Load Balancing and Locality in Range-Queriable Data Structures

James Aspnes [*†] Jonathan Kirsch[*‡] Arvind Krishnamurthy[*§]

ABSTRACT

We describe a load-balancing mechanism for assigning elements to servers in a distributed data structure that supports range queries. The mechanism ensures both load-balancing with respect to an arbitrary load measure specified by the user and geographical locality, assigning elements with similar keys to the same server. Though our mechanism is specifically designed to improve the performance of skip graphs, it can be adapted to provide deterministic, locality-preserving load-balancing to any distributed data structure that orders machines in a ring or line.

Categories and Subject Descriptors

C.2.2 [**Computer Communication Networks**]: Network Protocols—*Applications, routing protocols*

General Terms

Algorithms, Performance, Experimentation

Keywords

Peer-to-peer systems, Overlay networks

1. INTRODUCTION

A peer-to-peer data storage system can be viewed as a very large distributed data structure where pointers cross machine boundaries. The design of the data structure itself may or may not constrain which machine stores each element; current systems in the literature include both *distributed hash tables* (DHTs), in which assignment of elements (or, applying an additional level of indirection, pointers to elements) to machines is tightly controlled by the hash function, practical systems like *SkipNets* [9], in which elements do not cross organizational boundaries but in principle may otherwise be assigned arbitrarily within the servers owned by an organization, and more theoretical data structures like *skip graphs* [2], where the data structure links elements directly, with no requirements on where the elements are placed. Unlike DHTs, these latter systems provide additional capabilities, like support for range queries, and provide additional flexibility in choosing how to place elements. However, this additional flexibility comes at a price: because the system must track each element individually, the number of pointers in the data structure—each of which requires periodic network traffic to maintain—will be much larger than in systems that can group elements together. (We describe current systems in more detail in Section 2.)

The present work addresses this problem by designing a distributed load-balancing policy for allocating elements of a skip graph in which each machine controls some interval in the keyspace, with the property that elements with nearby keys are stored on the same machine. This allows the full skip graph to be replaced by a truncated data structure that contains only a constant number of sample elements from each machine, thus reducing the number of intermachine pointers from $O(n \log n)$, where n is the number of elements or keys, to $O(m \log m)$, where m is the (presumably much smaller) number of machines in the system. This gives a number of intermachine pointers comparable to that of systems like Chord [22], Pastry [21], or Tapestry [25], while providing better load-balancing and retaining the skip graph's ability to perform range queries.

The difficulty in supporting range queries while providing load balancing is that we must preserve the order of elements within the data structure. As a result, executions in which many elements with similar keys are inserted will tend to produce high load on the machine that handles the interval containing these keys. Any system that reduces this load must do so by dynamically recruiting new machines to carry part of this load by splitting off part of the interval. Our mechanism does so dynamically by adopting a pairing strategy in which heavily-loaded machines are placed next to lightly-loaded machines in the data structure, so that insertions on heavily-loaded machines can be dealt with by migrating elements to their lightly-loaded neighbors. We also maintain a small population of empty machines that can be moved freely within the data structure to add new lightly-loaded machines to particular regions as the machines in those regions fill up.

The pairing strategy ensures that a typical insertion requires moving at most one element to a new node to preserve ordering. Recruitment of new empty partners occurs only when a previously-recruited node fills, and so the amortized cost per insertion of the recruitment operation is inversely proportional to the capacity of a node. Finally, the skip graph is thinned by a simple sampling

[*] Yale University Department of Computer Science.

[†] Email: aspnes@cs.yale.edu. Supported by NSF grants CCR-0098078 and CNS-0305258.

[‡] Email: jonathan.kirsch@yale.edu.

[§] Email: arvind@cs.yale.edu. Supported by NSF grants CCR-9985304, ANI-0207399, and CCR-0209122.

PODC'04, July 25–28, 2004, St. Johns, Newfoundland, Canada.
Copyright 2004 ACM 1-58113-802-4/04/0007 ...$5.00.

mechanism, in which each node keeps one of its central elements in the skip graph, that ensures that the skip graph only needs to be updated when this sample element drifts to an adjacent node, an event that occurs in the worst case after a number of insertions proportional to the node capacity. It follows that most insertions (and any deletions) can be performed with no modification to the skip graph and at most one element move.

In the simplest version of the algorithm (described in Section 3), the basics of the insertion and migration of elements and the creation and recruitment of empty machines is handled locally, while a centralized mechanism is used to tune certain global parameters (such as the threshold that distinguishes lightly-loaded from heavily-loaded machines). Aside from requiring a central controller, which creates a single point of failure in the system, this basic algorithm also suffers from periods of sudden high network traffic as these global parameters suddenly change and all nodes simultaneously start migrating elements to achieve updated load-balancing goals. In Section 4, we describe a distributed mechanism that avoids these problems by replacing the global controller with a distributed mechanism based on statistical sampling and by staggering the local adjustments made by individual nodes to response to changes in global system load to avoid massive migratory stampedes.

Our results are not purely theoretical; in Section 5, we provide both simulation results and experimental results on a real implementation that show that our mechanisms provide excellent load-balancing and search performance in practice.

Our mechanism does not depend specifically on properties of skip graphs, and can be applied to any system that uses an ordered allocation of elements to machines organized in a line or ring. It also allows for arbitrary measures of the additional load created by a single element, so that it can be used, for example, to simultaneously balance space and network traffic. We believe that such a load-balancing mechanism may be useful in many existing systems that currently rely on the weaker averaging effects of probabilistic placement.

2. RELATED WORK

In this section, we start by describing how current systems achieve the goals of load-balancing and (for some systems) supporting range queries. We conclude by describing work on the problem, closely related to ours, of maintaining sorted lists when it is expensive to move individual elements.

2.1 Distributed Hash Tables

Distributed hash tables (DHTs) [19, 21, 22, 25] view the overlay as a distributed data structure that dictates both network topology and message routing. DHTs use hashing schemes, such as *consistent hashing* [12], to map machines and keys to a single, modular ID space. This results in a setting where the objects are (probabilistically) uniformly distributed over the ID space. These approaches have been shown to be massively scalable, requiring $O(\log m)$ neighbor information and guaranteeing $O(\log m)$ diameter for arbitrarily-sized networks.

DHTs effectively solve the load-balancing problem probabilistically; the difficulty is in supporting range queries. The problem is that the hash function destroys the logical integrity of the keyspace, making it difficult to efficiently support complex similarity searches and range queries [8, 14].

Ratnasamy *et. al.* [20] outline an approach to overcome this limitation of DHTs by organizing the keys into a distributed trie, with each node of the trie stored as an object in the DHT. Given a query, the system attempts to identify the longest prefix of the query that appears as a trie-node. Given the data domain D, this operation can be performed using $O(\log \log |D|)$ DHT lookups, where each DHT lookup typically incurring a $O(\log m)$ cost. The system is load-balanced, since the trie nodes are (probabilistically) uniformly distributed across the machines. The system does suffer from hotspots, since the top-level trie-nodes are likely to be more frequently accessed than the lower-level trie-nodes.

In addition to such work on permitting range queries, some recent work has concentrated on improving the load-balancing provided by random assignments in DHTs. In Chord, for example, each machine takes responsibility for all points in a circular keyspace that are closest to its hashed identity. If the machines are placed randomly in the circle (a reasonable assumption given a strong enough hash function), the ratio between the largest and smallest regions belonging to individual machines can be large, and it is likely that some particular machine will be responsible for an $\Omega(\log n/n \log \log n)$ fraction of the keyspace. Karger and Ruhl [13], have recently shown how to avoid this problem by allowing nodes to choose between $O(\log n)$ random locations according to a clever rule; with high probability this ensures that no node owns more than $O(1/n)$ of the keyspace.

Karger and Ruhl describe a second load-balancing algorithm that is similar in many respects to the one described in the present work. Their second algorithm is aimed at balancing load between machines when the distribution of items in the keyspace is unbalanced, and is based on a probabilistic work-stealing mechanism in which underloaded nodes periodically move themselves next to overloaded nodes found by sampling; it depends on nodes being able to move to arbitrary positions in the keyspace and is not compatible with their first algorithm. This yields a "push" load-balancing algorithm, where underloaded nodes push themselves into high-traffic parts of the keyspace; in contrast, we describe a "pull" algorithm where overloaded nodes pull waiting empty nodes from an explicit free list. The "pull" approach has the advantage of ensuring that no node ever becomes so overloaded that it must turn away inserts; instead, insertion of an item always succeeds as long as the free list is not empty and requires moving at most a constant number of items between nodes. The "push" approach has the advantage of simplicity, a more uniform placement of elements on nodes, and the avoidance of a separate mechanism to adjust the advertised capacities of nodes. It is an interesting question whether a combination of the two approaches might achieve the best properties of both.

2.2 Searchable Concurrent Data Structures

Distributed implementations of data structures such as *skip lists* and *skip graphs* can be used to support range queries. These data structures offer a randomized alternative to the more complex balanced-tree data structures, such as red-black trees or b-trees. They provide a probabilistic guarantee that the standard dictionary operations can be performed in $O(\log n)$ time, where *n* is the number of keys currently in the system. Skip lists are simply collections of linked-lists, and are organized as follows. All keys in the system appear in sorted order in the bottom-most list, which is referred to as Level 0. Each key that appears in the list at Level i, would also appear in the list at Level $i + 1$ with some probability p. At each level, a key stores pointers to its left and right neighbors (in the case of a doubly-linked skip list). To locate a key, one searches the highest level (which might have just a few keys), dropping down to the more densely-populated lower levels if needed. There are, on average, $O(\log n)$ levels in the system, meaning that a search will traverse $O(\log n)$ keys until it reaches its destination [15, 16, 18].

Skip lists are not directly suitable for use in a distributed environment for several reasons. First, since all operations begin in the highest level of the skip list, which is sparse, these top-level keys become hot-spots, and will be involved in an operation with high probability, potentially overwhelming the machines who own them. Furthermore, the sparsity of the top-level list creates single points of failure: if the machines owning these keys go down, the system will be partitioned. These issues are addressed by the skip graph.

The skip graph extends the skip list into a distributed environment by adding redundant connectivity and multiple handles into the data structure. It is equivalent to a collection of up to n skip lists that happen to share some of their lower levels [2]. More formally, all keys appear in sorted order in the list at Level 0. Each Level i, for $i > 0$, can now contain multiple linked-lists. Each key maintains a *membership vector*, which is a random string of bits. A list at Level i contains all keys that have the same i-length prefix for their membership vectors (as illustrated by the top portion of Figure 1). This continues until the key becomes a singleton, which will result in, on average, $O(\log n)$ levels in the skip graph. For a complete description of the data structure, please see [2].

The search, insertion, and deletion algorithms for a skip graph are essentially the same as for a skip list, with slight modifications to generalize them into a distributed environment. Every key becomes a handle into the data structure, making the skip graph both highly concurrent and resistant to node failures. More importantly, that the skip graph does not employ a hashing function allows it to support range queries, since logically similar keys will become neighbors in the skip graph. Despite these attractive features, there are still several barriers to the use of the skip graph as is, which we describe below.

Each key must store pointers to an average of two neighbors for each of the $O(\log n)$ levels. The result is a cost of $O(\log n)$ state *per key*, considerably more than in a distributed hash table, which requires $O(\log m)$ state per machine. Another limitation of the skip graph as it was proposed in [2] is that it does not describe the method by which keys are assigned to machines in the system. Therefore, the skip graph makes no guarantees about system-wide load-balancing nor does it make any guarantees about the geographic locality of neighboring keys.

2.3 Hybrid Strategies

Since we are able to achieve each of the desired goals using either the skip graph or the distributed hash table, it is logical to attempt to combine the desirable properties of the two into a single system. This is essentially what was proposed by Awerbuch and Scheideler in [3]. More formally, their scheme incorporates two orthogonal, concurrent data structures. One data structure, *F*, is used to maintain the keys, or files, in the system, and must support the range query operation. Another, *S*, is used to organize the sites in the system, and only needs to support the look-up operation. Together, these concurrent data structures interact through a minimal interface. Awerbuch and Scheideler suggest using the skip graph as *F*, and a distributed hash table, such as Chord, as *S*. Intuitively, this scheme simply uses the skip graph to perform the search operation, and then hashes the key to a machine using the Chord protocol. Note that the system can still support efficient range queries, because there is no need to repeatedly search through the skip graph after the initial search; one can simply follow the pointers along Level 0 of the skip graph. Furthermore, the system achieves the theoretical load-balancing property inherent to the use of the consistent hashing mechanism.

The real problem with this approach, however, is that the hash function destroys any notion of geographic locality for the keys. With keys assigned to random machines around the system, two keys are still likely to be geographically far apart, hurting performance. It also still suffers from the fact that the skip graph maintains more state than is ideal. Further, the traversal of any pointer in the skip graph requires a lookup operation in *S*, thus increasing the overall cost by a factor of $O(\log m)$.

One possibility might be to combine either load-balancing scheme described here or that of Karger and Ruhl with the hybrid mechanism of Awerbuch and Scheideler, to obtain both range queries and strong load-balancing. But the increased cost of searches relative to simple skip graphs might be prohibitive.

2.4 Concurrent B-trees

If most pointer dereferences are to be local, it seems clear that most logically-related keys should be located in close geographic proximity. One way to achieve this goal is to assign logically similar keys to the same machine. This method is used in the *dE-tree*, a data structure based on the distributed b-tree. In [11], Johnson and Colbrook define an *extent* to be a maximal length sequence of neighboring leaves that are owned by the same machine. Each machine then owns some portion of the extents in the system. Intuitively, the leaves of the system are grouped together, and each group is owned by a single machine.

We are, however, left with a scenario in which we can obtain good geographic locality, but suffer from potential data skew and load-imbalance. To remedy this, Johnson and Colbrook suggest that local changes can be initially attempted. For example, a heavily-loaded machine can try to dump some of its keys into the extent of a lightly-loaded, neighboring machine. If all machines are heavily-loaded, a new extent is created for the best candidate machine. This might mean the machine with the least data load, or the machine who would provide the best communication locality.

While the dE-tree sounds promising, there are several significant drawbacks that limit its applicability to efficient, peer-to-peer systems. The first problem is that the dE-tree requires a significant amount of data replication to reduce its message complexity. Each machine maintains a relatively large portion of the tree, with the motivation being that the most expensive b-tree operations, such as node-merges and node-splits, can be done on a (mostly) local basis. Such replication requires a sophisticated (and costly) cache-coherency scheme. The more important issue, however, is that no formal method is given for achieving system-wide load-balancing. Johnson and Colbrook describe the need to propagate load-balancing information throughout the system quickly, in order to keep the heuristic regarding the election of a candidate machine reasonably efficient and up-to-date. This is undesirable because it means that changes cannot truly be local, and therefore the need to propagate information will increase the message complexity of the system. Clearly we would like a more formal method which provides provable guarantees of load-balancing.

2.5 Ordered-Array Data Structures

Our problem is similar in some respects to the problem of maintaining an ordered array with gaps, where elements are inserted and deleted dynamically and the goal is to minimize the amortized number of element moves. This problem, known as the *on-line monotonic list labeling problem* or the *file maintenance problem*, has been extensively studied [1, 4, 6, 7, 10, 24]. Typical solutions (e.g. [4,24]) implicitly treat an array of n elements as an $O(\log n)$-level binary tree, and perform rebalancing operations on this tree when particular subtrees become too heavy or too light. The cost of rebalancing adds $O(\log^2 n)$ work in the worst case to each insert or delete. Our load-balancing mechanism, which maintains a sprinkling of light nodes throughout the data structure similar to the gaps in ordered arrays, is inspired in part by this work.

Unfortunately, a direct application of the ordered-array approach suffers from the same need to propagate load information through the system suffered by the dE-tree. Our suspicion is that attempts to avoid such problem would be more trouble than the results would be worth in our particular setting, as the ability to move nodes allows for simpler solutions that are not possible in the ordered-array setting. However, if node order is fixed (for example, because it reflects geographical placement of the nodes), then solutions based on ordered arrays may be necessary.

2.6 Game-Theoretic Issues for Load-Balancing Mechanisms

One of the goals of our experiments has been to measure the effectiveness of heuristic load-balancing strategies based on local information. Such uncoordinated strategies have been studied from a game-theoretic perspective by Suri, Tóth, and Zhou [23], where clients placing resources on servers are assumed to engage in selfish, strategic behavior. Their results are not directly applicable to systems that have to preserve ordering, but their work suggests interesting possibilities for further analysis of our algorithms and others from a game-theoretic perspective (which we defer to future work).

3. BASIC ALGORITHM

We assign similar keys to the same machine, so that logically-related keys are located in close geographic proximity and most pointer dereferences are local. Specifically, we group keys into *buckets*, with each machine owning some number of buckets and taking responsibility for all items whose keys fall into these buckets. We do not place any requirements on keys except that they can be ordered; applications may assign keys to items in whatever way is most useful for them.

Each bucket elects a representative key to appear in the skip graph. These representative keys are used during search operations to navigate to the appropriate bucket. Notice that this scheme reduces the space complexity of a skip graph to $O(b \log b)$ pointers, where b is the number of buckets in the system, by limiting the number of keys inserted into the skip graph to about one per bucket. This mechanism requires only minimal modification to the skip graph algorithms presented in [2], but significantly reduces the amount of state required by the data structure. Representatives are generally chosen from the central items of a bucket; this allows items on the end to be migrated to adjacent buckets without having to update the skip graph unless the representative itself is migrated and must be replaced.

It is worth noting that the bucketing scheme described above also has the useful property of dividing the system into two (almost) orthogonal data structures. We can therefore think of the system as being composed of two layers. We refer to the combination of the skip graph and the bucketing layer as the *two-layer system*.

The top layer consists of the skip graph, where each key now stores, in addition to its neighbor pointers, a pointer to the bucket in which it is located. This acts as an overlay network supporting both searches and the maintenance of a free list. The lower layer consists of the chain of buckets, distributed among the machines in the system. This division is useful because it allows us to make optimizations to the skip graph indirectly, by manipulating the bucket layer and the interface between layers, without losing the desirable properties of the top-layer skip graph.

While we can obtain good geographic locality with this scheme, it suffers from potential data skew and load-imbalance as keys are added to and deleted from the system. We now describe our load-balancing mechanism which addresses these issues within the context of the two-layer system described above.

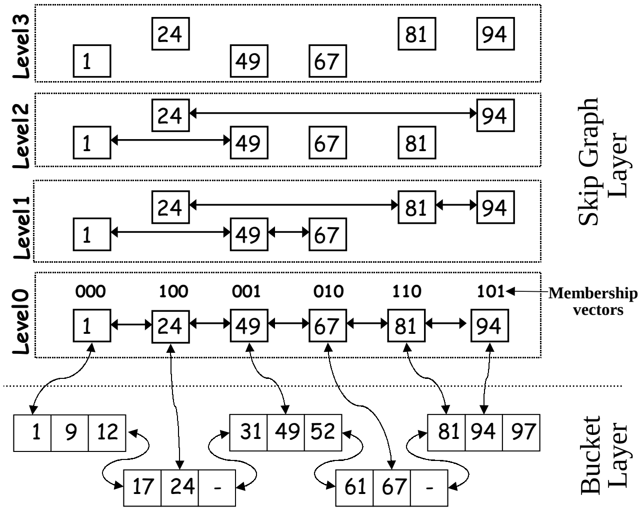

Figure 1: A sample layout illustrating the two-layered approach.

3.1 Searching

The search algorithm for the two-layer structure is essentially identical to a standard skip graph search for the skip graph element closest to the target, plus an additional search within the bucket that contains this element and the bucket's two immediate neighbors. The cost of a search is thus logarithmic in the number of skip graph elements, which will be approximately equal to the number of buckets. Range query operations proceed similarly.

THEOREM 1. *In a two-layer structure with b buckets, the operations of SEARCH, NEAREST-MATCH, NEAREST-PREDECESSOR, and NEAREST-SUCCESSOR require $O(\log b)$ time and $O(\log b)$ messages. Enumerating all elements between two targets takes $O(k + \log b)$ time, where k is the number of elements found.*

3.2 Load-Balancing

Our algorithm uses local changes to remedy load-imbalance. The basic idea is that a heavily-loaded machine can try to dump some of its keys into the bucket of a lightly-loaded neighboring machine. We maintain a "free list" of buckets, so that if all buckets are heavily-loaded, a new empty bucket is enlisted to bear some of the load, thereby classifying each bucket as *active* or *free*. We can use a separate skip graph to implement a fault-tolerant and efficient free-list. We simply associate each free bucket with a random key value, insert a pointer to the bucket into the skip graph, and satisfy requests for free buckets by returning any one of the elements stored in the skip graph. This strategy provides fault-tolerant location of free buckets at $O(\log b)$ cost from any one of the multiple entry points of the skip graph.

We further classify each bucket to be either *open* or *closed*. There are a number of ways we can make this distinction. For example, a closed bucket might have some threshold number of keys, or might be using up some threshold percentage of its network bandwidth. For the purposes of this discussion, we consider the former to be our criterion.

We next partition the list of active buckets into groups of two or three, maintaining the invariant that every closed bucket is adjacent to an open bucket and that every open bucket has a closed bucket to its left. Note that this invariant requires that there are at least two buckets in the system; if there is only a single machine in the initial state, we assume that it provides both buckets. Each group must have one of the following two patterns, with *C* representing a closed bucket, and *O* representing an open bucket:

1. C-O
2. C-O-C

We maintain this structure by transferring keys from neighboring buckets as needed. Thus, if a key is to be inserted into a closed bucket, one key from the closed bucket is transferred to the adjacent open bucket. Similarly, the deletion of a key from a closed bucket involves transferring a key from an open bucket. As an open bucket takes on more keys, it can declare itself closed, requiring a regrouping of the bucket structure. The details of such regroupings are described below.

3.2.1 Insertions

Insertions on closed buckets which do not cause the adjacent open bucket to become closed are straightforward, and involve the key transfers described above. There are two interesting patterns to consider for insertions involving a regrouping of the buckets. In each of the following cases, *O'* represents a fresh bucket from the free list, and a key is being inserted into the closed bucket *C1*:

1. C1-O2 ⇒ C1-O'-C2
2. C1-O2-C3 ⇒ C1-O2 | C3-O'

In Case 1, the new key is inserted into *C1*, causing the transfer of a key from *C1* to *O2*. This causes *O2* to become closed. The new bucket O' is taken from the free list to restore the structure. In Case 2, the new key is inserted into *C1*, causing the transfer of a key from *C1* to *O2*. In turn, *O2* transfers a key to *C3*, in order to stay open. *C3* must then transfer a key to the empty bucket, *O'*.

3.2.2 Deletions

There are three interesting patterns to consider for deletions involving a restructuring of the buckets. Note that a bucket is only placed back on the free list when a restructuring occurs; it is perfectly valid to have an open bucket, with no keys, as part of a group. In each case below, *O** represents the empty bucket which will be returned to the free list:

1. C1-O*-C3 ⇒ C1-O3
2. C1-O2 | C3-O* ⇒ C1-O2-C3
3. C1-O2-C3 | C4-O* ⇒ C1-O2 | C3-O4

In Case 1, *C3* transfers one of its keys to *C1*, and they form a 2-group. In Case 2, after the key is deleted from *C3*, *O2* transfers a key to *C3*, resulting in a single 3-group. If *O2* is also empty, then *C1* can pair up with *C3*, which is now open, to form a single 2-group. In Case 3, similar shifting can occur to form two 2-groups.

3.2.3 Analysis

We now analyze the scheme proposed above, highlighting both its attractive features and its limitations. The first thing to notice about this scheme is that it only requires highly-localized changes. Operations which do not require a restructuring of the groups involve at most two buckets. Since one machine owns the entire bucket, at most two machines will need to communicate for the given operation. Similarly, operations which require a restructuring of the groups involve at most three buckets, and one move of a bucket to or from the free list. Furthermore, since at least half of the active buckets are closed, this mechanism ensures that, when the free list is empty, the system is within a factor of two of the maximum load obtained under perfect load-balancing conditions.

Formally, we have:

LEMMA 2. *Between 1/3 and 1/2 of all buckets are open.*

PROOF. Immediate from the decomposition of the buckets into $C-O$ and $C-O-C$ groups. □

COROLLARY 3. *Let M be the capacity of a closed bucket, and let $M^* = n/b$ be the average load on any bucket if all n items are evenly distributed. Let ϵb be an upper bound on the number of buckets on the free list. Then the ratio M/M^* between the maximum load and the ideal average load bounded by*

$$M/M^* \leq \frac{2}{1-\epsilon}.$$

PROOF. There are at least $(1-\epsilon)b$ buckets not on the free list, of which at least $\frac{1}{2}(1-\epsilon)b$ are closed by Lemma 2. These closed buckets between them contain $\frac{1}{2}(1-\epsilon)bM \leq n$ elements. It follows that $M \leq \frac{2n}{(1-\epsilon)b}$, and that $M/M^* \leq \frac{2}{1-\epsilon}$. □

Two points are worth noting about the bound in Corollary 3: the first is that even with an empty free list, we may still lose a factor of two in maximum load compared to even load balancing. That this is a real issue and not merely an artifact of the proof can be observed by considering a scenario like the following, in which each *O** represents an empty, open bucket which has not been returned to the free list:

C1-O* | C2-O* | C3-O* ...

Here, while the average load across each group might be roughly the same, one machine might be doing considerably more work than another. It is possible that increasing the amount of virtualization by using more buckets per machine could address such load skews.

A second point is that the quality of the load balancing depends strongly on keeping the free list small. We describe a strategy for doing this by resizing buckets in Section 3.3 below. Such a mechanism is also necessary for handling growing dynamic loads.

The payoff for the complexity of the bucket-handling mechanism is that we can guarantee that any insertion or deletion involves only a small, constant number of machines (not counting any preceding search to find the right location) and a constant number of item relocations. We state this result formally as:

THEOREM 4. *In the basic load-balancing algorithm, any insertion or deletion moves a total of at most 2 items between at most 3 buckets.*

3.3 Dynamic Resizing of Buckets

If the number of keys in the system continues to grow, there are several approaches one might take. The first would be to continue to allocate more and more buckets. Although this would keep the system load-balanced, it increases the likelihood that a pointer dereference will be non-local and also increases the state associated with the skip-graph layer. Therefore, such a scheme threatens the locality that we gained by grouping keys into buckets in the first place. In order to preserve locality and minimize global state, a mechanism is needed by which the number of buckets in the system is proportional to the number of machines in the system, not the number of keys. This is accomplished by dynamically adjusting the threshold number of keys required to classify a bucket as closed. We now proceed to describe such a mechanism, and in turn analyze some of its properties.

We first describe the process of *bucket compression*, which occurs when the threshold is increased. Recall that we begin with some set of buckets, divided into 2-groups and 3-groups. We increase the threshold by a factor of two. Note that all buckets in the system now become open. Our scheme transfers keys between neighboring buckets in order to restore the 2-group/3-group structure described above. The compression proceeds in two phases. During the first phase, each 2-group is combined to form a temporary singleton, returning the left-over bucket to the free list; we also combine each 3-group into a 2-group:

1. C-O ⇒ O
2. C-O-C ⇒ C-O

In Phase 2, we restore the structure by either grouping adjacent singletons together into 2-groups, or by grouping an adjacent 2-group and singleton into either a 2-group or a 3-group:

1. O | O ⇒ C-O
2. C-O | O ⇒ C-O
3. C-O | O ⇒ C-O-C

Note that Case 2 occurs when the total number of keys in the two groups is less than twice the new threshold.

In the case of *bucket decompression*, which occurs when the threshold is decreased by a factor of two, we once again start with a set of 2-groups and 3-groups. Clearly, in order to maintain this structure, we will need to allocate some empty buckets from the free list, based on the number of keys in the group in question.

1. C-O ⇒ C-O-C
2. C-O ⇒ C-O | C-O
3. C-O ⇒ C-O | C-O-C
4. C-O-C ⇒ C-O | C-O-C
5. C-O-C ⇒ C-O-C | C-O-C
6. C-O-C ⇒ C-O-C | C-O | C-O

Case 1 shows that if the number of keys in the open bucket of a 2-group is less than the new threshold value, we can simply form a 3-group. Otherwise, we will need to make either two 2-groups or one 2-group and one 3-group, in order to accommodate the keys according to the decreased threshold. Similar logic can be applied to 3-groups.

To reduce the storage complexity of the skip graph, we implement a scheme in which, on average, one key from each bucket will appear in the skip graph layer. When a key is first inserted into the system, we generate a string of bits at random for the key. Only those keys which match on the first $\log t$ bits, where t is the current bucket threshold, will appear in the skip graph. As the threshold is dynamically adjusted, the number of elements participating in the skip graph will change accordingly.

We observe that this mechanism does not affect the grouping invariant:

LEMMA 5. *The bucket compression mechanism preserves the grouping of nodes into $C-O$ and $C-O-C$ groups.*

COROLLARY 6. *Lemma 2 continues to hold with bucket compression.*

It remains to specify a policy for when to adjust the load threshold for closing buckets. One simple strategy is to double the threshold when the free list becomes empty and halve it when half the buckets are on the free list; this guarantees a maximum free list size of $b/2$ (giving a worst-case maximum load of 4 times the optimum) while providing enough hysteresis to avoid frequent relocation of items. It has the disadvantage of requiring global control to implement, and putting the system through periodic mass migrations as all buckets simultaneously resize. In Section 4, we describe other ways to estimate average system load and also propose a heuristic approach that randomly staggers resizing to avoid simultaneous resizing.

4. ALGORITHM ENHANCEMENTS

The basic algorithm described in the previous section suffers from several problems that led to poor performance in our experiments. In this section, we describe improvements on the algorithm that reduce or eliminate these problems. At present we have only experimental validation of these techniques (described in Section 5).

4.1 Localized resizing of buckets

The biggest problem is caused by simultaneous global resizing of buckets. Not only does this require a global controller, but it also causes many machines to simultaneously attempt to migrate data.

We can eliminate the global controller by employing a distributed algorithm for determining when to resize buckets. We first modify the resizing policy to work based on the average load of buckets, thereby enabling the use of schemes where active buckets exchange load information along with heartbeat messages in order to compute the current system load. One possibility is to exploit the probabilistically balanced property of skip graphs to accumulate system load information in a distributed and replicated manner. Recall that a skip graph comprises of a number of skip lists, each of which could be used to accumulate the current system load after an overall delay associated with sending messages up through the $O(\log b)$ levels of a skip list. Another possibility is to use a distributed sampling approach, wherein pairs of nodes exchange their current estimates of average system loads. In this scheme, each node maintains its estimate of average load as l_n, and when two nodes s and t exchange load information, they average out their estimates and set them to $(l_s + l_t)/2$. This scheme is inspired by previous efforts that have used work stealing to balance load in massively parallel systems (Blumofe and Leiserson [5]) and peer-to-peer systems (Karger and Ruhl [13]). The difference, however, is that the pair-wise interactions are used to only exchange load estimates rather than relocate items. We expect the load estimates to converge quickly (as shown in Figure 11) based on the analysis presented in the earlier papers. The pair-wise interactions could be between random pairs of nodes in the system, or it could be performed along some of the skip graph links. Because the high-level links in the skip graph approximate a random graph, a simple procedure that samples load on neighbors in the skip graph performs well. This sampling operation is still more expensive than a simple search or insert operation, so it is performed only when the number of elements stored at a particular machine increases or decreases by more than a fixed threshold.

To avoid the further problem of simultaneous expansion or contraction, the buckets are organized into groups, each consisting of two closed-open pairs or a single closed-open-closed triplet, and for each group g, two random thresholds are chosen. One is the expansion threshold e_g that satisfies $1/4 < e_g < 1/2$. When the local load estimate of g indicates that the system has utilized a fraction

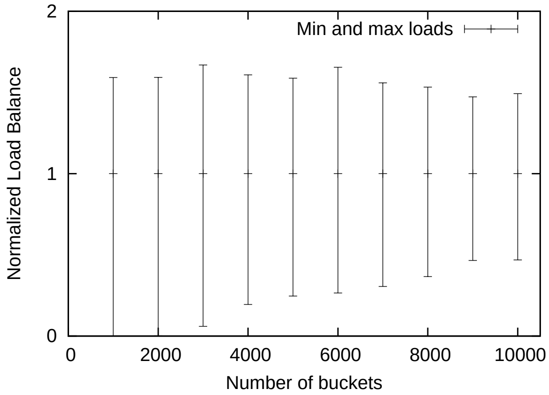

Figure 2: **Normalized loads of machines with our bucketing scheme.**

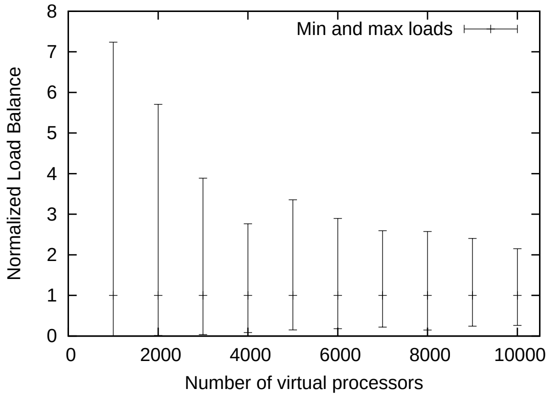

Figure 3: **Normalized loads of machines with keys mapped to machines using SHA-1.**

of maximum capacity that is in excess of e_g, it performs localized expansion. The other threshold is the contraction threshold c_g that satisfies $1/8 < c_g < 1/4$ and is used in a similar manner for contraction. It follows that as new elements are added to the system, expansion or contraction occurs in a staggered fashion, avoiding sudden mass migrations. Furthermore, the ranges for the thresholds are chosen to introduce hysteresis into the system and prevent rapidly alternating bucket expansions and contractions.

The effect of this strategy can be seen by comparing Figures 5 and 6, which show the number of element moves with global bucket resizing under sequential and random insertions, with Figures 7 and 8, which show the corresponding results with local resizing and does not exhibit drastic spikes in data movement traffic.

4.2 Weighted objects

We have seen that one possible way to define the openness or closedness of a bucket is directly related to its maximum number of keys; a bucket becomes closed when the number of keys reaches this threshold value. It is important to note that this is not the only way to define a bucket as "closed," and by adjusting our definition, we can obtain an alternate notion of system-wide load-balance. For example, we can obtain customizable load-balance with the following scheme.

We assign each key a weight, corresponding to the load it represents. This load may reflect storage space requirements, network traffic generated by the object (with more "popular" objects assigned higher weights), or other properties of the object that affect the machine that hosts it. The result is that the machine that owns this key will balance its popular key with fewer keys overall. We can continue to adjust the bucket threshold as before, expanding and contracting buckets as needed. The benefit of such a scheme is that the structure of the bucket layer can more accurately reflect the usage of the individual keys.

4.3 Heterogeneous systems

So far we have been describing a system in which all machines are assumed to have comparable capacity, and load is balanced evenly among machines. In practice, we are likely to find that some machines have much higher storage capacity or network bandwidth than others, and as a result we will want to assign more objects to these high-capacity machines. A simple strategy would be to have high-capacity machines volunteer to hold more buckets, but this increases the number of inter-machine pointers. Instead, it may make sense to define a machine-specific threshold for bucket sizes, so that higher-capacity machines store larger buckets. The disadvantage of such an approach is that it increases the complexity of the system; because of this added complexity, we have not yet carried out experiments to determine if this approach improves performance.

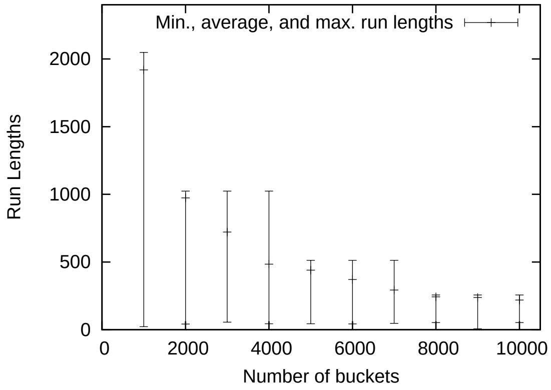

Figure 4: **Run lengths of buckets.**

5. EXPERIMENTAL RESULTS

In this section, we evaluate various aspects of our proposed mechanisms and present results obtained from both a simulator and a real deployment on the PlanetLab infrastructure [17]. We examine the trade-off between locality and load-balance and empirically compare the load-balance obtained by our system with that obtained by systems that use hashing. We then examine the communication costs associated with different kinds of workloads and show that the average number of keys moved per operation is small and that the data movement traffic associated with bucket resizing operations could be staggered to avoid mass simultaneous movements. We then provide measurements that demonstrate that the size of the skip graph is roughly linear in the number of buckets maintained by the system, thereby allowing the system to perform operations using $O(\log b)$ overlay hops.

5.1 Load Balance and Key Locality

We begin by studying the load-balance and key locality properties of our proposed system. We consider the load after inserting a million keys into a system with 1000 machines. We vary the number of buckets per machine from 1 to 10. Figure 2 shows

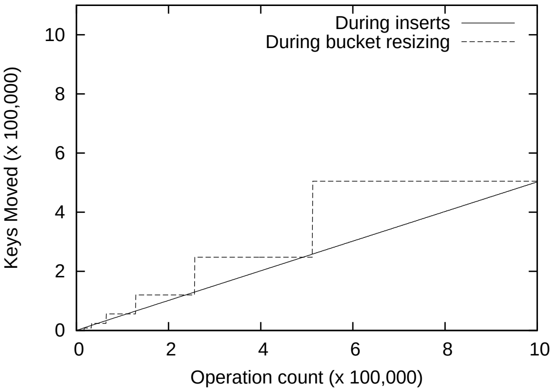

Figure 5: Inserts made in a sequential order under globally synchronized resizing.

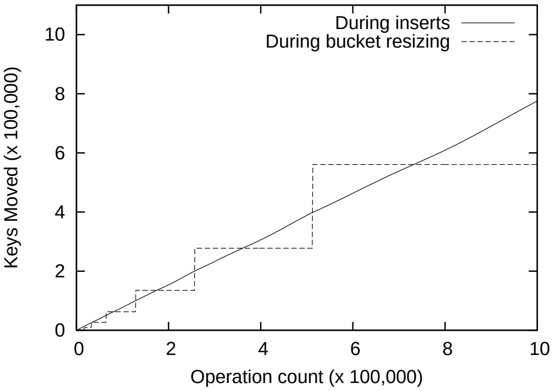

Figure 6: Inserts made in a random order under globally synchronized resizing.

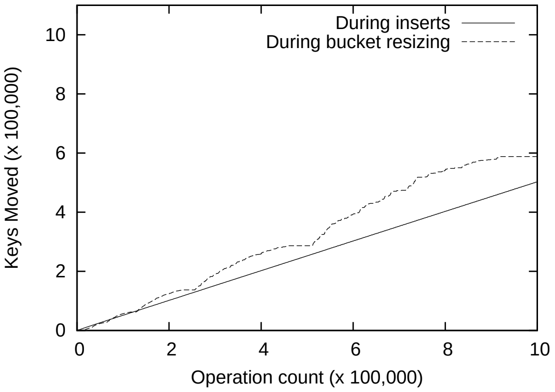

Figure 7: Inserts made in a sequential order under staggered, localized resizing.

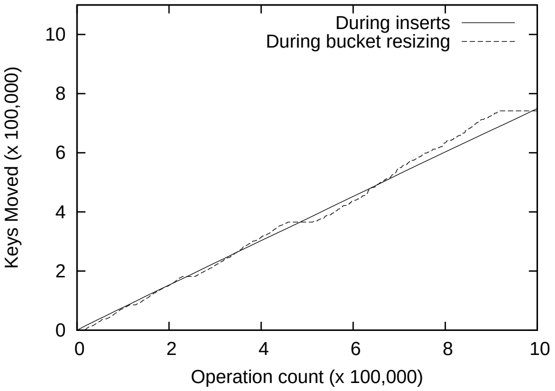

Figure 8: Inserts made in a random order under staggered, localized resizing.

the minimum and maximum loads on machines after normalization. For purposes of comparison, we also consider the key assignments made by a system like Chord that uses the SHA-1 hash function. Figure 3 presents normalized load-balance results for such a scheme as we again vary the amount of virtualization from 1 to 10. Our load-balancing mechanism spreads keys much more evenly across machines.

The load skew could be reduced by increasing the number of buckets assigned to a machine. An increase in virtualization, however, results in a corresponding decrease in the extents of logically consecutive keys assigned to the same machine, as illustrated by Figure 4.

5.2 Key Reorganization Costs

We consider different kinds of workloads under the two bucket restructuring techniques. Our first workload inserts a million keys in sequential order into a 1000 machine system. This workload would evaluate the costs associated with spreading the load when the key insertions are localized to a particular region of the keyspace, namely at the current extreme position. The second workload inserts a million keys in random order. We consider both globally synchronized resizing as well as the staggered mechanism that makes localized changes. We classify the key movements into two categories: those corresponding to offloading inserts made into closed buckets and those corresponding to the resizing of buckets. Figures 5 through 8 demonstrate that the average number of keys moved per operation is less than two and that simultaneous mass migration of data could be avoided through the use of randomized thresholds. Similar behavior is exhibited by the system for a wide variety of workloads, including ones that perform repeated sequences of insertions and deletions localized to a particular interval. Figures 9 and 10 graphs the data movement traffic associated with workloads that repeatedly perform 100,000 insertions followed by 100,000 deletions, thereby triggering a sequence of expansion and contraction operations.

We also evaluated the distributed, sampling algorithm used to compute the average system load. We consider systems with highly skewed loads, where a randomly chosen processor has an extremely high load and all other processors experience minimal load. Each processor begins with its local load as its estimate of the average system load and uses pairwise interactions to refine this estimate. We consider a setting where each processor, in each iteration, interacts with a fixed set of four other processors that are chosen at random, possibly from among the processor's peers at the top-levels of the skip graph. Figure 11 graphs the mean absolute deviation of the local estimate from the actual system average (measured in terms of number of keys) and shows that the local estimates converge after a small number of iterations even for large systems.

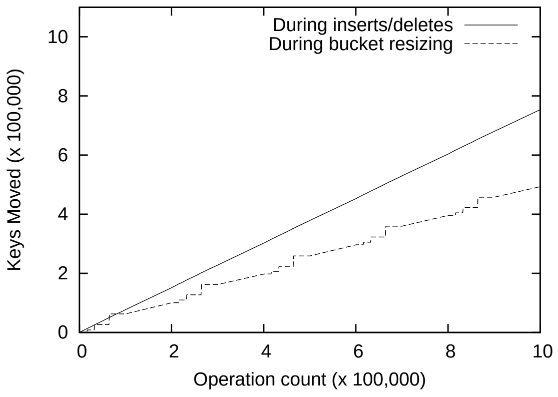

Figure 9: **Repeated sequence of inserts and deletes made in a sequential order under globally synchronized resizing.**

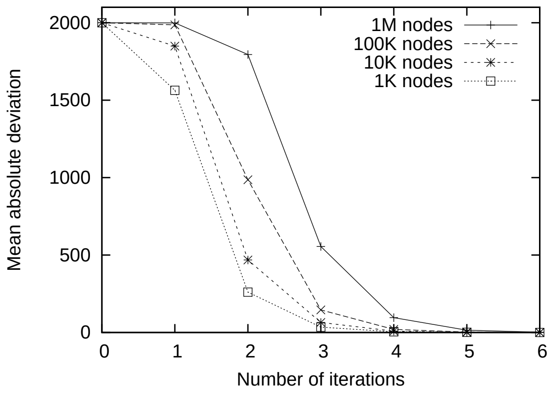

Figure 11: **Convergence of estimate of average load.**

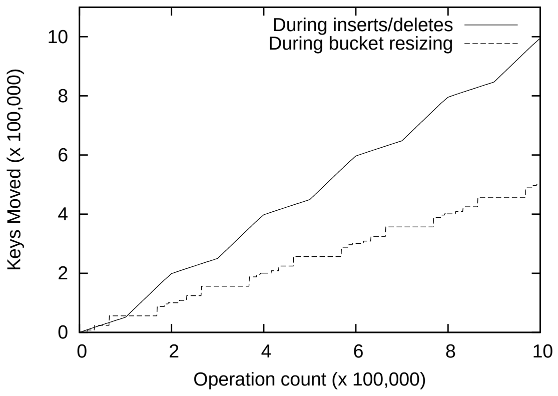

Figure 10: **Repeated sequence of inserts and deletes made in a random order under globally synchronized resizing.**

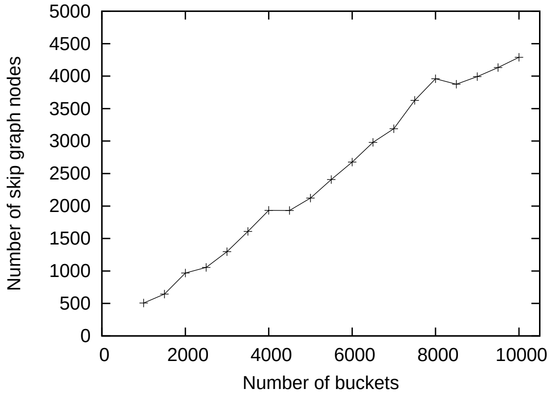

Figure 12: **Number of keys inserted into the skip graph layer.**

5.3 Skip Graph Costs

We next consider the performance of the skip graph that comprises the top layer of our two-layer scheme. Since we select on average one key from each closed bucket to insert into the skip graph, the number of keys in the skip graph is typically less than the number of active buckets in the system. This property limits the amount of global state maintained to $O(\log b)$ per bucket, where b is the number of buckets in the system, as previously observed in Theorem 1 and as illustrated by Figure 12. Furthermore, the cost of an insert, delete, or search operation is simply the cost of performing $O(\log b)$ overlay hops to navigate the skip graph data structure, assuming that the costs of intra-bucket operations are negligible. This is supported by experimental results obtained on a 100-node overlay network (see Figure 13).

6. CONCLUSIONS AND OPEN PROBLEMS

We have described a mechanism for providing load balancing in skip graphs and similar distributed data structures, which both provides better load balancing than the randomized approaches favored by many previous systems and eliminates the excessive inter-machine pointers that plague unmodified skip graphs. In its simplest form, the mechanism is based on a global threshold, where nodes with load below the threshold continue to accept new elements and nodes with load above the threshold attempt to shed elements. We provide a simple heuristic for adjusting this threshold locally, without requiring a global controller, and for avoiding mass migrations caused when all nodes simultaneously adjust their threshold.

Several questions remain. The greatest of these is: though our mechanism appears to work in practice, does it work in theory? We have yet to do a complete analysis of the effect of the distributed threshold heuristic on stability and efficiency of the load-balancing mechanism over very long executions, and it is possible that odd behavior might occur in atypical segments of the skip graph. It would be particularly interesting to analyze the combination of our pairing mechanism and threshold sampling based on the work-stealing algorithm of Karger and Ruhl [13].

Another question that deserves further study is how best to handle arrival and departure of machines. In our current testbed system new machines are simply assigned to the free list, but it is possible that a more sophisticated strategy in which new machines immediately take on load from existing machines could lead to better performance in systems with high turnover. This is closely related to the question of fault-tolerance, since departing machines are unlikely to politely migrate off all of their items before vanishing. We do not address this question at all in the present work, but believe that some sort of local replication strategy should handle all but adversarial faults.

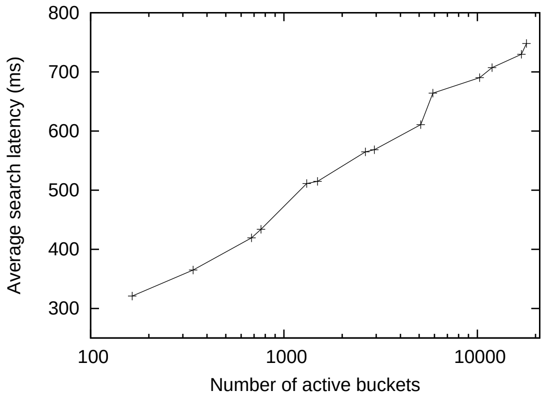

Figure 13: **Search latency on a 100-node PlanetLab overlay network.**

7. REFERENCES

[1] A. Andersson and O. Petersson. Approximate Indexed Lists. *Journal of Algorithms*, 29, 1998.

[2] J. Aspnes and G. Shah. Skip Graphs. In *Proceedings of Symposium on Discrete Algorithms*, 2003.

[3] B. Awerbuch and C. Scheideler. Peer-to-peer systems for Prefix Search. In *Proceedings of the Symposium on Principles of Distributed Computing*, 2003.

[4] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two Simplified Algorithms for Maintaining Order in a List. In *10th European Symposium on Algorithms*, pages 152–164, 2002.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[6] P. Dietz, J. I. Seiferas, and J. Zhang. A Tight Lower Bound for On-Line Monotonic List Labeling. In *4th Scandinavian Workshop on Algorithm Theory*, volume 824 of *Lecture Notes in Computer Science*, pages 131–142. Springer-Verlag, 1994.

[7] P. Dietz and J. Zhang. Lower Bounds for Monotonic List Labeling. In *2nd Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, pages 173–180. Springer-Verlag, 1990.

[8] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In *Proceedings of IPTPS02*, 2002.

[9] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proc. of Fourth USENIX Symposium on Internet Technologies and Systems*, 2003.

[10] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Eighth International Colloquium on Automata, Languages and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431. Springer-Verlag, 1981.

[11] T. Johnson and C. A. A Distributed Data-Balanced Dictionary Based on the B-Link Tree. Technical Report MIT/LCS/TR-530, MIT Laboratory for Computer Science, 1992.

[12] D. Karger, E. Lehman, F. T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of Symposium on Theory of Computing*, 1997.

[13] D. R. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *ACM Symposium on Parallelism in Algorithms and Architectures*, June 2004.

[14] P. Keleher, B. Bhattacharjee, and B. Silaghi. Are virtualized overlay networks too much of a good thing. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems*, 2002.

[15] P. Kirschenhofer and H. Prodinger. The path length of random skip lists. *Acta Inf.*, 31(9), 1994.

[16] T. Papadakis, J. I. Munro, and P. V. Poblete. Analysis of the expected search cost in skip lists. In *Proceedings of the second Scandinavian workshop on Algorithm theory*, 1990.

[17] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Workshop on Hot Topics in Networks (HotNets)*, 2002.

[18] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. In *Workshop on Algorithms and Data Structures*, 1989.

[19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *Proceedings of ACM SIGCOMM*, 2001.

[20] S. Ratnasamy, J. Hellerstein, and S. Shenker. Range Queries over DHTs. Technical Report IRB-TR-03-009, Intel Research, 2003.

[21] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms*, 2002.

[22] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of SIGCOMM*, 2001.

[23] S. Suri, C. D. Tóth, and Y. Zang. Uncoordinated Load Balancing and Congestion Games in P2P Systems. In *Proceedings of the Third International Workshop on Peer-to-Peer Systems*, 2004.

[24] D. E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, Apr. 1992.

[25] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 2004.