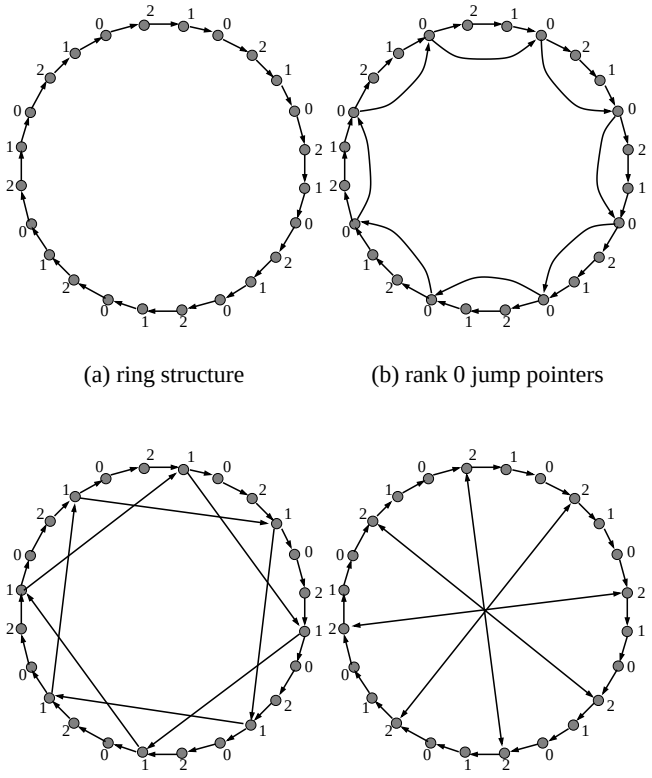
Examining butterfly and banyan networks, a common structure is the use of $\lg n$ switching layers where the "distance" that can be jumped at each layer changes in powers of two. We mimic this by forming groups of consecutive nodes and adding an edge from each node to a node in another group. The shortest such edge points to a node in the next group. The rest of the edges increase the number of groups jumped in powers of two. We call these edges "jump pointers" since they are used to quickly jump around the ring structure.

More formally, each node in a group is assigned a rank starting from zero and increasing by one. Each node of rank i has a jump pointer to the node of rank i that is 2^i groups away moving clockwise. Ranks are assigned in decreasing order when moving clockwise within a group. In the ideal case, there exists an integer m such that the number of nodes n is $m * 2^m$. In this case, there are 2^m groups of size m and the longest jump pointers (rank $m - 1$) jump 2^{m-1} groups away. When the ideal m does not exist as an integer, we prefer to use the next largest integer (i.e. rounding down) to avoid jump pointers that wrap around the ring. This structure is illustrated in Figure 1.

An alternative view for understanding the structure of our topology looks like a comb. This view is illustrated in Figure 2. In this view, nodes are approximately arranged in a grid. The left edge is a ring of the rank zero nodes. Connected to each rank zero node are the remainder of its group forming a row to the right in ascending order by rank. Jump pointers appear as connections between each row, where each jump pointer at a rank i moves 2^i rows. From this point of view, search proceeds by moving to the right where the jump pointers are the longest and then moving back to the left as the jump pointers extend too far.

If rows in the comb view are considered as a whole, their jump pointers collectively act similarly to the finger tables of Chord. That is, the jump pointers of a row provide jumps of $1, 2, 4, 8, \ldots$ groups while the finger tables provide jumps over $1/2, 1/4, 1/8, 1/16, \ldots$ of the hash space. Both schemes provide jumps with ranges increasing in powers of two - those of our scheme measure distances in rows of nodes, while those of Chord measure distances in the hash space.

There is now a clear search algorithm running in $O(\log n)$ rounds.



(a) ring structure (b) rank 0 jump pointers

(c) rank 1 jump pointers (d) rank 2 jump pointers

Figure 1: Minimal structure in our topology

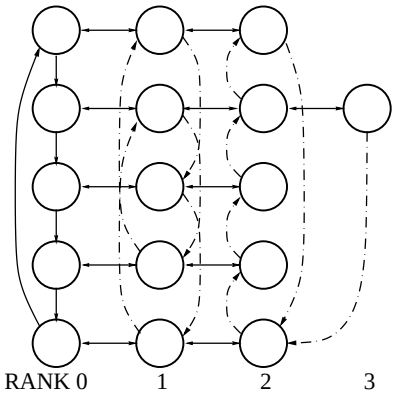


Figure 2: Comb view of our topology

Searching begins by first advancing along the ring to find a node of rank $(m-1)$ with a long jump pointer. From here, a simple gradient-based search suffices. That is, if the jump pointer does not extend past the goal, it is followed. Otherwise, the shorter pointer to the clock-wise neighbor along the ring is followed. This search algorithm runs in $O(n/2^m + m)$ rounds. Since the ideal m (as discussed above) is approximately $\lg n - \lg\lg n$, this is $O(\log n)$ rounds.

Theorem 1 *Using the minimal topology just described, searches can be formed in* $4\lg n + O(1)$ *rounds.*

Proof: This theorem follows directly from counting the number of steps moving to a rank $(m-1)$ node ($\lg n$), taking jump pointers and moving to lower rank nodes ($2\lg n + O(1)$) and linear search over the group of the desired node ($\lg n$). ■

This network topology suffices to provide fast indexing but it is non-trivial to maintain in a distributed fashion. It is also very prone to failure in the case of failures since only a small number of nodes need to be removed to disconnect a particular node. Instead of dwelling on these issues, we proceed to further augment this topology with more edges to make it more maintainable and more robust to failure. At the same time, this added structure will simplify the organization of the ring into groups.

3.2 More Maintainable Topologies

One difficulty in maintaining a network topology with an out-degree of only two is that information about neighbors either is all from the same side (clock-wise) or takes longer to propagate through jump pointers. By simply adding an edge in the counter-clock-wise edge and making the ring doubly linked, this difficulty is significantly alleviated. In particular, much of the maintenance of the jump pointers becomes trivial. That is, for nodes that have non-zero rank, the jump pointer is simply the result of a "left,down,down,right" traversal from the comb point of view. [1]

Non-trivial aspects of maintaining the topology described so far are discussed in the remainder of this section. Maintenance of the ring topology using Chord stabilization algorithms is described in Section 3.2.1. Organization of the rings and assignment of ranks is discussed in Section 3.2.2. Finally, Section 3.2.3 discusses how insertion, deletion and repairs work within this framework.

3.2.1 Ring Maintenance

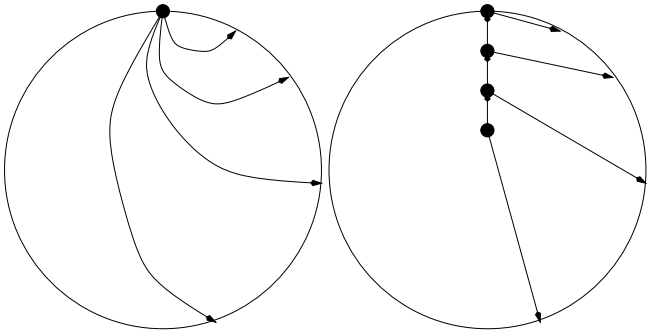
Once the ring topology becomes doubly linked, it is identical to that of Chord without the finger tables, i.e. only using the successor and predecessor pointers. See Figure 3 for this comparison. Since this is the only portion of the topology that they consider when analyzing their stabilization algorithms, it suffices to use their stabilization algorithms to maintain our ring topology.

The basic Chord stabilization routines are very simple, yet are provably good under many standards such as maintaining connectivity and reachability. Basically, each node keeps track of its successor and predecessor (neighbors along ring) by checking for coherence (following successor and predecessor edges should be an identity) and notifying neighbors of one's presence. See Figure 4 for pseudo-code for the basic versions of these functions.

3.2.2 Group Maintenance

Once the ring topology has been constructed, the next key component to maintaining the jump pointers is to estimate the value of m. Once m is estimated, we can divide the ring into groups

[1] This is essentially the well-known pointer jumping technique in parallel computing [4].



(a) Chord topology (b) Our topology

Figure 3: Visual comparison between the finger tables of Chord and the jump pointers of our topology

```
x = successor.predecessor
if self.hash < x.hash < successor.hash
    successor = x
notify(successor)
```

(a) Chord stabilization function

```
if predecessor.hash < x.hash < successor.hash
    predecessor = x
```

(b) Chord notify function

Figure 4: Pseudo-code for Chord stabilization

and setup jump pointers. Once m is determined, ranks can be assigned by choosing a first node of rank zero and letting the remaining nodes deduce their rank from that of their clock-wise neighbor. See Figure 5(a) for the pseudo-code. Given ranks, groupings and jump pointers follow naturally.

In our experimental implementation, we found that attempting to estimate network sizes using the coverage of hash space by jump pointers tended to give wildly varying results. Briefly, the circular dependency between jump pointer construction and size estimation caused both to fluctuate as partially built jump pointers changed size estimates which caused the jump pointers to be reorganized again. Since this is a distinct problem in its own right, we defer this question to Appendix A where we present a novel solution based on a distributed version of skip lists [13] and the classic pointer doubling approach [4]. For the rest of this discussion, we assume that the network size is known with sufficient accuracy to pick an appropriate m [2] within $O(\log n)$ rounds of any changes at the cost of a constant number of edges per node.

To speedup the process of assigning ranks, we first divide the ring

[2]We consider the maximal integer m such that $m * 2^m \leq n$ the optimal value of m.

```
/* start assignment from 1-0 boundary of hash space */
if ring_successor.hash > self.hash
    rank = 0
else
    rank = (ring_predecessor.rank + 1) mod m
```

(a) Rank assignment without blocking

```
if is_border(self.hash, self.ring_successor.hash)
    rank = 0
else
    rank = (ring_successor.rank + 1) mod m
```

(b) Rank assignment with blocking

Figure 5: Pseudo-code for assigning ranks

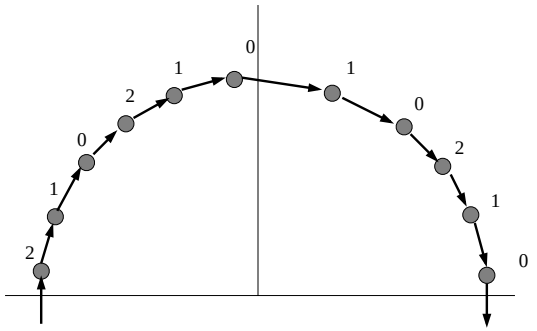


Figure 6: Fast rank assignment through blocking of nodes

into blocks of nodes covering uniformly sized arcs of the ring (in terms of hash space coverage) and organize groups within these blocks. Nodes on the clock-wise end of these blocks assign themselves rank zero allowing the remainder of the block to deduce their ranks in a number of rounds proportional to the number of nodes in the block. These blocks should contain about m^α for some $\alpha > 1$ on average so that each block contains more than one group with high probability. Larger blocks minimize the number of incomplete groups at the counter-clock-wise edges of blocks (i.e. the only groups with fewer than m nodes). This is important since nodes in incomplete groups interfere with indexing since jump pointers that would have used the missing nodes are not correct. We address this by skipping incomplete groups when constructing jump pointers so it is important that they compose only a small fraction of all nodes and are infrequent (i.e. they rarely occur consecutively). We choose $\alpha = 2$ as a balance between speed of reorganization (linear in block size) and the fraction of nodes in incomplete groups (at most $m - 1$ per block) but more experimentation is necessary.

Claim 1 *The probability a node is in an incomplete group is at most* $1/m$.

Proof: This is simply a counting argument - each block has at most $m - 1$ nodes in incomplete groups and the expected number of

groups per block is m^2. ■

To avoid instabilities and incompatible block boundaries (too close) from directly using estimates of n in calculating block sizes, we use the underestimate $m*2^m$ (since m is generally an underestimate) to calculate the ideal number of blocks and round it down to the nearest power of two. This encourages consensus about block boundaries since there is generally consensus on m even when there are significant variations in estimates of n. Additionally, using powers of two for block sizes implies that if a node within a block overestimates m by one, then the block size is evenly divided and the increased inefficiency is small. Underestimates of m do not matter since increasing m adds boundaries without removing them. Given our size estimation method, disagreement about m only lasts $O(\log n)$ rounds as updates propagate.

3.2.3 Basic Operations

Basic operations within this structure are fairly trivial. By using periodic probing of neighbors and integrating maintenance of jump pointers with Chord stabilization routines, insertion, deletion and repairs are automatically taken care of given a single connection to the network. However, one should note that insertion is much faster if a search for the node's location is performed first.

Theorem 2 *Topology changes (insertions and deletions) are performed within $O(\log^2 n)$ rounds.*

Proof: Changes to an individual row are reflected within $O(\log n)$ rounds of the stabilization routines. The $O(\log^2 n)$ arises from reranking nodes within a block. ■

Topology changes can be sped up by using smaller values of α, the blocking parameter, but are asymptotically the same as Chord insertions.

3.3 Adding Robustness

In the Chord paper, one interesting result was that their network topology was very robust to node failures. Given a constant probability of individual node failures, their network stays connected and usable with high probability. Such results are impossible within our model - high probability results for staying connected require $\Omega(\log n)$ degrees (counting both incoming and outgoing edges). Instead we show that with high probability, all but a small constant fraction of the surviving nodes form a connected component. From this result it follows that the fast indexing will be available within $O(\log^2 n)$ rounds.

In the Chord model, the high probability result is based on the probability that the $\Theta(\log n)$ immediate successors of a node are disconnected. Extra edges are maintained to each of these nodes so as to guarantee with high probability that at least one of them will survive in the event that up to $1/2$ of all nodes fail.

As with Chord, the main idea we use is to add more edges to increase the probability of staying connected. In the Chord case, the added edges covered more successors to guarantee that a pointer to the first surviving successor was available with high probability. Since we do not use our added edges to augment indexing, we merely use random edges. The graph induced by these edges is a random directed regular graph (each node has a constant number of outgoing edges) so they form an expander graph with high probability. We note that most of the literature about random regular graphs and expander graphs works with either undirected regular graphs or bipartite regular graphs, for example [3, 16], but similar proofs apply to random directed regular graphs. If we use a similar number of edges for redundancy, we obtain a result close to that of Chord.

Theorem 3 *Suppose that each node maintains $\Omega(\lg n)$ random edges and nodes spontaneously fail with probability $1/2$. With high probability, the network is still connected.*

Proof: This result follows trivially from the properties of expander graphs. First, with high probability, each surviving node still has at least d edges to other surviving nodes, for appropriately chosen d. Such a graph has a directed random regular graph with out-degree d. For sufficiently high d, these graphs are expander graphs with high probability and are therefore connected. ■

Since the graph stays connected, the stabilization routines will eventually restore it to a usable state. However, such a result violates the spirit of our work by using more than a constant number of edges from each node.

Theorem 4 *Given a directed random regular graph of degree d with n nodes. If each node spontaneously fails with probability $1/2$, there exists a connected component of size $(n/2)(1-(1/2)^{O(d)})$ with high probability.*

Proof Sketch: We sketch our proof as follows. First, we argue that there remains a connected component of size at least $n^{O(1)}$ with high probability. We then show that this connected component actually has size $O(n)$ with high probability. Finally, we show that this connected component has edges to most of the remaining nodes. We use $d = 3d'$ where d' is sufficiently high for the expander properties we invoke and edges from each node are divided into 3 groups of d', one for each step of the proof.

To show the existence of a polynomial sized connected component, we consider any d'-ary tree embedded in the graph before failures. Assuming $d' \geq 4$, limiting this tree to depth $(3/8)\lg n$ means that the tree has $\Omega(4^{(3/8)\lg n}) = \Omega(n^{3/4})$ distinct leaves with high probability (there are very few "collisions" between tree nodes since the total size is $o(n)$). Considering the losses again, each path is intact with probability $n^{-3/8}$, so $\Omega(n^{3/8})$ leaves are expected to be reachable from the root. We note that there is a constant probability that the tree is immediately disconnected (has a constant upper bound on depth) but note that there are $\Theta(n)$ roots to consider so the largest component has size $\Omega(n^{3/8})$ with high probability.

Given this polynomial sized connected component, we consider the next set of the d' edges from each node. This set of edges is an expander graph with high probability, so by expanding the component found in the previous step (using an unrelated set of random edges), we can expand it to cover $\Theta(n)$ edges in $O(\log n)$ steps.

Finally, to show that at most $(1/2)^{O(d)}$ nodes are not reachable from this component, we observe that there are $O(d'n) = O(dn)$ edges leaving this component from the third set of random edges and that they cover all but a fraction $(1/2)^{O(d)}$ of all nodes. ■

This sketch only provides loose bounds but Figure 7 shows the results of simulations to find the size of the largest connected component.

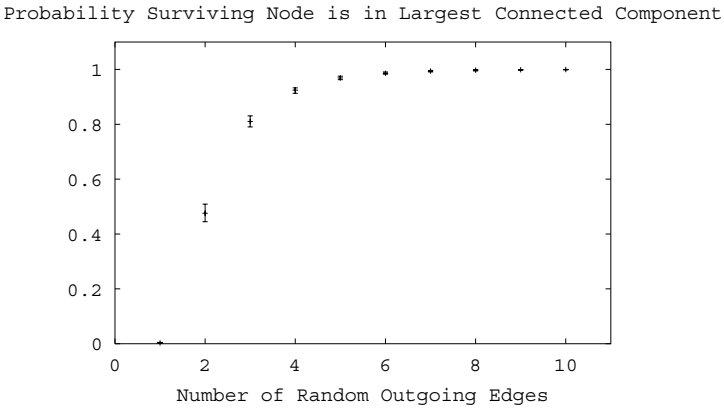


Figure 7: Robustness to node failures. Starting with 10,000 nodes, half of them were removed from the network. The error bars show the minimum and maximum values over 100 trials.

Using random edges for redundancy means that they are easy to maintain using random walks. Since the graphs are expanders, a random walk of $\Omega(\log n)$ steps suffices to randomly sample nodes of the network. An alternative to random walks is to choose a random hash value and use the corresponding node as the random edge, updating it as necessary when new nodes take over that hash value. This second approach has the advantage of not changing the edge at regular intervals, thus avoiding any problems of the large connected component described above disappearing during recovery.

An interesting consequence of choosing to use random edges is that nodes can choose how much effort they are willing to spend on robustness. That is, both the number of random edges and the frequency with which they are polled and updated can be changed without adversely affecting indexing performance. While we advocate a minimum number of random edges to backup our analysis, more paranoid users may wish to add more random edges to ensure that they stay connected to the large component in case of disaster.

3.4 Optimizations

3.4.1 Topology Based Optimizations

The most basic topological construct that enables logarithmic time searching in our scheme as described above is the jump pointer. However, there may exist situations where additional edges may be of great benefit, especially where such an edge would reduce the number of hops taken on average. It may be argued then that constant factor penalties in search time can be offset with the addition of a constant number of additional optimization pointers. We present several intuitive optimizations before presenting a more generalized optimization scheme.

Knight Moves Consider Figure 8, which depicts a node in a sample topology with its jump pointer (labeled J), and an additional pointer (labeled K) which is termed the *knight move* pointer. This pointer is directed effectively to the target node obtained by traversing the jump edge of the source node, and then the clock-wise ring pointer of the intermediate node. Thus, each node collects the node at distance two from it, in the most expensive axis of the topology- the direction of the jump pointer.

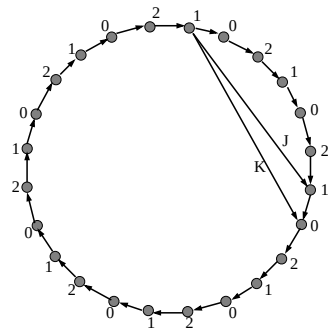


Figure 8: Knight Moves, labeled K

In a typical search, it is unlikely that the destination of the jump pointer is the target of our search - it is more likely to be some intermediate node. Further, if the jump leads to an intermediate node, it is easy to verify that the jump pointer of that node extends past the jump pointer of a node previously visited in the search. Thus, in the case of an intermediate node, we must always traverse the ring clockwise. In the average case, we save one ring pointer traversal per jump pointer traversal if we implement knight moves. Further, each transition of rows that would normally require two edge traversals, and the possibility of detouring through a less desirable section of the underlying topology would require only one hop.

The knight moves also naturally provide a bound on the number of hops to reach a target row. As each knight move decrements the rank between source and target, we can only make as many knight moves as the maximum number of nodes in a row.

It must be noted that the knight move pointer cannot fully replace the jump pointer. For example, the target node of the jump pointer may in fact be the object of the search. In this case, prematurely accepting the knight move pointer as the edge to traverse would add an expense of moving counter clock wise on the ring at the target level, when one hop would have sufficed. It is important to note, that if the jump pointer remains part of the node state, then a simple comparison of the search key to the hashed IP address of the jump pointer and knight move pointer indicates which of the two edges is appropriate.

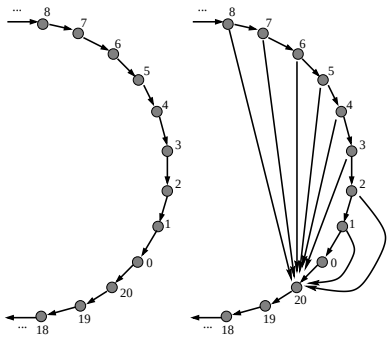
End Pointers An obvious result of the search procedure is the importance of finding the largest possible jump pointer to traverse without overshooting the target row. Many traversals of clockwise ring edges before finding this jump would be unnecessarily expensive assuming a large network. A method of circumventing this would be to add *end* pointers to nodes in the overlay. An end pointer would represent a link to the node at the end of the row, or the end of the next row, for convenience.

Search	Basic	Knight Moves	End Pointers	Binary Search	All	Chord
Finding first jump	$\lg n$	$\lg n$	1	$\lg n$	1	
Jump Traversals	$2 \cdot \lg n$	$\lg n$	$2 \cdot \lg n$	$2 \cdot \lg n$	$\lg n$	$\lg n$
Search within row	$\lg n - 1$	$\lg n - 1$	$\lg n - 1$	$\lg \lg n$	$\lg \lg n$	$O(\lg n)$
Total Cost	$4 \cdot \lg n - 1$	$3 \cdot \lg n - 1$	$3 \cdot \lg n - 1$	$3 \cdot \lg n + \lg \lg n$	$\lg n + \lg \lg n$	

Table 1: Search Algorithm Component Cost, Worst Case

Search	Basic	Knight Moves	End Pointers	Binary Search	All	Chord
Finding first jump	$\frac{1}{2} \cdot \lg n$	$\frac{1}{2} \cdot \lg n$	1	$\frac{1}{2} \cdot \lg n$	1	
Jump Traversals	$1\frac{1}{2} \cdot \lg n$	$\lg n$	$1\frac{1}{2} \cdot \lg n$	$1\frac{1}{2} \cdot \lg n$	$\lg n$	$\frac{1}{2} \cdot \lg n$
Search within row	$\frac{1}{2} \cdot \lg n - 1$	$\frac{1}{2} \cdot \lg n - 1$	$\frac{1}{2} \cdot \lg n - 1$	$\lg \lg n$	$\lg \lg n$	$O(1)$
Total Cost	$3\frac{1}{2} \cdot \lg n - 1$	$2\frac{1}{2} \cdot \lg n - 1$	$2 \cdot \lg n - 1$	$2 \cdot \lg n + \lg \lg n$	$\lg n + \lg \lg n$	$\frac{1}{2} \cdot \lg n + O(1)$

Table 2: Search Algorithm Component Cost, Average Case



(a) Basic (b) with End Pointers

Figure 9: Hop-savings due to End Pointers

```
assign_range(min, max, node)
mid = max + min /2
if mid == node:
   No pointer required (node gets pointed to)
   return
if max == node:
   Node is responsible for range (min, max)
   Node makes pointer to mid
   return
if node < mid:
   range(min, mid, node)
else
   range(mid+1, max, node)
```

Figure 10: Pseudo code for assigning "binary search" pointers

Thus, when a node joins a row, a message must be passed along the nodes of the row, allowing the update of each node's state to reflect the addition. At search time, a node would immediately route its query to the end of the row, and then proceed to find the longest jump link. While this increases the load of the end nodes relative to other nodes, one should observe that the load of the end nodes does not actually change - it was repetitive work by the other nodes that was cut out.

Binary Search Pointers From Table 1, a logarithmic cost is paid, once the target row of the search is found, in locating the target node on that row. A natural means of expediting this process would be to organize each row into a structure more conducive to fast searching. As the row can often be thought of as linear, a natural means of searching over a linearly organized set is binary search.

To implement such a search algorithm, we must augment our topology with an additional pointer per node. Let m represent the maximal rank of the row. Thus, a node at rank 0 is responsible for maintaining a pointer to the node at $\frac{m-1}{2}$. Each node must determine which range it owns, and the mid point of that range to which it must maintain a pointer. Using the pseudo-code algorithm of Figure 10, a node can determine its midpoint and range.

3.4.2 Analysis

Tables 1 and 2 detail the cost of a worst and average case search, respectively. Note that the table is not fully accurate: $\lg n$ metrics for our results are based on the size of a row, and do not reflect additive factors such as $-O(\log \log n)$. Thus the actual metrics are slightly smaller than depicted above.

The important observation is that the addition of knight moves bounds the number of jumps taken between rows to the size of the row, while maintaining end pointers eliminates the need to traverse to find the largest jump pointer (or knight move in the case of utilizing both end pointers and knight moves). Thus, with the addition of just two edges per node, we reduce our search time by a factor of two. Our protocol, when implemented using these two simple optimizations, yields search times only three times that of Chord on average, while requiring only a fixed node degree (five without random edges). Adding one more edge for binary search and some overhead to maintain it drops search times further to only twice that of Chord.

At six edges per node, Chord can only support 64 nodes or allow search times to increase to $O(n^{1/6})$. In contrast, our scheme can handle any number of nodes with the same asymptotic speed. Furthermore, we can further improve the constant factor an arbitrary amount by "expanding the horizon", of which the knight move is

a selective example, all while keeping some (significantly larger) constant number of edges. Note that we are not counting either the random regular edges of our topology or the $\Theta(\log n)$ successors of the Chord topology in this comparison - this comparison is based purely on edges used for indexing.

3.4.3 General Optimizations

So far in this section, we have discussed a few natural optimizations of our scheme, though we do not claim these techniques are best. They do indicate a more general optimization schema that may be used to optimize, for any constant factor, any such connected topology. Such an optimization strategy would allow, with a constant amount of additional connectivity, to reach any desired node in $\epsilon \log n + o(\log n)$ steps for any constant $\epsilon > 0$.

To give a brute force version of such schema, consider a topology where you wish to achieve a constant factor speed-up of k. We construct a new topology, where each node is connected to all other nodes within distance k from it in the original topology. This brute force scheme gives the desired speed-up, though it is impractical in terms of the very large constant factors involved. The examples of topological optimizations listed above are examples of the efficacy of choosing, strategically, just a few of these additional network connections. It remains an open question, what the best method to optimize speed given a number of connections, or what the minimum degree of connectivity for a desired speed is.

4 Conclusions

Solving the problem of indexing is a necessary component of any viable peer to peer networking application. The method we presented in this paper solves this problem: given a key and a querying node, learn the end system location of the associated value for that key. Our method is completely decentralized, avoiding a single point of failure, and fully scalable. By building a topology based on the Chord protocol [17], our solution to this problem results in a topology that naturally supports fast searching and insertions while requiring only a fixed number of connections to peers. Further, our protocol contributes a means of performing logarithmic time searches over a fixed degree network, while avoiding network unfriendly per-query flooding.

5 Acknowledgments

We would like thank John Byers for key suggestions at the beginning of this work and critical feedback later. We would also like to thank Mina Guirguis and Trevor Macdowell for their feedback.

References

[1] Gnutella. http://www.gnutella.org.

[2] Kazaa. http://www.kazaa.com.

[3] FRIEDMAN, J. On the Second Eigenvalue and Random Walks in Random d-Regular Graphs. *Combinatorica 11* (1991), 331–362.

[4] HILLIS, W. D., AND GUY L. STEELE, J. Data parallel algorithms. *Communications of the ACM 29*, 12 (1986), 1170–1183.

[5] KARGER, D. R., LEHMAN, E., LEIGHTON, F. T., PANIGRAHY, R., LEVINE, M. S., AND LEWIN, D. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *ACM Symposium on Theory of Computing* (May 1997), pp. 654–663.

[6] LAW, C., AND SIU, K.-Y. Distributed algorithms for broadcasting and searching on peer-to-peer topology. Draft, 2001.

[7] LAW, C., AND SIU, K.-Y. Distributed construction of random regular graphs. Draft, 2001.

[8] MALKHI, D., NAOR, M., AND RATAJCZAK, D. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st annual ACM symposium on Principles of distributed computing* (2002), ACM Press.

[9] MARKATOS, E. P. Tracing a large-scale Peer to Peer System: an hour in the life of Gnutella. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid* (2002).

[10] MITZENMACHER, M. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Systems 12*, 10 (2001), 1094–1104.

[11] PANDURANGAN, G., RAGHAVAN, P., AND UPFAL, E. Building Low-Diameter P2P Networks. In *Proceedings of the 42nd Annual IEEE Symposium on the Foundations of Computer Science (FOCS)* (2001).

[12] PANDURANGAN, G., RAGHAVAN, P., AND UPFAL, E. Building P2P networks with good topological properties, 2002.

[13] PUGH, W. Skip Lists: A Probabilistic Alternative to Balanced Trees. In *Workshop on Algorithms and Data Structures* (1989), pp. 437–449.

[14] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A Scalable Content Addressable Network. In *ACM SIGCOMM* (Berkeley, CA, 2000).

[15] RATNASAMY, S., SHENKER, S., AND STOICA, I. Routing Algorithms for DHTs: Some Open Questions. In *Electronic Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)* (2002).

[16] SIPSER, M., AND SPIELMAN, D. A. Expander Codes. In *IEEE Symposium on Foundations of Computer Science* (1994), pp. 566–576.

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

A Size Estimation

To estimate the size of our network, we construct an overlay based on skip lists [13] and pointer jumping techniques []. Skip lists are a randomized data structure meant to replace balanced search trees. In particular, their performance for insertions and deletions is provably within a constant factor of any balanced tree.

Skip lists work by randomly assigning each inserted element a rank k determining how many pointers it has to skip ahead. This rank k is geometrically distributed in powers of $1/2$ so $1/2$ of all nodes

have rank 1, $1/4$ have rank 2, etc. The ith pointer of each node then points to the next node with rank i or greater. Search proceeds by always taking the longest pointer that does not pass the desired value.

We distribute skip lists over several nodes in a fashion similar to how we distribute the finger tables of Chord. First, each node picks a bit uniformly at random specifying whether it is the beginning of a simulated skip list node. A node that picks 1 for the bit is responsible for the 1st pointer of each skip list. The other pointers are handled by the nodes picking 0 for the random bit up until the next node picking one. This gives the simulated nodes the same rank distribution as that of a normal skip list and allows edges update in a fashion similar to that of the jump pointers. The key difference in these updates are that the distance between two connected nodes of the same rank is expected to be constant (2) when traveling along pointers at the rank below or along the ring for rank 1 pointers.

Once this structure is in place, it is trivial to exactly calculate the number of nodes assuming no joins or departures. This done in a bottom fashion by rank. Each node of rank i keeps track of the pointers and distances to the next node of rank 1, the next node of rank i, and the next node of rank $(i+1)$. Each of these distances is easily calculated based on the distances of successor nodes. Estimates of the total size of the graph are then taken from the next node of rank $(i+1)$. If no such node exists, the node has the highest rank. If there is only one node of the highest rank (the next node of the same rank is itself), the corresponding distance is the number of nodes in the network. If there are multiple nodes of the highest rank, the distance to the next node of that rank and the portion of the hash space between them is used to estimate the total. [3] Since the highest rank is $O(\log n)$ with high probability, estimates are built up and propagated in $O(\log n)$ rounds.

Note that distributed skip lists are not appropriate for routing since nodes with high rank pointers will be forced to handle a constant fraction of all traffic. However, when used as part of an indexing scheme based on consistent hashing which provides fast searches, it can provide quick estimates of the network size at the cost of only three edges.

[3] With slightly more complication, one could also traverse all of the nodes at the highest rank but most applications will expect frequent node arrivals and departures and the extra accuracy is not necessary anyway.