Cone: A Distributed Heap Approach to Resource Selection

Ranjita Bhagwan, Priya Mahadevan, George Varghese, Geoffrey M. Voelker
University of California, San Diego

Abstract

In this paper, we propose a new distributed heap-based data structure called Cone. Cone maintains an ordering of key values in a distributed fashion, and can support queries of the form, "Find x resources of size $> S$." Cone can be built on any routing substrate as long as the substrate supports longest prefix match-based lookups. We describe the Cone data structure, the operations it supports, and its load balancing properties. We have implemented and evaluated Cone on a 1000-node ModelNet emulation platform and a 50-node PlanetLab distributed testbed. We show that Cone has good load-balancing properties and that it is stable and reactive even when there is considerable amount of dynamism in the system.

1 Introduction

Distributed Hash Tables (DHTs) are a popular architecture for building wide-area distributed applications. They have been used as the basic building block for a number of file-sharing networks, storage systems, and content delivery networks. The root of this popularity lies in the fact that DHTs provide very desirable properties such as resilience and self-organization.

However, there are a number of other applications such as distributed computing platforms [5], distributed testbeds [14], and server selection for massively multi-player online games [7] that would benefit from using DHTs, but would not be very well supported by current popular DHT architectures. DHTs have been largely limited to exact match queries and support simple mapping and lookup operations, while these applications require more advanced functionality.

Consider the following example. A distributed computing platform needs to discover four of the most lightly loaded servers to run a compute-intensive application. DHTs in their native state do not have any global information about server load of hosts, and hence will not be able to perform this resource discovery operation. Consequently, to build peer-to-peer applications that require resource discovery of this nature, a different kind of distributed data structure is required. The data structure needs to be able to perform queries of the nature "Find x resources that have value $>$ V" or "Find the x largest resources available".

Thus a fundamental question we pose in this paper is to ask whether a distributed data structure can go 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 and other resource selection problems 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.

In this paper, we propose a new distributed data structure called Cone that implements a distributed heap to maintain an ordering of objects based on their attributes — without using an index. Cone can support ordering based on any *aggregate* operator (such as max, min, sum, union, etc.). In spite of having a tree structure, Cone has the same load-balancing properties as a DHT. Cone is an independent data structure and can be built on any routing substrate.

However, we believe that there is considerable intellectual leverage building on the techniques already developed within DHTs. Building on existing DHTs has the advantage of adding new functions (heap functionality) without losing useful older functions (exact match). 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).

Hence in this paper we also introduce the notion of augmenting DHTs — that is, starting with a DHT such as Chord or CAN as a substrate, we show an instance of how one can augment the DHT with minimal additional state to support an additional data structure and functionality. However, it is important to note that Cone does not use any of the DHT's primitives for querying: it only uses the DHT when a new node joins or an old one leaves. As a result, in practice, we believe Cone will be more efficient than solutions that use a DHT-based index.

The main contributions of this paper are:

1. To introduce the generic approach of augmenting DHTs to enhance their search capability. Our approach augments a DHT and builds a prefix trie on node IDs and adds augmenting information to nodes. We illustrate this approach for the most part using a Max operator applied to resource selection; however, we also briefly comment on the generality of this approach using other operators in Section 8.

2. To 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.

3. To 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.

4. We implement Cone and evaluate it on ModelNet, an emulation framework for distributed systems, and on the PlanetLab distributed testbed. Our experiments showcase the performance and load balancing properties of Cone, and also show how the data structure reacts to rapid changes in key values and group membership in the system.

The rest of the paper is structured as follows. Section 2 describes related work. Section 3 describes the Cone data structure and Section 4 presents 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 describe several load-balancing technique to improve it. In Section 6, we describe the system design and implementation of the Cone data structure. In Section 7, we present the results of our evaluation of the Cone system. In Section 8, we discuss some extensions that could be applied to the Cone data structure to enhance its functionality. Section 9 summarizes the contributions of this paper, describes future work and concludes.

2 Related Work

Iamnitchi et al. [11] propose heuristic solutions for decentralized distributed resource discovery; heuristic solutions may not scale well to a large number of resources. [1,16,17] 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 [20], PIER [10] and INS/Twine [3] also maintain distributed indices, but their concentration is not on supporting range-based queries and heap functions. SOMO [22] 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. SDIMS [21] concentrates on aggregating information in DHTs by mapping attribute names and values along different routes in the DHT. In contrast, Cone does not map information to the DHT ID space. In that sense, Cone is an independent data structure, only using DHTs when hosts join or leave.

Skip graphs [2] and SkipNet [9] can provide range searches that can be used for resource location. They also provide additional features such as enumerative queries. Our approach does not support enumerative queries, but it can be used to support any aggregate operator on keys (e.g., Max, Sum, etc.) 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 instead there is a list of nodes. While we focus in this paper on the Max aggregate operator, Cone can be used for other operators as well, such as set union. 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 ID.

3 Cone data structure

In this section, we describe the Cone data structure and how it is integrated with a DHT. Although Cone can be built based on any aggregate operator, for most of this paper we focus on the Max operator. We briefly suggest other aggregate operators such as union in Section 8. As with a heap, the Cone data structure is a tree of nodes, with the maximum key at the root of each subtree. However, Cone differs from a standard heap in two ways. 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.

Cone uses a simple binary tree-based data structure with the following property. For any non-leaf node N:

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

This formula implies that if a node N is at level $l > 0$ in the tree, it is one of its own children. Further, this 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

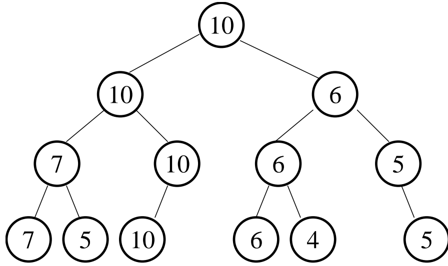

Figure 1: A basic Cone tree.

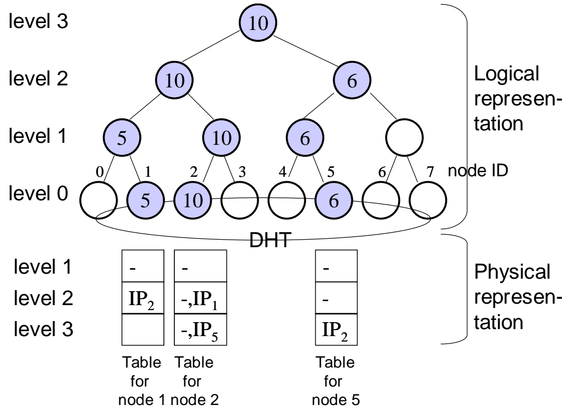

Figure 2: An example Cone tree constructed from 3 nodes overlayed on a DHT with a 3-bit ID space. The 3 nodes have IDs 1, 2 and 5, with key values 5, 10 and 6, respectively. The tree is the logical representation, while the tables for each node show how the data structure is physically maintained in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

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. Note that the Cone key is in no way related to the DHT ID of the node. The ID only decides the node's position in the DHT and its position at the lowest level in Cone; the key determines all positions that the node occupies at the higher levels of Cone.

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 stored at the node used to maintain the Cone data structure.

Each node in Cone maintains a table consisting 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. So a lth-level table entry X, Y for a node implies that at level l, its parent is x and its child, apart from itself, is y. A "-" represents an edge from a node to itself. For example, in Figure 2, 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 points to node 2. Hence its table holds the IP address of 2 in the second position. To complete the representation of this edge, the second table entry for node 2 also maintains the IP address of its immediate child, which is node 1. Hence node 2's second-level table entry is "-, IP_1", denoting that node 2 is its own parent, and its immediate child, other than itself, is node 1.

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 simplified the table entries of the form "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). Note that S may not be unique. 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 their 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; the level up to which it goes is determined by its key. Going back to the example, the two nodes 0 and 1 compare their keys, finding 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 "-, 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 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

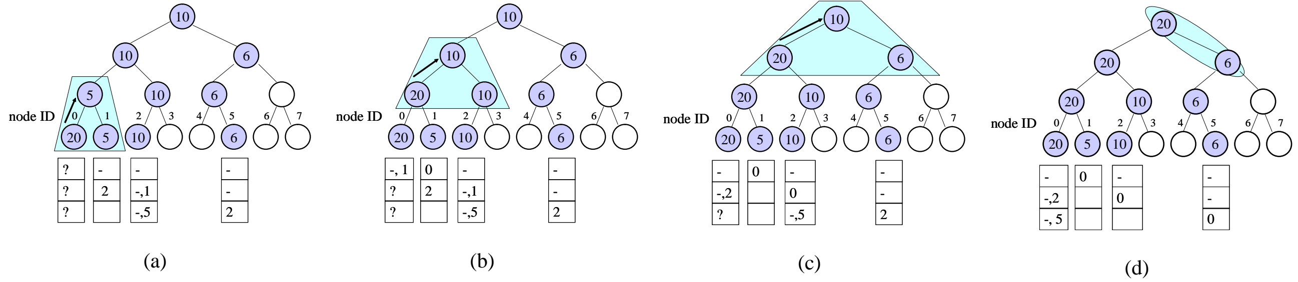

Figure 3: The Cone join operation.

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: Both finding the least common ancestor and the trickling take expected $O(\log N)$ messages, making the expected number of messages required for a node join $O(\log N)$.

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 example tree shown in Figure 3(d). 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 mechanisms 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 sub-tree 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* sub-tree", 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 a node in the neighbouring 01* sub-tree. Consequently, 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. This is shown in Figure 4(c). 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. However, in the example, 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 sub-trees is $O(\log N)$. Hence the expected number of messages for an involuntary leave to be handled is $O((\log N)^2)$. However, the `reconnect` phases can be performed in parallel, so 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 can be used to find a node containing a resource greater than a specified threshold x 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 search terminates. Hence the search completes with expected $O(\log N)$ messages. Note that finding the largest value node, a traditional heap operation, is a special case. The Cone query messages traverse IP-level hops. Hence, compared to DHT based solutions, Cone is much more efficient, since it does not use any DHT lookups to satisfy a query.

4.4 Change key

Changing the key can be gracefully handled in Cone using expected $\log N$ 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 far more efficient than node deletion ($O((\log N)^2)$ messages). This is advantageous because we expect key changes to be much more frequent (as resources get used

Figure 4: The Cone leave operation.

and freed) than involuntary node failures. For example, in a distributed computing application that builds Cone using free CPU cycles as the key, when a new job starts on a host, its key decreases, and when a job finishes executing, the key increases.

5 Load Balancing

As in DHTs, some nodes in Cone (nodes at higher levels in Cone) will experience more load than others. For DHTs like Chord [18] the imbalance factor in the number of items stored at a node is $O(\log N)$. Thus in Chord the load imbalance factor (the expected ratio of the maximum load to the minimum load faced by a node), assuming uniform access to items, is also $O(\log N)$.

In this section, we first describe two kinds of load on a node in Cone and then present an 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: $O(\log N)$. Moreover, the analysis does not assume any specific distributions of resource values (keys) or query values. Finally, we outline several techniques that can be used to further improve load balancing in Cone.

In Cone, there are two aspects to load balancing:

Data traffic: The load of query satisfaction should be shared equally by 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 $i, j \leq k, i \neq j, P_C(N_i) = P_C(N_j)$.

In the following subsections we show that, in both data and control load balancing, the imbalance factor is h, where $h = log(N)$ and 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 we make no assumptions about the probability distribution of resources; thus 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, which is 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 traffi c load

The worst-case data load imbalance occurs when two nodes N_1 and N_2 are the only two nodes that can satisfy a given query, and one of these nodes, say N_2, is at the root (adding more nodes only reduces data imbalance). If N_1 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.

In the degenerate case, N_1 and N_2 hash to the same value in the DHT. The probability of this happening is $1/2^h$. In this case, one node say N_1 does not respond to any queries other than the ones that originate at N_1 itself, while all the other data load is taken by N_2, which forms the root of the Cone tree, and therefore has a maximum height of h. Hence the difference in levels of the two nodes is h. This scenario creates the maximum imbalance factor for the the data traffic load faced by two nodes in Cone, which is $2^h - 1$. Similarly, for the case that the two nodes are adjacent in the DHT, N_1 stays at level 0 while N_2 goes up to level h. In this case too, the imbalance factor is $2^h - 1$. The probability of the two nodes being adjacent in the DHT is $1/2^h$.

We now generalize this observation. Let N_2 be the larger of the two nodes. Then, the probability that N_1 is in an adjacent sub-tree of height k is $2^k/2^h$, since the mapping of nodes to IDs is random. Hence the probability of the two nodes N_1 and N_2 being in adjacent sub-trees each of height k is $1/2^{h-k}$, and N_2 loses to N_1 at level k.

Also, the probability that N_1 satisfies the query is $2^k/2^h$, assuming the query starts at a randomly chosen point in the

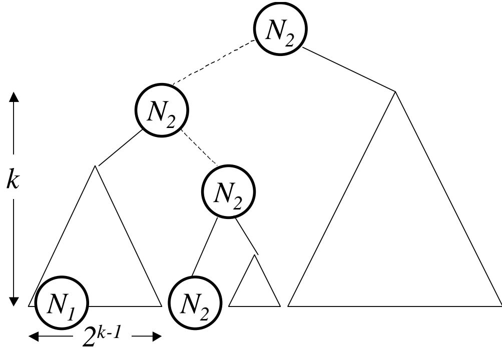

Figure 5: Control traffi c load imbalance in Cone.

Cone tree. All other queries go to N_2. Hence the probability that N_2 satisfies the query is 1 - $2^k/2^h$. This makes the ratio of the data traffic load of the two nodes $(2^h - 2^k)/2^k$. So for a fixed value of k, the expected value of the data imbalance is $(1/2^{h-k})(2^h - 2^k)/2^k = (2^h - 2^k)/2^h$.

Summing over all values of k, we can calculate the expected value of the data imbalance factor as

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

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

5.2 Control traffi c load

We now calculate the maximum control 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.

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 sub-tree 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 (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.). 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 no requests come to N_2 at level k from the right sub-tree, since all request messages to it from this sub-tree 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 sub-tree, 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 $h = \log N$.

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. It is possible that for largely varying key and query distributions, the imbalance factor will be high, with nodes closer to the root facing larger amounts of control traffic. In such cases, nodes can use the change key operation to demote themselves in the tree, which will in effect reduce the amount of control traffic they deal with. In Section 7.3, We evaluate our implementation of the Cone data structure and confirm the results of this analysis.

5.3 Load balancing techniques

In our analysis for control load we have assumed that the query distribution is the smae as the control distribution. However, this may not be the case, and the load imbalance factor can be high in certain situations. There exist several techniques that mitigate this problem. In this section, we shortly describe these techniques, a few of which we evaluate in more detail.

One technique that can be used to further balance control load 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 other nodes in the DHT to see if they satisfy the query. It could use the nodes that already exist in its routing table. If not, the query is propagated up the Cone tree. It is possible that these random queries satisfy the query, and so we do not have to use the Cone tree to find a suitable query result. Since the routing table already has the IP address of the nodes, using the random fingers approach will increase the query cost by at most f DHT hops.

Another technique to balance load is to use a *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, which is a preferable trade-off because we expect joins and leaves to occur much less frequently than queries.

Nodes at lower levels can *cache* previous query results, thereby reducing the control traffic load on nodes at higher levels. However, stale cache entries can result in repeated queries in the Cone tree, thus increasing the control load. It remains to be seen how these two opposing trends interact.

The use of *Virtual servers* to perform load-balancing has been studied in [15]. 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.

Apart from the techniques mentioned above, several

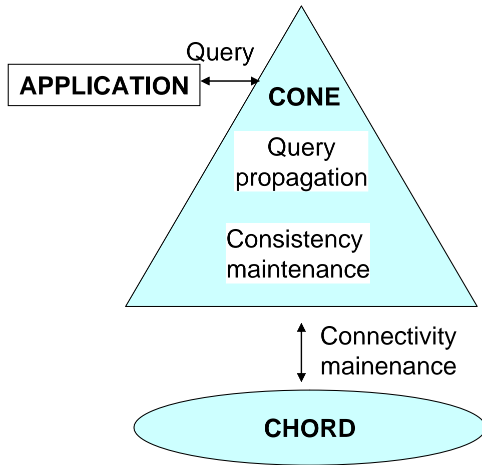

Figure 6: Global overview of the system.

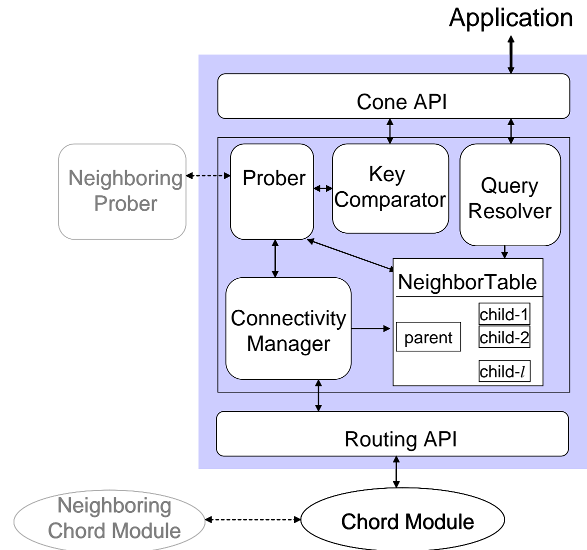

Figure 7: The Cone system design. All components within the shaded area are part of the Cone system.

application-level techniques may also be applied to the Cone structure. Let us consider an example of resource selection application where the host's key represents the amount of free resources on that node. In this case, when a host X satisfies a query made by a node Y, Y may use some resources on X thus reducing X's key value. In that case, X may drop down to lower levels in the Cone structure, thereby reducing the control and data load on that host. We evaluate one such strategy in Section 7.

6 System Design and Implementation

In this section we describe the software architecture and design that we used to implement the Cone data structure. We also describe how our system deals with disconnections caused by multiple simultaneous failures of clients.

6.1 System structure

Figure 6 gives a global picture of the system. Cone is layered on the Chord DHT and interacts with it only when nodes join or leave the system. At these times, Cone uses the DHT's *lookup* primitive to find other hosts in the system to connect with. The application uses an interface that the Cone system provides for its queries. All query traffic and data structure consistency maintenance is internal to Cone and does not use the DHT.

Now we describe the design and the various components of the Cone system that are implemented on each host in the system. The system architecture, as shown in Figure 7 consists principally of four components: the connectivity manager, the key comparator, the prober, and the query resolver. Each component is described below.

The *Cone API* supports two main operations: `set_key` and `query`. The application uses the former to change the key value of the host, while it uses the latter to query the data structure. The API can be accessed remotely, so that an application can communicate with the Cone layer of a host over the network.

The *connectivity manager* is responsible for keeping the Cone client connected to the rest of the system. It performs three major functions: it implements the `join` procedure as described in Section 4.1 and the `reconnect` procedure that follows a leave as explained in Section 4.2. When the client first joins the system, the connectivity manager requests the address of the closest neighbour from the routing API, after which it executes the `join` operation. The connectivity manager uses the prober to poll the client's parent so that it may detect when the parent leaves, disconnecting the client (and the sub-tree rooted at the client) from the rest of the system. When such a disconnect happens, the connectivity manager uses the Routing API to find another client to connect to, as explained in Section 4.2. It is also responsible for accepting and responding to `join` and `reconnect` requests from other clients in the system.

The *key comparator*'s principal function is to maintain the consistency of the data structure, i.e., it maintains the invariant that the key of the parent is always greater than or equal to that of its child. This component also uses the prober to poll the parent node. If it discovers that the parent's key is smaller than its own key, it takes over as the parent.

The main function of the *prober* is to poll the host's parent and supply information on the availability and the key value of the parent to the connectivity manager and the key comparator. In our implementation, the prober polls its parent every second. It should be noted that the prober only probes the parent, and no other node. This simplifies the system design in two main ways — first, the client does not have to store any hard state related to its children, and second, by ensuring that all probes go in one direction (upwards), we

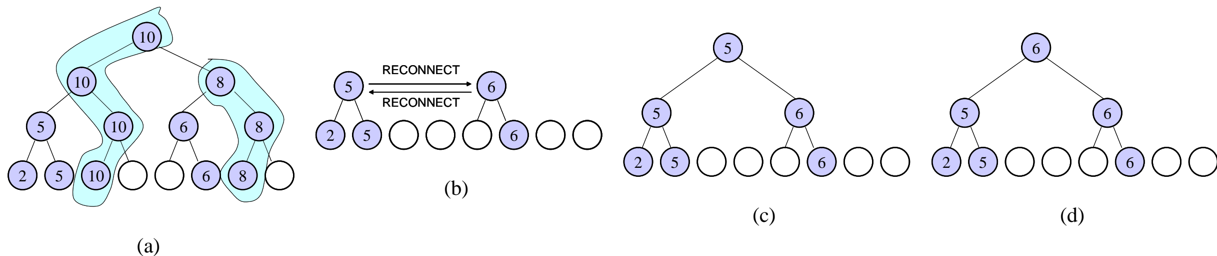

Figure 8: Dealing with simultaneous failures.

avoid several race conditions and cycles that could be caused in the process of maintaining structure connectivity and consistency. We note that the prober does not use the Chord layer for communication with other Cone clients.

The *query resolver* The Cone API forwards all query requests to the query resolver. Given the query parameters, it first checks to see if the host satisfies the query request. If not, it returns the address of the parent. Hence our current implementation of Cone supports only iterative queries. However, we intend to extend the query resolver to support recursive queries as well.

The *routing API* provides a longest prefix match-based lookup operation to Cone. In our implementation we use Chord as the routing substrate, which provides closest successor-based lookup. However, since Cone requires longest prefix match-based routing, the API implementation requires a wrapper around Chord to extract a longest prefix match from it. The Chord `lookup` implementation can provide both the predecessor and the successor of a given ID. We now show that either the predecessor or the successor will have a longest prefix match with the ID.

Claim: Consider an ID, say LR_1, and an ID LR_2 which provides the longest prefix match to LR_1. L is the longest prefix that is common to the two IDs. Now, let us say that there exists an ID A between LR_1 and LR_2 such that $LR_1 < A < LR_2$. In other words, A is *closer* to LR_1 than $LR2$ is to LR_1. Then, A also provides a longest prefix match to LR_1.

Proof: We shall prove this claim by contradiction. Suppose A did not have a longest prefix match with LR_1. Then, let A be $L'MR_3$, such that $|L'| < |L|$, which means that A differs from both LR_1 and LR_2 in a more significant bit than LR_1 and LR_2 do from each other. But if this is the case, either $A < LR_1$ or $A > LR_2$, which is a contradiction. By symmetry this argument would hold also for the case that $LR_2 < A < LR_1$.

So Chord can be used to lookup an ID LR_1 and obtain the two hosts that are $closest$ to this ID, the predecessor and the successor. Using the above claim we conclude that one of the two has to provide a host ID with the longest prefix match to LR_1.

Using this simple principle, we build a wrapper around Chord to implement longest prefix match-based lookups. This wrapper serves as the routing API.

6.2 Dealing with simultaneous failures

In Section 3, we explained how the Cone data structure recovers when a single node leaves the system. However, in reality, it is possible that multiple Cone clients may leave the system simultaneously. In this case, there could be multiple disconnected sub-trees of varying heights seeking to reconnect. To reconnect, each sub-tree root seeks a node in its neighbouring sub-tree. However, in the scenario of multiple simultaneous disconnects, it is possible that client C_A from sub-tree A sends a `reconnect` request to client C_B from sub-tree B, and vice-versa. This could potentially cause a cycle. To break this cycle, we use a global arbitrary ordering of requests — if a client C_A receives a `reconnect` request from a client C_B to which it has already sent a `reconnect` request, it takes over as parent of C_B if and only if its ID is less than that of C_B.

We show an example of such an event in Figure 8. Two nodes, with keys 10 and 6, simultaneously fail, as shown in Figure 8(a). This causes two disconnected sub-trees of the same height, and the roots of these two sub-trees both start the `reconnect` operation to each other (Figure 8(b)). This potentially causes a cycle. The tie is broken when the node with key 5 detects that it is trying to reconnect to the same node from which it has received a `reconnect` request. It realizes that there have been multiple simultaneous failures, promotes itself to root, and takes over as the parent of node with key 6 (Figure 8(c)). Following this, the key comparator detects the inconsistency in the data structure and corrects it (Figure 8 (d)). In this way, we ensure that after a transient period of disconnectivity, the data structure stabilizes and reaches a consistent state.

6.3 Implementation Status

We have implemented Cone using MIT's implementation of the Chord DHT, and ibasync, part of the SFS toolkit [13]. The current implementation consists of all the system components described above. While we have implemented the Cone data structure in its totality in our prototype, there are several optimizations that we have not yet implemented, but intend to in the near future.

7 Prototype Evaluation

We evaluated the Cone implementation in two different network environments, the ModelNet [19] emulation platform and the PlanetLab [14] distributed testbed. With ModelNet we evaluated large-scale Cone deployments of 1,000 hosts on an emulated network, and with PlanetLab we evaluated smaller-scale Cone deployments of 50 hosts widely distributed on the Internet. Using these testbeds, we evaluated the worst-case performance of the Cone implementation, its load balancing properties, and its performance and behavior under highly dynamic system configurations.

7.1 Testbeds

We used the ModelNet and PlanetLab testbeds to evaluate the Cone implementation. ModelNet consists of edge machines and emulator machines. The edge nodes are responsible for hosting the target applications (Cone and the query executables). The emulator nodes are responsible for imposing the hop-by-hop delay, bandwidth, loss rate, and congestion of an Internet topology. For our experiments, we used a 10,000 node INET [6] topology with both bandwidth and latency limits. We used 10 2Ghz Pentium-4's running Linux 2.4.20 as our edge machines and multiplexed 1,000 instances of Cone daemons across these 10 machines. This is feasible since Cone is neither bandwidth nor CPU intensive. We used two emulator machines (1.4Ghz Pentium III, running FreeBSD-4.7) to impose the specified network attributes, including loss rate and congestion, among the Cone daemons running on the edge machines.

We also ran our system on a 50-node PlanetLab population. The PlanetLab nodes chosen were primarily distributed across educational institutions in the Continental USA. Each PlanetLab host ran one instance of the Cone daemon.

In all of our experiments on both testbeds, each host held a single key value, and each child host probed its parent host every 1 second to maintain the consistency of the data structure.

7.2 Performance

In our first experiment we evaluate the worst-case performance of Cone on ModelNet and on PlanetLab. To perform this evaluation, we first start a Cone system with a fixed size. The keys assigned to the hosts followed an exponential distribution with mean 50,000. We gave the system time to stabilize and the data structure to become consistent. We then queried every host in the system for the maximum key value in a round-robin fashion, i.e., after the query to one node returned, we immediately queried the next node. The time to query the maximum is the worst-case query time, since every query will be routed to the root of the Cone tree. We performed 10 such iterations of querying every node in the system.

On ModelNet, we repeated the experiment for system sizes of 10, 50, 100 and 1000. On PlanetLab, we used systems of size 10, 20, 30, 40 and 50. Figure 9 shows the average query time for the maximum key for systems of different sizes on ModelNet and on PlanetLab. From the ModelNet results, we see that worst-case query time of Cone increases logarithmically with system size. It is difficult to make such observations with the PlanetLab curve since the population is small. However, we see a sizable difference in the performance of the system on the two frameworks. We believe that this is because ModelNet uses an INET topology with bandwidth and latency constraints, while the PlanetLab hosts are located in well-connected educational institutions many of which are connected to each other on Internet2.

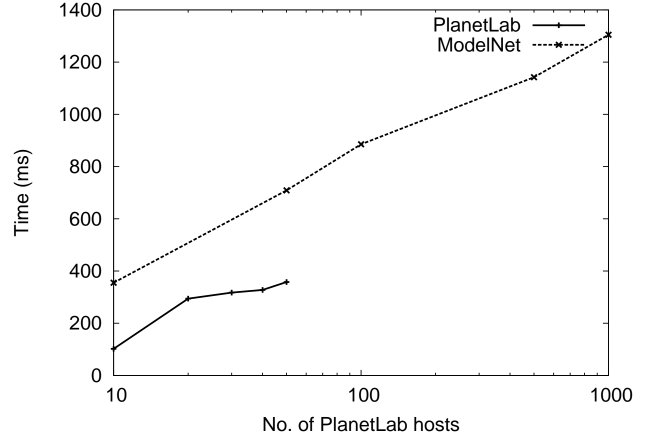

Figure 9: Worst-case performance of Cone:Comparison between ModelNet and PlanetLab.

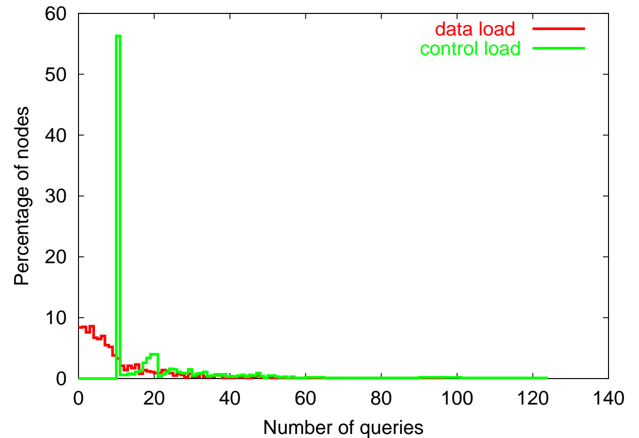

Figure 10: Distribution of the load on nodes using exponential key distribution and exponential query distribution with the same mean of 50,000.

7.3 Load balancing

We evaluated the load-balancing properties of Cone on a 1000-node ModelNet population. In this section, we describe our methodology and summarize our results.

The experimental setup was as follows. We emulated 1000 hosts on ModelNet, with each host running an instance of

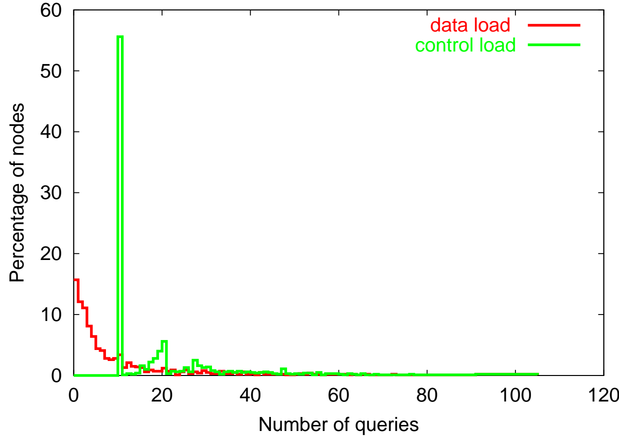

Figure 11: Distribution of the load on nodes using exponential key distribution and uniform random query distribution with the same mean of 50,000

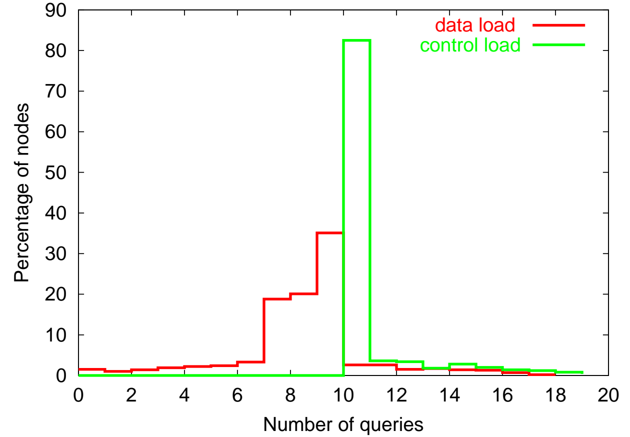

Figure 12: CDF of load on nodes using exponential key and query distributions, key values decrease by the query values.

Cone. The Cone keys were chosen from an exponential distribution with a mean value of 50000. Once the Cone data structure had stabilized, we queried each of the 1000 hosts in a round-robin fashion for a specific query value, say X. Hence the queries were of the form, "Find 1 resource with key value at least X". For each experiment, we performed 10 iterations of round-robin queries across all hosts for a total of 10,000 queries.

The goal of our first experiment was to confirm our theoretical analysis of the load balancing properties of Cone when key and query distribution are the same. Therefore in our first load-balancing experiment, the query values also followed an exponential distribution with a mean value of 50,000. Once the queries completed, we measured how many of the queries were satisfied by each host, thereby calculating the data load for each host. We also measured how many query messages went through each host, thereby calculating the control load on each host.

Figure 10 shows the distribution of the data load and control load for all 1000 hosts. We calculated that both the data load and control load values for 90% of all nodes are within one standard deviation of the mean, thereby confirming our hypothesis about a good load balance being achieved when both node keys and query keys follow the same distribution. Furthermore, we found that the maximum control load on any node is roughly 124, while the minimum is 10. This makes the control imbalance factor for the system 12.4. According to our analysis for control load in Section 5.2, the control load imbalance should be $log(N)$, which is our case is 10 (since there are 1000 nodes). The empirical value of control imbalance is only slightly more than the expected control imbalance factor, supporting our earlier analytical findings.

To see how using different key and query distributions affects load balance, we repeated the above experiment with the queries instead following a uniform random distribution with the same mean value of 50,000. The key distribution remained exponential with the same mean. Figure 11 plots the control load and data load distributions for the queries. In spite of differing key and query distributions, the load-balancing for this setup is good, with 90% of all nodes having a load within one standard deviation of the mean. The control load imbalance for this graph is less than 10.

We mentioned in Section 7.3 that applications can use an application-specific load-balancing technique to improve load-balancing. We now evaluate one such technique.

Let us consider a resource selection application, such as a distributed computing platform, in which the resource is CPU cycles and the key of each host represents the number of CPU cycles it is willing to provide. The application queries Cone to provide it with a host that has at least X CPU cycles to provide. Once Cone responds with the address of a suitable host, say H, the application runs a job on the host, effectively decreasing the available CPU cycles on H by X. Hence the key value of H drops, and H also drops down to lower levels in the Cone tree. This will have the affect of reducing both the control and data load on H. So, the more queries that the host H satisfies, the lower its key gets. This drops it further down the Cone tree, thereby reducing both its control and data load.

To investigate the effectiveness of this application behavior, we performed the query experiment using the same exponential key distribution with mean 50,000. In this experiment, however, we use an exponential query distribution with mean 1000. And each time a host satisfies a query, we reduce its key by the query value. All other parameters for the experiment remain the same as before.

Figure 12 shows the results of this experiment. In this case, we see that the maximum data load and control load (number of queries) on hosts are 18 and 19, respectively, which is much smaller than that shown in Figures 10 and 11. As a result, we conclude that application-specific load-balancing techniques can be successful in balancing both

data and control load on Cone.

7.4 System Dynamics

Our main goal in this section is to explore how well Cone operates when faced with a large amount of dynamism in the system, created not only by key value changes, but also by membership changes. We first show how Cone copes with frequent key changes while the membership remains constant. We then explore how Cone reacts with membership changes as well as key changes.

7.4.1 Changes in key value

In this section, we evaluate three key aspects of Cone: its reactiveness to rapid changes in key values, its stability to random and frequent changes in key values, and the accuracy of its query results.

To do this evaluation, we performed the following experiment. We ran Cone on 50 PlanetLab machines chosen to be geographically distributed across the continental USA. The experiment consisted of running two separate processes, `vary-ds` and `query-ds`. At a high level, `vary-ds` creates the Cone data structure and causes it to change frequently, while `query-ds` queries the Cone data structure to find the host with the maximum key value. Both processes run on a local (not PlanetLab) machine and they communicate with the Cone daemons on the PlanetLab hosts using the Cone API.

The `vary-ds` process starts the Cone daemon on all PlanetLab hosts and it initializes the key value for the hosts using the `set_key` operation. It sets keys of the hosts based on an exponential distribution with a mean value of 50,000. Once the Cone daemon is running on all 50 hosts, `vary-ds` iteratively changes the keys of hosts in the following way. In each iteration, it generates one `change_key` event, either on the host with the maximum key or on a randomly chosen host according to a random choice. For each `change_key`, `vary-ds` chooses a new key from the same exponential distribution with mean 50,000. By changing the key of the maximum host, `vary-ds` forces the maximum key value to drop. By changing the key of a randomly chosen host, `vary-ds` maintains the intended key value distribution in the system. In both cases, it introduces inconsistency into the data structure and forces Cone to adapt and re-arrange itself. Introducing change in the system every 6 seconds is an aggressive policy. Using the analogy of half-life of a peer-to-peer system as defined in [12], this implies that half the system is changing state in 25x6=150 seconds, or every two-and-a-half minutes. For applications such as a distributed computing platform, this translates to the application asking for resources and changes the resource values of half the hosts in the system every two-and-a-half minutes, thus making it a very dynamic system.

The `query-ds` process queries the Cone data structure to find the maximum key. It runs on the same local machine as `vary-ds` and uses the Cone API on the PlanetLab host to make the query. It uses a round-robin strategy

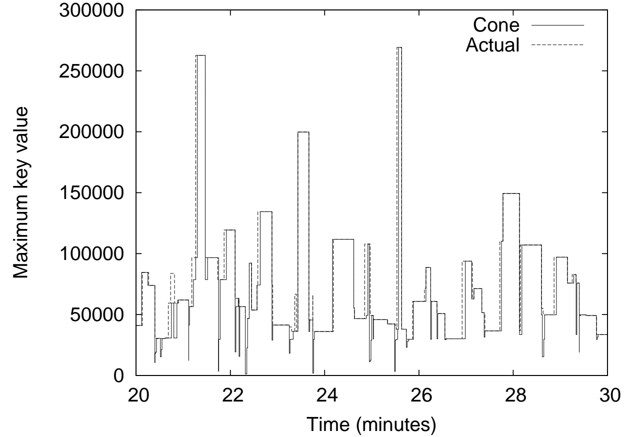

Figure 13: The reactiveness of Cone to key changes in the PlanetLab experiment.

to make queries, i.e., after the query to one host completes, `query-ds` starts querying the next host without delay, and so on. One cycle of queries takes approximately 18 seconds to complete.

We keep the two processes `vary-ds` and `query-ds` independent and unsynchronized since we wish to model how applications will interact with Cone: some applications will be changing the key values while others will query the data structure.

Both processes run continuously for approximately 50 minutes. Figure 13 shows the results of this experiment from minute 20 through minute 30 of the experiment. The x-axis shows time duration in minutes, while the y-axis shows the maximum key values. The dotted line shows the true maximum key value, while the bold line shows Cone's replies to queries for the maximum key.

In terms of reactiveness, the figure shows that there is a brief lag between when the maximum value changes and when the Cone queries begin to return the new maximum value. On the average, this lag is 2.2 seconds. From the performance results in Figure 9, we see that query time for a 50-host PlanetLab system is approximately 360 ms. Thus, on an average, 180 ms out of the 2.2 second-lag is purely query time added on after the system has reached a consistent state. So, it roughly takes 2 seconds for the system to stabilize. Considering that the parent query interval is 1 second, the system takes an average of only 2 probe periods to regain consistency after a change in the maximum value. Of course, for larger systems this probe period will be higher, but the increase would be logarithmic in the size of the system given the structure of Cone.

The absolute time that Cone takes to reach a consistent state can be further reduced by increasing the frequency at which hosts probe their parents. Hence there is a trade-off between the reactiveness of the system and the bandwidth used to maintain the consistency of the data structure. We calculated that to maintain a 50-host Cone, the system re-

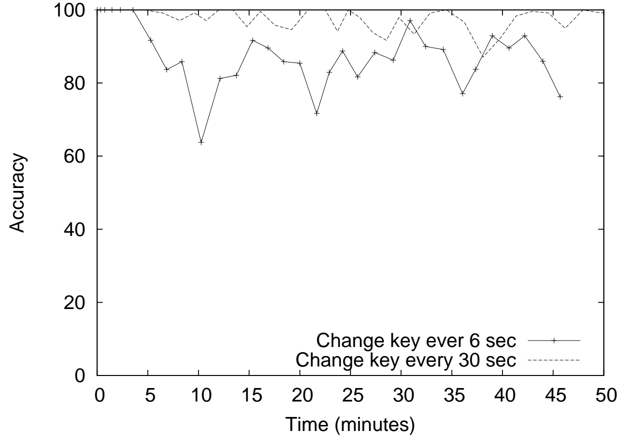

Figure 14: The accuracy of Cone in the PlanetLab experiment, with key values changing frequently.

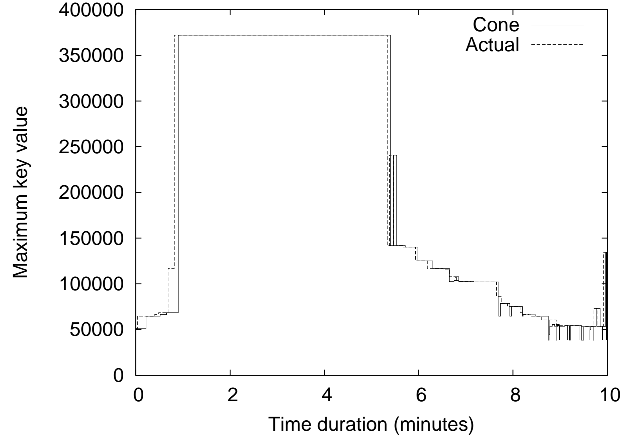

Figure 15: Key changes in PlanetLab.

quires roughly 760 bps per host on average. This includes not only the traffic that Cone generates, but also the traffic that Chord generates for DHT maintenance. We intend to study how this bandwidth usage changes with different probing frequencies in Cone.

Figure 13 also shows that the Cone data structure remains stable in spite of random key changes and frequent changes to the maximum key value. It responds correctly to queries albeit with a lag of 2.2 seconds on the average. On some occasions, the query results dip considerably when the the maximum key changes. This happens to the queries that `query-ds` issues before the child of the host with the previous maximum key probes its parent and discovers that its parent does not hold the maximum key any more. In this situation, it routes all queries to its parent, even though the parent no longer holds the maximum key value. This situation can be remedied if the child, before responding to the query, checked to see if its parent still holds the maximum key. We intend to add this optimization to the system in the near future.

Finally, we evaluated the accuracy of Cone. We define the accuracy of a set of Cone queries as the percentage of these queries that were successful in retrieving the true maximum key. First, we repeated the above experiment by changing keys every 30 seconds instead of every 6 seconds. All other parameters remained the same as the previous run. We then measured the accuracy of Cone for the two experiments, aggregating every 250 successive queries. In our experiments, 250 queries were made approximately every 100 seconds.

In Figure 14, we plot the level of accuracy for the two experiments versus time. The accuracy for the first run (key changes every 6 seconds) is not very good. This is not surprising since, as we calculated, there is an average lag of 2.2 seconds for every change in maximum value, and that the maximum value changes at least once every 12 seconds on average. The ratio of time that Cone returns an incorrect maximum value is therefore approximately 2.2/12, which is 0.18. Hence the expected accuracy is 82%. The calculated average accuracy for this curve over the entire experiment is 87.9%.

For the second run of the experiment, the accuracy is better. This is because the lag of 2.2 seconds forms a considerably smaller fraction of the time duration of 30 seconds between changes of the maximum key. This fraction is 0.07, which makes the expected value or accuracy 93%. The calculated average accuracy for this run is 97.4%. Using the same calculation, we determine that to achieve 99% accuracy, the time duration between maximum key changes should be at least 2.2/0.01 = 220 seconds. However, note that since increasing the probe frequency decreases the lag, it will also increase the level of accuracy. We intend to explore how this probe frequency affects the lag, and as a consequence, the accuracy of Cone.

7.4.2 Changes in membership

To measure how Cone's reactiveness is affected by membership changes, we next performed an experiment in which the hosts as well as the maximum key in the system changed periodically. The experimental setup was as follows. We used a population of 50 PlanetLab hosts. They use an exponential key distribution with a mean of 50,000. To perform this experiment, we had three controlling processes: `join-ds`, `leave-ds`, and `query-ds`.

The `join-ds` process causes hosts to join the Cone system. It uses a Poisson process with a mean inter-arrival time of 12 seconds to control the `join` events. The `leave-ds` process causes the host with the maximum key to leave the Cone system. It also uses a Poisson process with the same mean of 12 seconds to model inter-departure times. The `leave-ds` process always makes the node with the maximum key leave so that it induces a considerable amount of churn in the system within the duration for which this experiment was run. As in the previous experiment, we also had a `query-ds` process that continuously queried all nodes in the system in a round-robin fashion.

At the start of the experiment, only `join-ds` operates until 40 of the PlanetLab hosts have joined the system. At this point, both the `join-ds` and the `leave-ds` processes operate, thus keeping the steady state size of the system at 40. The half-life of the system in the course of this experiment was therefore 6*20 = 120 seconds, or 2 minutes. We use such a small half-life so that we can evaluate the Cone system under a significant amount of stress. Note that `query-ds` also starts running right at the start of the experiment.

We ran this experiment for 10 minutes. Figure 15 shows the results of this run. The x-axis shows the time duration of the experiment, while the y-axis shows the maximum key values in the system. We plot one curve for the actual maximum key value of the system and one for the query responses that Cone provides. We first see an increase in the maximum value as more and more hosts join the system. We see a sudden increase after the 1st minute of the run because a host joined the system with a key of roughly 370,000 (mean value is 50,000). All subsequently joining nodes had a smaller key value, and hence this host stayed as the root of the Cone structure until the `leave-ds` process started.

Once the `leave-ds` process started, the maximum key value in the system fell. As in the previous experiment with key changes, we notice that there is a lag between when the key changed and when Cone responded to the key change with correct query results. On average in this experiment, this lag is 6.6 seconds. This is more significant than in the previous case since host joins and leaves create a lot of reordering not only in Cone, but also in the Chord DHT. While for this experiment, a lag of 6.6 seconds causes the accuracy of the system to be low, namely 45%. However this is because the half-life of the experimental system was extremely aggressive, i.e., 2 minutes. In a recent study [4], the findings indicated that the half-life of a highly dynamic wide-area file-sharing system was approximately 4 hours. Using this more realistic value of half-life, we measure the accuracy to be 99%.

8 Extensions to Cone

In this paper, we have concentrated on designing and evaluating the Cone data structure for the Max operator with one key value per host, In reality, the Cone structure is much more generic. In this section, we describe some extensions that can be applied to Cone to broaden the range of applications it can be used with.

Using Cone for multiple data items per host: While cone uses only one key value per host, one could extend it to support multiple data items per host. Each host in Cone could maintain a list of data items which are sorted based on their key values. A host advertises its key as the maximum of all keys that it stores. The advertised key value of a host can change when new keys are inserted in the system, or old keys are removed from it. Hence key inserts and deletes translate to the `change_key` operation describe earlier. However, to prevent data item loss when a Cone host leaves the system, the keys have to be replicated on a number of other hosts. This is a generic shortcoming of all distributed indexing schemes.

Using multiple attributes in a single Cone: Cone supports ordering based on one key value. However, this key could be constructed in many ways. For example, if we want to represent two attribute values in a single key, we can use several database techniques such as bit-interleaving. This approach does have some problems since bit-interleaving reduces the accuracy of the ordering. This can be mitigated by performing multiple queries on Cone and selecting the result which best matches our query.

Using Cone for union operations: In this paper, our concentration has been on the Max (or greater than) operator. The descriptions in this paper hold for any operation that describes a total order. However, with a few changes, Cone can also be used for operations such as union. To do this, a parent node holds the union of its child's values and its own values. When one searches for a particular value in the system, the request trickles up the Cone tree until it finds a node that holds the value in its union set. The request then trickles down one path in the Cone tree until it finds the node that first advertised the requested value. Hence the maximum number of hops in this case would be $2log(N)$.

9 Conclusion and Future Work

In essence, Cone builds a tree over random IDs assigned to nodes. The standard argument against trees (compared to richer hypercube-like interconnections such as Chord and Pastry) is that of load balance and fault tolerance. For a tree, it may appear that all requests must pass through the root, leading to a $O(N)$ load imbalance. However, the combination of random assignment of resources to tree leaves (regardless of resource values, because of random ID assignment) and short-cutting at sub-trees is surprisingly powerful. Our analysis and evaluation confirm this. The Cone structure remains stable in the face of rapid key value changes and rapid membership changes.

A fundamental trade-off we made while designing Cone was to give up some functionality for simplicity and performance. Cone in its current form cannot efficiently support enumerative operations such as "Find *all* objects with value V". However, for our targeted applications, we believe this nature of query to be of secondary importance. However, we intend to develop algorithms to efficiently support closed-range queries such as "Find an object that has value $>$ V and $<$ W". We also intend to explore the possibilities of supporting multiple attributes using well-known database techniques such as bit-interleaving. Another direction of exploration is building Cone without a DHT. Since Cone is a heap as well as a trie , it can also be used for routing based on identifiers [8].

In the near future, we plan to study how the reactiveness of Cone can be improved by changing probe values and by piggybacking key value probes on query traffic. We believe that these changes will significantly improve the lag we saw in Section 7.4. We also wish to implement caching in the Cone system. This will not only reduce query time, but it will also help to load-balance the control traffic in Cone. Apart from these, we also intend to implement recursive lookups in Cone. We also wish to study in more detail the Cone extensions mentioned in Section 8. Completely working out such extensions will take us closer to realize the promise of the general approach of augmentation.

References

[1] A. Andrzejak and Z. Xu. Scalable, effi cient range queries for grid information services. In *Proceedings of P2P2002*.

[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 *Proceedings of Pervasive2002*, 2002.

[4] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *Proceedings of the second international workshop on peer-to-peer systems (IPTPS)*, 2003.

[5] Berkeley open infrastructure for network computing (boinc). http://boinc.ssl.berkeley.edu/.

[6] H. Chang et al. Towards capturing representative as-level internet topologies. In *Proc. of ACM SIGMETRICS*, 2002.

[7] Everquest website. http://everquest.station.sony.com.

[8] M. J. Freedman and R. Vingralek. Effi cient peer-to-peer lookup based on a distributed trie. In *Proceedings of IPTPS*, 2002.

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

[10] R. Huebsch et al. Querying the internet with PIER. In *Twenty-ninth International Conference on Very Large Data Bases*, Berlin, Germany, September 2003.

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

[12] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of PODC*, 2002.

[13] D. Mazieres. A toolkit for user-level fi le systems. In *Proc. of the Usenix technical conference*, 2001.

[14] The PlanetLab website. http://www.planet-lab.org.

[15] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *Proceedings of IPTPS 2003*.

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

[17] D. Spence and T. Harris. XenoSearch: Distributed resource discovery in the XenoServer open platform. In *Proceedings of HPDC 2003*, Seattle, WA.

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

[19] A. Vahdat et al. Scalability and accuracy in a large-scale network emulator. In *Proc. of OSDI*, 2002.

[20] R. van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed systems monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(3), 2003.

[21] P. Yalagandula and M. Dahlin. A scalable distributed information management system. Technical Report Tr-03-47, University of Texas, Austin.

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