# Cone: Augmenting DHTs to Support Distributed Resource Discovery

Ranjita Bhagwan, George Varghese and Geoffrey M. Voelker
Department of Computer Science and Engineering
University of California, San Diego

## 1 Introduction

Together with the revolution in peer-to-peer *content sharing*, as exemplified by Napster and Gnutella, there has been a parallel revolution in peer-to-peer *distributed computing* as exemplified by SETI@home and Javelin. As with content sharing, P2P computing networks have evolved from centralized to distributed resource discovery. Thus a fundamental problem in P2P computing is to scalably answer relational queries of the form, "Find a resource of size X or higher," in distributed fashion. Networked server systems can also benefit from such queries. For example, in high-bandwidth content delivery networks it would be beneficial to have the ability to locate a server with sufficient bandwidth to deliver a large object. Another example is that of wide-area gaming systems. If these systems supported these queries, players would be able to locate and join a game server with minimum load to ensure better response time.

Current P2P computing systems such as Javelin use flooding together with heuristics to locate resources, reminiscent of Gnutella's content discovery methods. By contrast, recent advances in P2P content discovery via DHTs have demonstrated scalable $O(\log N)$ algorithms such as Chord [14], Pastry [11], and CAN [10]. However, DHTs have been largely limited to exact match queries, with some later work (e.g., skip graphs [2]) generalizing to prefix match and simple range queries. As a result, it seems difficult to coerce DHTs into scalably answering resource discovery queries.

A fundamental abstraction for resource discovery is to find any resource above a given size. In terms of abstract data structures, this function is provided by a Heap. Thus a fundamental question we pose in this paper is to ask whether DHTs can be generalized beyond exact and range lookups *to also provide heap functionality.* The answer to this question has both theoretical and practical ramifications. On the practical side, a positive answer can offer similar benefits to P2P distributed computing that scalable DHTs such as Chord offer to P2P content sharing. On the theoretical side, a positive answer opens the door to investigating other distributed data structures with richer abstract operations.

One can always approach the problem of providing a distributed heap from a clean slate, ignoring past work in scalable DHTs. On the other hand, there can be considerable intellectual leverage building on the techniques already developed within DHTs. Building on existing DHTs also has the advantage of adding new functions (e.g., heap functionality) without losing useful existing functions (e.g., exact match).

Thus in this paper we introduce the notion of *augmenting DHTs*: starting with a DHT such as Chord or CAN as a substrate, we show how to augment the DHT with additional information to support the added functions. We are, of course, strongly influenced by the analogy to augmenting binary trees, in which *centralized* binary trees [4] are enhanced to provide rank operations by augmenting tree nodes with auxiliary information such as subtree sizes. However, to the best of our knowledge the general question of augmenting *distributed* data structures has not been explored.

Our general strategy is to start with a Chord-like ring of identifiers, and then to build a trie on these identifiers leading to a structure that resembles a cone. We then augment the trie to contain additional information (e.g., the max resource value in the subtree).

The four main contributions of this paper are:

**1.** We suggest that both P2P Content Sharing and Computing can benefit from a unified *perspective* via distributed data structures with suitably chosen abstract operations.

**2.** We introduce the generic *approach* of augmenting distributed data structures. Our approach augments a DHT and builds a prefix trie on node IDs and adds augmenting information to nodes. The augmentation can use any aggregate operator on keys (Max, Min, Sum, etc.).

**3.** We apply the augmentation approach to introduce a *new distributed data structure* called a Cone. Cones support a variety of queries to locate resources, such as locating a resource of maximum size or a resource of at least a given size . For a DHT with $N$ nodes and IDs of $m$ bits, queries and updates take an expected-case $O(\log N)$ and worst-case $O(m)$ messages.

**4.** We provide an *analysis* of the load-balancing properties of Cone with minimal assumptions made on the probability distribution of resources. Although Cone is essentially a lightweight tree, we show that it has the same small load imbalance factor as a DHT (i.e., log $N$). We also discuss several techniques for balancing load in Cone, and evaluate one via simulation.

The rest of the paper is structured as follows. Section 2 describes related work. In Section 3, we describe the Cone data structure. In Section 4, we describe the Cone operations and provide bounds on the number of messages used for the operations. In Section 5, we calculate the load imbalance factor in Cone and discuss several load-balancing techniques for improving it. Finally, in Section 6 we summarize the contributions of this paper and describe future work.

## 2 Related work

Iamnitchi and Foster [7] propose heuristic solutions for decentralized distributed resource discovery, but heuristic solutions may not scale well to a large number of resources. [1, 12, 13] modify DHTs to do resource discovery by mapping key ranges to different nodes in a DHT, with each node in the DHT keeping track of all resources that fall within its key range. These solutions have load-balancing problems since it is possible that a large number of resources have the same key value, and this could lead to overburdening some nodes in the DHT. Also, node joins and leaves can cause a substantial amount of index copying and maintenance overhead. Our approach circumvents these problems by not using distributed indices. Each host is responsible for maintaining its own key value. Systems such as Astrolabe [15], PIER [6] and INS/Twine [3] also maintain distributed indices, but their concentration is not on supporting range-based queries and heap functions.

SOMO [16] uses a tree-like overlay on DHTs to perform metadata gathering and dissemination. Cone is an augmentation to DHTs, and not a DHT overlay. Hence it does not require DHT-based lookups for operations other than node join and leave. Moreover, SOMO in its current form does not support range-based searches or heap functions.

Skip graphs [2] and SkipNet [5] can provide range searches which can be used for resource location. In contrast, Cone can augment almost any DHT and support any aggregate operator on keys. It appears fundamentally difficult to modify skip graphs or SkipNet to also perform aggregate operations on keys because there is no aggregating node (as in a tree) for a level, but rather a list of nodes. Also, skip graph operations in the worst case can take $O(N)$ messages, while Cone operations require $O(m)$ messages in the worst case, where $m$ is the number of bits in the DHT identifier.

## 3 Data structure

As with a heap, the Cone data structure is a tree of nodes with an aggregation key at the root of each subtree. The aggregation key can be the result of any aggregation operation, such as Max, Min, Sum, etc., but in the rest of this paper we use the Max aggregate operator for clarity of exposition.

Cone differs from a standard heap in two ways, however. First, the same physical node can be the root of all logical subtrees to which it belongs. Second, the underlying tree is a trie, and hence may not be perfectly balanced. We exploit these differences to smoothly integrate Cone with DHTs. In this section, we describe the Cone data structure and how it is integrated with a DHT.

Cone uses a simple binary tree-based data structure with the following property. A non-leaf node in Cone is set by using the following formula:

$$N = \begin{cases} left(N) & \text{if } left(N).key > right(N).key \\ right(N) & \text{otherwise} \end{cases}$$

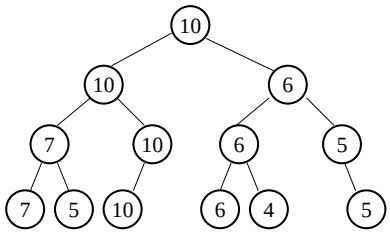


Figure 1: A basic Cone tree.

A non-leaf node is set if and only if at least one of its children exists. This formula implies that if a node $N$ is at level $l > 0$ in the tree, it is one of its own children. Further, it also implies that node $N$ exists in all levels $0, \ldots, l$ of the tree.

Figure 1 shows a simple example of a Cone tree for finding the node with the maximum key. At the lowest level, two sibling leaf node keys are compared and the node with the larger key is made the parent. Next, the siblings at the next level are compared; the larger one becomes the parent, and so on. Finally, the root is the node with the largest key.

We now describe how the Cone structure can be integrated with a DHT. Assume the DHT uses an $m$-bit ID space. The Cone data structure starts with a trie built over the ID space of the DHT. The trie has $m$ levels with the DHT forming the lowest level. When a node joins the DHT, it also joins the lowest level of Cone, i.e., nodes form the Cone tree leaves. Their positions in Cone are determined by their (random) IDs. This ensures that node joins occur at random points in the Cone tree, which is essential for load-balancing. Cone is a dynamic data structure and nodes can join and leave at any time, just as they join and leave the DHT. Cone can also support multiple simultaneous joins and leaves to the extent that the DHT can.

Figure 2 shows an example of a 3-bit Cone/DHT structure. The shaded circles denote nodes that have joined the network with the corresponding IDs; unshaded circles represent unassigned node IDs. The tables below each node show the state used to maintain the Cone data structure.

For each node, the table consists of $m$ entries, one for each level of the tree. Each entry represents an edge in the tree, and holds the *IP address* (not the DHT ID) of the node which is the immediate parent of the node at that level. If a node is a parent to a different node at a given level, the table entry for that level also contains the IP address of the child node. A "-" represents an edge from a node to itself. For example, node 1 has an edge to itself from level 0 to level 1. This is because node 1 does not have an immediate sibling, so by default it is its own parent. However, since node 1's key (5) is less than node 2's key (10), at the second level node 1 is the child of node 2. Hence node 1's table holds the IP address of node 2 in the second entry. The second table entry for node 2 also maintains the IP address of node 1, its immediate child: node 2's second-level table entry is "-, $IP_1$", denoting that node 2 is its own parent and that its immediate child, other than itself, is node 1.

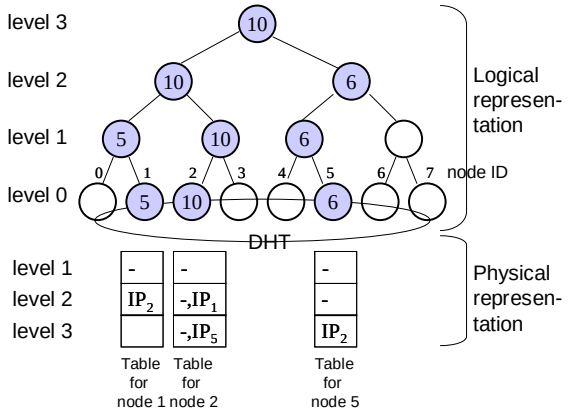


Figure 2: This fi gure shows how a Cone tree is constructed from 3 nodes, starting from a DHT with a 3-bit ID space. The 3 nodes have IDs 1, 2 and 5, with key values 5, 10 and 6 respectively. The tree is the logical representation, while the tables for each node show how the data structure is physically maintained in distributed fashion.

# 4 Cone operations

In this section, we show how Cone can be maintained in a completely distributed fashion using $O(\log N)$ state at each node. Cone supports four main operations: join, leave, find and change key. We describe these in the following subsections. Note that in the following figures, we have replaced the table entries of form $\text{IP}_x$ to $X$ for clarity.

## 4.1 Join

When a node $R$ joins the network, it joins the DHT using the DHT's join operation. In addition, it also joins the Cone tree by first using the DHT to find a node $S$ with which it shares the longest common prefix (its neighbour). Using $S$, $R$ finds the least common ancestor (LCA) in the tree that it shares with $S$. This is the point at which $R$ joins the Cone tree. Figure 3 shows an example. A node with ID 0 (binary:000) joins with key 20, as shown in Figure 3(a). Its neighbour, with which it shares the longest common prefix, is node 1 (binary:001). By comparing prefixes, node 0 knows that its LCA is at level 1 of the tree, or at the 00* position of the trie, which in this case is node 1 itself.

Once the LCA is found, the "trickling" phase of the insert begins. The new node trickles up starting from the LCA, and its key determines the level up to which it trickles. Going back to the example, the two nodes 0 and 1 compare their keys and find that node 0 has a larger key (20) than node 1 (5). Hence the parent of the two nodes should now be node 0. Node 0 enters a "-, 1" in its first table entry, denoting that it is its own immediate parent, and its immediate child at level 1 is node 1. Likewise, node 1 needs to change its table to reflect that its parent at level 1 is node 0. It therefore replaces the "-" in its first table entry with "0", as shown in Figure 3(b).

At level 2, node 0 knows that it has to compare its key with node 2 by referring to node 1's table. In doing so node

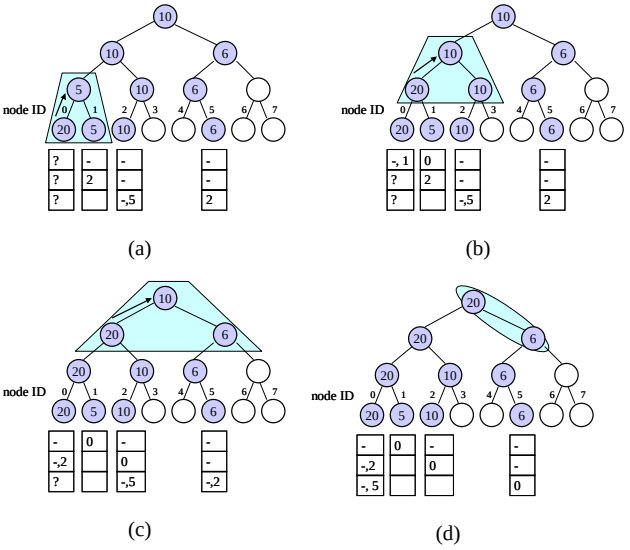


Figure 3: The Cone join operation.

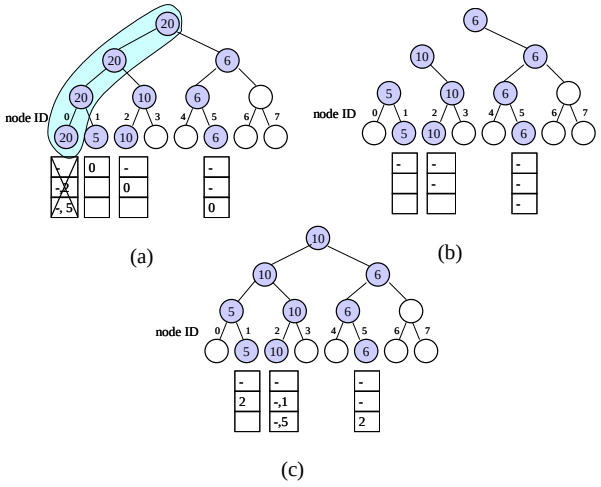


Figure 4: The Cone leave operation.

0 finds that it has a higher key than node 2, and a change similar to the previous step results in Figure 3(c). Similarly, at the third level, node 0 takes over as the root since it has a higher key than node 2. From the third-level table entry of node 2, node 0 learns that node 5 was an immediate child of node 2. As shown in Figure 3(d), it changes its own table to make node 5 its child at level 3, and informs node 5 that it is now its parent at level 3. Consequently, node 5 changes its third-level table entry from 2 to 0. This concludes the join operation.

**Complexity:** For a DHT with $N$ nodes and IDs of $m$ bits, node joins consisting of the combination of finding the least common ancestor and the trickling take an expected-case $O(\log N)$ and worst-case $O(m)$ messages.

## 4.2 Leave operation

When a node leaves the network unexpectedly, it can momentarily create up to $m$ disconnected components of the Cone tree. This is the worst case, which happens if the root

leaves. Consider the previous example tree shown in Figure 4 again. Suppose the root leaves, as shown in Figure 4(a). This creates three disconnected subtrees of the Cone tree, shown in Figure 4(b), which need to be reconnected. The roots of these subtrees detect (by timeouts) that their parent has left and make themselves their parents by changing their tables. The stabilization of Cone after arbitrary failures relies on the stabilization of the underlying DHT together with a simple tree stabilization mechanism in which each node periodically checks for and corrects (if necessary) its parent. As shown in Figure 4(b), node 1 becomes its own parent at level 1, node 2 becomes its own parent at level 2, and node 5 becomes its own parent at level 3.

The reconnect operation proceeds as follows. Each disconnected subtree root finds its parent using the DHT and re-attaches to it. Node 1, which is the root of the subtree consisting of nodes 0 (binary:000) and 1 (binary:001), or what we call the "00* subtree", needs to find its parent at level 2, which is the node at position 0* in the trie. To do so, node 1, using the DHT, looks up any node in the neighbouring 01* subtree (either node "2" or "3"). In this case, node 1 discovers node 2 (binary:010). Node 1 then uses node 2's table to trace back to the node at the 0* position in the trie, which in this case is node 2 itself. Figure 4(c) shows this case. In this way node 1 can reconnect to node 2, its parent at the second level, and the two nodes make appropriate adjustments to their table entries. Similarly, to reconnect to the main tree, node 2 finds node 5 and becomes its child at level 3. In the example, however, node 5 has a smaller key than node 2. Consequently, node 2 takes over as root after the nodes make the required changes to their respective tables.

**Complexity.** The reconnect phase for every disconnected subtree takes expected $O(\log N)$ messages. This is because the DHT lookup to find a node in the closest subtree takes $O(\log N)$ messages, and the trace-back to find the point of reconnection also takes $O(\log N)$ messages. The expected number of disconnected subtrees is $O(\log N)$. Hence the expected number of messages for handling an involuntary leave is $O((\log N))^2)$. However, the reconnect phases can be performed in parallel so that the entire delete operation takes as long as the longest reconnect phase, which is $O(\log N)$. Note that if a node terminates gracefully, the leave operation can be implemented using $O(\log N)$ messages.

### 4.3 Find operations

The Cone data structure supports finding a node containing a resource greater than a specified threshold by starting from any node in the DHT, and tracing up the tree until the search reaches a node satisfying the given condition. At this point the search terminates, requiring an expected-case $O(\log N)$ messages. Note that Cone naturally supports finding the largest value node, a traditional heap operation.

### 4.4 Change key

Changing the key, or key update, can be gracefully handled in Cone using expected $O(\log N)$ messages. A change in key value of a node can cause it to be higher than that of its parent in the Cone tree, or lower than its child. Thus the node either trickles up (in the former case) or down (in the latter case) until the Cone property is restored. Note that change key ($O(\log N)$ messages) is much more efficient than node deletion ($O((\log N)^2)$ messages). This is desirable since we expect key changes to be much more frequent (as resources get used and freed) than involuntary node failures.

## 5 Load balancing

As in DHTs, some nodes in Cone (nodes at higher levels) will experience more load than others. For DHTs like Chord the expected maximum imbalance in the number of items stored by two random nodes is $O(\log N)$ [9]. Thus in Chord, assuming uniform access to items, the ratio of the maximum to the minimum load experienced by any two nodes is also $O(\log N)$. We call this ratio the *load imbalance factor*.

In this section, we first describe two kinds of load on a node in Cone. Next, we describe a novel analysis that shows that despite the Cone data structure being a binary tree, the load imbalance factor for both aspects of load-balancing is the same as that of a DHT, i.e., $O(\log N)$. Moreover, the analysis does not assume any specific distributions of resource values (keys) or query values. Finally, we outline several load-balancing techniques that can be used to further improve load balancing in Cone.

As with any system used for distributed resource discovery, there are two aspects to load balancing in Cone:

**Data traffic:** The load of query satisfaction should be shared equally by all nodes that are capable of satisfying the query. Let $N_1, \ldots, N_k$ have key values greater than $q$. Let $P_D(N_1)$ be the probability that $N_1$ satisfies a query `find > q`. Ideal data load balance is achieved when for any $i, j \leq k, i \neq j, P_D(N_i) = P_D(N_j)$.

**Control traffic:** The amount of control traffic passing through all nodes in the system should ideally be the same. Let $P_C(N_i)$ be the probability that, for some query, a control message is sent to node $N_i$. Ideal control load balance is achieved when for any $i, j, i \neq j, P_C(N_i) = P_C(N_j)$.

In the following subsections we show that, in both data and control load balancing, the imbalance factor is $h = \log N$, where $N$ is the number of nodes in the system. In our analysis, we assume that all nodes generate queries using the same distribution, and have the same frequency of requests. Note that the only assumption we make is that the resource value probability distribution is the same as the query value distribution. As a result, our results are equally valid for resource distributions that range from a uniform distribution to a power law.

Below, we assume for simplicity that the number of nodes in the Cone tree, $N$, is equal to the total number of IDs allowed in the DHT, which is $2^m$. However, the results generalize to provide identical results for the more general case that the number of nodes is smaller than $2^m$.

## 5.1 Data traffic load

The worst-case data traffic load imbalance occurs when two nodes $N_1$ and $N_2$ can satisfy a given query, and one of the nodes is the root (adding more nodes only improves data imbalance). If $N_1$ is the root and $N_2$ is a leaf, the data imbalance can be $O(N)$. Fortunately, the analysis below shows that randomization of node IDs makes this scenario relatively rare.

The key to the analysis is the observation that the probability of the two nodes $N_1$ and $N_2$ being in adjacent subtrees each of height $k$ is $1/2^{n-k}$, and the ratio of the data traffic load of the two nodes is $(2^n - 2^k)/2^k$. Hence, the expected value of the data imbalance factor is:

$$\frac{2^h-1}{2^h}+(\frac{1}{2^h}\cdot\frac{2^h-1}{1}+\frac{1}{2^{h-1}}\cdot\frac{2^h-2}{2}+\ldots+\frac{1}{2}\cdot\frac{2^h-2^{h-1}}{2^{h-1}})$$

$$=(1-\frac{1}{2^h})+(h-\Sigma_{k=1}^{n}\frac{1}{2^k})=h=\log N$$

## 5.2 Control traffic load

We now calculate the maximum control traffic imbalance factor in the Cone tree. For this analysis, we assume that the key value distribution is the same as the query value distribution, and that this distribution is continuous.

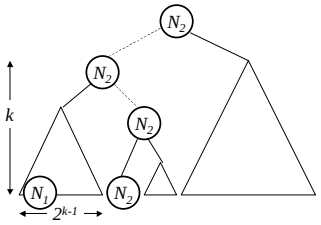


Figure 5: Control traffic load imbalance in Cone.

Consider the Cone tree depicted in Figure 5. The probability that any query originating at node $N_1$ will reach its ancestor node $N_2$ at level $k$ is equal to the probability that the query value is larger than the key values of all $2^{k-1}$ nodes in the subtree of height $k-1$. This is the same as picking $2^{k-1}+1$ (where the extra one represents the query) samples from a distribution, and estimating the probability that one of them (i.e., the query) is the maximum.[1] By symmetry any sample could be the maximum with equal probability, and hence this probability is $1/(2^{k-1}+1)$.

Next, the number of nodes from which queries can reach $N_2$ at level $k$ is $2^{k-1}$. Recall that $N_2$ is one of its own children (its right child in Figure 5 at level $k$), and that no requests come to $N_2$ at level $k$ from the right subtree since all request messages to it from this subtree are accounted for at earlier levels. Hence the expected number of queries reaching $N_2$ at level $k$ is that coming from only the left subtree, which is $2^{k-1}\cdot 1/(2^{k-1}+1)$. Consequently, the root of the Cone tree has to handle an expected number of $\Sigma_{k=1}^{h}(2^{k-1}/(2^{k-1}+1)) < h$ query messages since the root is present at every level of the tree. On the other hand, a node that occupies only a leaf position in the Cone tree needs to handle an expected value of 1 control message. Hence the control imbalance factor is $h = \log N$. We performed extensive simulations of the basic Cone data structure that confirmed the results of our analysis.

If the key distribution and the query distribution are distinct, we can no longer rely on symmetry and the results will depend on the specific distributions chosen. However, the basic framework of the analysis can still be reused.

## 5.3 Load balancing techniques

Finally, we discuss several techniques for improving the balance of load in Cone. In our analysis we have assumed that the nodes pick IDs randomly and are therefore evenly distributed in the ID space. This may not always be the case in practice. For example, an adversary can take over some part of the ID space and advertise very small resource values. In this scenario, both control-traffic and data-traffic load can be severely imbalanced.

One technique to reduce such imbalance is to use what we call "random fingers". In this technique, when a query is made at a node $N_1$, it first checks if it satisfies the query. If not, it tries $f = \log N$ other random nodes in the DHT to see if they satisfy the query. If not, the query is propagated up the Cone tree.

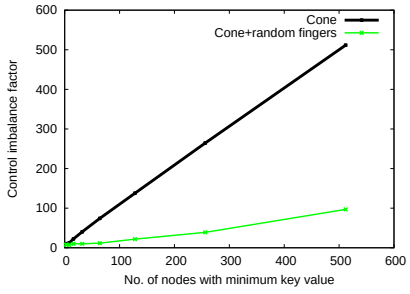


Figure 6: Improvement in the control load imbalance factor by using random fingers.

To evaluate the effect of random fingers, we simulated a Cone network of 1024 nodes and skewed the distribution of key values among nodes to induce load imbalance. To skew key values, we randomly chose a subtree of a given size and set the values of the nodes in that subtree to the minimum key value. Figure 6 shows the effect of skewed key distributions on control traffic load experienced by nodes in Cone as the size of the minimum-key subtree increases. The dark curve shows the control load imbalance factor for the original Cone find operation, and the light curve shows the effect of using $f = 10$ random fingers. From the graph we see that, as the size of the subtree increases and the number of nodes with minimum key value increases, the control imbalance factor for basic Cone increases rapidly. However, the use of random fingers significantly reduces the imbalance factor.

[1] Note that because the distribution is continuous we can ignore the probability that the key value is *equal* to one of the $2^{k-1}$ resource values.

In addition to using random fingers, we intend to explore several other load-balancing techniques:

**Variable node degree:** The analysis indicates that both data and control load imbalance are proportional to tree height. Thus the simplest technique to reduce imbalance further is to use tries of higher radix (rather than the binary tries we used so far). Note that higher node degrees also makes search proportionately faster at the cost of making node joins and leaves more expensive.

**Caching:** Nodes at lower levels can cache previous query results to reduce control traffic on nodes at higher levels.

**Virtual servers** This technique is similar to the one mentioned in [9]. A node can create several virtual servers, the number of which depends on its key value. Doing so potentially improves the data load imbalance factor of the system.

# 6 Conclusion

In essence, Cone builds a tree over the random IDs assigned to nodes within a DHT. Compared to recent distributed data structures, this seems very straightforward. Yet there are possibly unexpected aspects to Cone. First, consider building a tree. How should nodes find their positions in the tree? Choosing a trie of given radix provides a deterministic answer; further, the DHT provides the trie building facility by which a node efficiently finds another node that shares the same initial set of bits, while dealing with holes in the ID space that appear in the trie.

Second, and more importantly, the standard arguments against trees (compared to richer hypercube-like interconnections such as Chord and Pastry) are issues of *load balance* and *fault tolerance*. For load balance it may appear that all requests must pass through the root, leading to a $O(N)$ load imbalance. However, random assignment of resources to tree leaves (regardless of resource values) is surprisingly powerful. Our analysis uses symmetry arguments to show that the expected imbalance factor for data and control is $O(\log N)$, regardless of the probability distribution of resources.

Similarly, Cone takes only logarithmic messages to add a node or change its value, and log-square messages to delete a node. By contrast, a simple augmentation of a richer data structure such as Chord (e.g., in which every Chord arc is augmented with the maximum of all nodes contained within the arc) can lead to $O(N)$ deletion scenarios. Thus the very simplicity of a tree seems helpful in enabling augmentation with efficient update properties.

Finally, our paper raises the following broader research questions. First, are there other interesting tasks besides resource discovery that are useful in grid-based and P2P distributed computing that can be solved in scalable fashion? For example, a potentially useful aggregate operator is to combine sets representing node software attributes to support queries about node diversity for fault-tolerant systems [8]. Second, are there other interesting and useful augmentations of distributed data structures besides the Max operators mentioned in this paper, and can separate operators be combined economically without the overhead of a separate Cone structure for each operator? Third, the load balance analysis in this paper are dependent on the use of the Max operator; can this analysis be generalized to other aggregation operators? Fourth, what are the system issues that arise when deploying in a real setting — for example, what other heuristics (e.g., hysteresis, artificial lowering of resource values for load control) will be needed?

To extend our evaluation beyond the simple analysis and simulations done so far, we plan to experiment with a working prototype using publicly available DHT code as a base. We intend to drive our experiments with models of resource usage patterns that are relevant to distributed computing and networks. We also intend to explore the interaction and efficient integration of multiple Cones built for different resources and operators.

# References


[1] A. Andrzejak and Z. Xu. Scalable, effi cient range queries for grid information services. In *Proc. of P2P 2002*.

[2] J. Aspnes and G. Shah. Skip graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[3] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A scalable peer-to-peer architecture for intentional resource discovery. In *Proc. of Pervasive 2002*, 2002.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, 1989.

[5] N. J. A. Harvey et al. SkipNet: A scalable overlay network with practical locality properties. In *Proc. of USITS 2003*, Seattle.

[6] R. Huebsch et al. Querying the internet with PIER. In *Proc. of VLDB 2003*.

[7] A. Iamnitchi and I. Foster. On fully decentralized resource discovery in grid environments. In *Intl. Workshop on Grid Computing*, 2001.

[8] F. Junquiera et al. Phoenix: Rebuilding from the ashes of an internet catastrophe. In *Proc. of HotOS 2003*.

[9] A. Rao et al. Load balancing in structured P2P systems. In *Proc. of IPTPS 2003*.

[10] S. Ratnasamy et al. A scalable content addressable network. In *Proc. of ACM SIGCOMM*, 2001.

[11] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[12] C. Schmidt and M. Parashar. Flexible information discovery in decentralized distributed systems. In *Proc. of HPDC 2003*, Seattle, WA.

[13] D. Spence and T. Harris. Xenosearch: Distributed resource discovery in the xenoserver open platform. In *Proc. of HPDC 2003*, Seattle, WA.

[14] I. Stoica et al. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM*, 2001.

[15] R. van Renesse et al. Astrolabe: A robust and scalable technology for distributed systems monitoring, management, and data mining. *ACM Trans. on Computer Systems*, 21(3), 2003.

[16] Z. Zhang, S.-M. Shi, and J. Zhu. SOMO: Self-organized metadata overlay for resource management in p2p dht. In *Proc. of IPTPS 2003*.