An Implementation and Analysis of the Skip Graph Data Structure

Jonathan Kirsch
Advisor: Arvind Krishnamurthy

December 2003

Abstract

This paper describes research done in analyzing the properties of the skip graph, a highly-distributed, fault-tolerant data structure which supports the standard dictionary operations of search, insertion, and deletion. Unlike previous work, which employed distributed hash tables [5, 6], the skip graph can support logarithmic-time complex queries (such as range queries) because it preserves the logical integrity of the key space, storing logically-related keys in close proximity within the data structure. The goal of this paper is to describe those issues left unresolved in the original paper by Aspnes and Shah [1], such as system-wide load-balancing and the need for reduced space complexity. We propose methods for dealing with these issues, and implement a simulator to assess the merit of these methods. We also present the results of a distributed implementation of the skip graph.

1 Introduction

Over the past four or five years, there has been an increasing amount of research conducted in the area of Peer-to-Peer (P2P) systems. While there is some debate regarding the precise definition of “peer-to-peer,” we define such a system as a collection of loosely-coupled processors which interact for the purpose of sharing some resource. This resource could be anything from a data file to CPU processing time.

The first surge of P2P systems arrived in 1999, and included Napster, Gnutella, and Freenet. All three systems used different models and protocols for efficiently locating a given resource. Napster maintained a centralized indexing server, Gnutella used a flooding algorithm in a decentralized environment, and Freenet used routing-based lookups. While popular, each model suffered from considerable limitations. For example, Napster’s centralized scheme limits scalability and creates a single point of failure; if the server goes down, the system is unable to process queries. Similarly, the high message complexity inherent to Gnutella’s flooding algorithm overwhelms the network, and performance therefore degrades as more processors enter the system. In spite of their flaws, the study of these first systems is worthwhile because it helps us to define some of the goals in the design of P2P systems:

Scalability. The system should not be limited to a threshold number of processors. This ties in closely with the notion of performance, in that as the number of processors participating in the system increases, the system should still be able to achieve good performance.

Efficiency. A query should be handled in such a way that it minimizes message costs, response time, and the use of system resources.

Decentralization. A system where each processor is, in some sense, “equal,” is more robust and can handle processor failures gracefully. This is especially important in a distributed environment in which processors can crash, or more generally exit and enter the system, in a dynamic and unpredictable fashion.

In 2001, the “next generation” of P2P systems was developed, including Chord, CAN, and Tapestry[5, 6]. These systems employ a distributed hash table (DHT), where the location of a given resource can be found via a known hashing function (see Section 3.3 for a more detailed description of DHT’s). While the distributed hash table scheme provides efficiency and a low degree of required state, one of its
primary drawbacks is that any resemblance among keys is destroyed by the hashing function. This makes complex queries, such as range queries, inefficient because the system does not use any notion of “key locality,” where two similar keys would hash to nearby locations in the key space. On the other hand, DHT’s benefit from the implicit load-balancing mechanism afforded by the use of a uniform hash function. Thus, all processors in the system handle roughly the same amount of keys, ensuring that no processor is particularly overloaded. Furthermore, system using DHT’s are able to perform locality-improvement optimizations to take advantage of geographic proximity in the underlying network, without upsetting the system’s load-balancing properties. With both the goal of supporting range queries, and the attractive features of DHT’s, in mind, James Aspnes and Gauri Shah developed the skip graph, which is a generalization of Pugh’s skip list[4].

The remainder of this paper is organized as follows. In Section 2, we summarize the basic properties of the skip graph, in preparation for our analysis of the data structure. Section 3 describes in detail the main weaknesses of the skip graph as it was originally proposed by Aspnes and Shah[1]. In particular, we describe those properties we would like to achieve, and discuss the challenge of integrating all of these features into a single system by highlighting the tradeoffs made by existing systems. In Section 4, we propose and analyze a number of modifications to the skip graph which move it closer to the ideal system described in Section 3. Section 5 describes insights gained from simulating the behavior of the skip graph in a uni-processor environment, while Section 6 describes our distributed implementation. Section 7 discusses areas for future work, and Section 8 concludes.

2 The Skip Graph

The skip graph is an extension of the skip list, a randomized, balanced-tree data structure initially proposed by William Pugh in [4]. We briefly describe skip lists to provide motivation for the use of the skip graph in a distributed environment.

2.1 Skip Lists

Skip lists 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 $O(\log k)$ time, where $k$ 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 the bottom-most list, which is referred to as Level 0. Each key appears in the list at Level $i$, for $i > 0$, 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, $O(\log k)$ levels in the system, meaning that a search will traverse $O(\log k)$ keys until it reaches its destination.

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 hotspots, and will be involved in an operation with high probability, potentially overwhelming the processors who own them. Furthermore, the sparsity of the top-level list creates single points of failure: if the processors owning these keys go down, the system will be partitioned. These issues are addressed by the skip graph, which we now describe.

2.2 Skip Graphs

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.”[1] More formally, all keys appear in the list at Level 0. However, 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 key appears in Level $i$ if it can find neighbors who have the same $i$th-length prefix for their membership vectors in Level $i$-1. This continues until the key becomes a singleton, which will result in, on average, $O(\log k)$ levels in the skip graph. For a complete description of the data structure, please see [1].

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.

3 Unresolved Issues

We first list the properties we would ultimately like to achieve in an ideal system. We then describe in detail the extent to which previous work (including the skip graph) has met, or failed to meet, these requirements.

3.1 An Ideal System

Our ideal system exhibits the following properties. We assume that the minimum requirement for such a system is support for $O(\log k)$ searches, insertions, and deletions, where $k$ is the number of keys in the system. To avoid any confusion related to the terminology of the data structures we will be discussing, we make the distinction between keys, or the objects in the data structures, and processors, or those entities performing computations on, and storing, these objects.

Logarithmic space complexity. The amount of space required by the system should be no greater than $O(\log n)$ per processor, where $n$ is the number of objects in the system.

Range-query support. We would like our system to support range queries, such as finding all keys greater than a given key. This would be especially useful in file-sharing systems, since it would allow for efficient queries containing wildcard values, such as “Find The Matrix*”.

System-wide load-balancing. To avoid network bottlenecks or hotspots, processors should each be responsible for roughly $\frac{k}{p}$ keys. This ensures that no processor is doing considerably more work than any other processor.

Geographic locality. For efficiency, we would like most pointer dereferences to be local. More formally, the goal is to achieve at most $O(\log p)$ non-local dereferences, reflecting an effective clustering of logically-related data.

3.2 Limitations of the Skip Graph

One of the primary benefits of the skip graph is its ability to support efficient range queries. As described above, keys are organized in collections of distributed linked-lists, ordered by their logical values. Range queries simply become traversals of these linked-lists, without the need for repeated searches. To pay the cost of supporting complex queries, however, each key must store pointers to an average of two neighbors for each of the $O(\log k)$ levels, where $k$ is the number of keys in the system. The result is a cost of $O(\log k)$ space per key, considerably more than in a distributed hash table.

Another limitation of the skip graph as it was proposed in [1] is that it does not describe the method by which keys are assigned to processors in the system. Therefore, the skip graph makes no guarantees about system-wide load-balancing. Furthermore, if we assume a random distribution of keys around the network, then two keys will be separated by, on average, half the diameter of the network. Thus, the skip graph also makes no guarantees about the geographic locality of neighboring keys. We now examine the properties of distributed hash tables in order to gain insight into potential ways to overcome the shortcomings of the skip graph.

3.3 Limitations of the DHT

In order to analyze the use of a distributed hash table, we use Chord [5] as a model. The Chord protocol uses a consistent hashing scheme to map processors and keys to a single, modular ID space. This results in the formation of the so-called “Chord ring,” with objects (theoretically) uniformly distributed over the ID space. A key $k$ is owned by its successor processor in the system, which is defined as the first processor with an identifier greater than or equal to the identifier of $k$. In order to make look-ups efficient, while minimizing the amount of state needed to perform successful routing, each processor maintains information about $O(\log n)$ objects in the system, where an object is defined as either a processor or a key. This information is kept in routing tables, termed finger tables, containing pointers to objects with hash values of $k + 2^i$, where $k$ is the processor's identifier. This yields a search time of $O(\log n)$, where $n$ is the total number of objects in the system.

An analysis of systems built upon the Chord protocol shows that, without optimization, they satisfy
the goals of logarithmic space complexity and system-wide load-balancing. The former is achieved via the use of finger tables (as described above), and the latter via the hashing mechanism, which scatters objects in a pseudo-random fashion around the ID space. Chord can also exploit the geographic topology of the underlying network by performing a cost-benefit analysis, weighing the latency of taking a particular path against the distance in the Chord ring gained by traversing that path.

The main problem with Chord (and with DHT’s in general), however, is that while we can locate a single key in logarithmic time, finding a series of keys logically related in the key space cannot be done efficiently. The hash function destroys the logical integrity of the key space. Since no state is maintained regarding logical associations among keys, it is believed that systems built upon distributed hash tables cannot support efficient complex queries.

3.4 Combining Skip Graphs and DHT’s

Since we are able to achieve each of the goals outlined at the beginning of this section 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 [2]. 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 search 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, and form what they refer to as a “distributed data structure.” 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 processor 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 processors 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.

4 Skip Graph Modifications

We now suggest ways in which the skip graph can be modified to overcome its shortcomings as described above. The result is a system which has most of the properties of the ideal system, but is currently incomplete.

4.1 Space Complexity

We can reduce the space complexity of a skip graph to \( O(\log p) \) pointers per key, where \( p \) is the number of processors in the system, by limiting the number of keys inserted into the skip graph. Specifically, we group the keys into buckets, with each processor “owning” some subset of all buckets in the system. Each bucket will then elect a representative key to appear in the skip graph, resulting in a skip graph with only \( O(p) \) keys. This mechanism requires only minimal modification to the skip graph algorithms presented in [1], but significantly reduces the amount of state required by the data structure. There is also the potential for a more dynamic method of choosing bucket representatives, in which the number of keys appearing in the skip graph from a single bucket varies with the load in the system. Such a method will be the topic of future research.

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 levels. We refer to the combination of the skip graph and the bucketing level as the two-level system. The top level 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. The lower level consists of the chain of buckets, distributed among the processors in the system. This division is useful because it allows us to make optimizations to the skip graph indirectly, by manipulating the bucket level and the interface between levels, without losing the desirable properties of the top-level skip graph.
The main problem with the bucketing scheme, however, is that by grouping keys together, there is now an increased likelihood for data skew in the system. The processor in charge of a bucket containing the keys beginning with the letters “Th,” for example, will experience much more network traffic than the processor in charge of the “Xy” bucket. More formally, data skew will result when we are using a key space in which keys are distributed unevenly. This is certainly the case if the key space is the set of all valid UNIX file names currently in use. Another issue, beyond the scope of this paper, is how to deal with processor failures in light of the fact that all keys do not appear in the skip graph. Some form of data redundancy might be needed to ensure the connectivity of the original model described in [1].

4.2 Load Balance and Locality

The data skew problem inherent to the bucketing scheme described above illustrates the need for a load-balancing mechanism associated with the skip graph. We first describe the issue of load-balancing generally. We then discuss the limitations of previous approaches, and detail our own scheme.

4.2.1 The Goal

The ultimate goal of any load-balancing mechanism in a distributed system is to utilize the system resources in an efficient manner. This might manifest itself in different ways. For example, one resource in the system is storage space. In a system where machines possess similar hardware, load-balancing might mean distributing data evenly among processors, ensuring that no processor is overloaded. It is also important that a load-balancing mechanism be dynamic and flexible, taking into account the changing state of the system in order to allocate resources efficiently. For example, it is desirable to balance the network traffic in the system so that particular links do not become flooded. A dynamic load-balancing mechanism might adjust certain thresholds or re-route data to achieve this goal.

Load-balancing becomes particularly challenging in a peer-to-peer setting, since we would like to avoid the need for a global resource allocator. Such a global controller would create a single point of failure, and would limit the scalability of the system. Instead, peer-to-peer systems must employ mechanisms by which processors can make localized changes, while still guaranteeing, perhaps probabilistically, system-wide load balance. We consider the problem of balancing keys among processors below.

4.2.2 A Hash-based Approach

The most obvious way to provide load balance in the two-level system is to hash each key into a bucket. Since all processors own roughly the same amount of buckets, a good hash function will map keys to buckets in an effectively random manner. Thus, in theory, each processor will own roughly the same amount of keys, reflecting system-wide load-balance. Note that this does not destroy the ability of the system to support range queries. The top-level skip graph still maintains a linked list of logically-similar keys. We simply traverse this list as needed, using the hash function to locate the key in the bucket level.

This scheme is essentially equivalent to the one proposed by Awerbuch and Scheideler in [2], as described above, and therefore still does not guarantee geographic locality between neighboring keys in the system. Therefore, what is missing is some way to provide load-balance, locality, and the ability to support range queries. We first describe one attempt at achieving these goals, used in the distributed b-tree, and then describe our own variant for use in the skip graph.

4.2.3 A Balanced-Tree Approach

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 processor. To relate this to our two-level system, similar keys would be assigned to the same bucket. This method is used in the DE-tree, a data structure based on the distributed b-tree. In [3], Johnson and Colbrook define an extent to be a “maximal length sequence of neighboring leaves that are owned by the same processor.” Each processor 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 processor.

Once again, however, we are left with a scenario in which we have obtained good geographic locality, but still suffer from potential data skew and loadimbalance. To remedy this, Johnson and Colbrook suggest that local changes can be initially attempted. For example, a heavily-loaded processor can try to
dump some of its keys into the extent of a lightly-loaded, neighboring processor. If all processors are heavily-loaded, a new extent is created for the best candidate processor. This might mean the processor with the least data load, or the processor 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 processor 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 propogate load-balancing information throughout the system quickly, in order to keep the heuristic regarding the election of a candidate processor reasonably efficient and up-to-date. This is undesirable because it means that changes cannot truly be local, and therefore the need to propogate information will increase the message complexity of the system. Clearly we would like a more formal method which provides provable guarantees of load-balancing.

4.3 Our Approach

We now describe our load-balancing mechanism for use in the skip graph. It is built upon the two-level system described in Section 4. To summarize, keys are grouped into buckets according to their values. Each processor owns some subset of all buckets currently in the system, much like the extents found in the dE-tree. Our approach formalizes the heuristic by which keys are transferred between neighboring buckets in the bucket-level.

4.3.1 Description

We first define 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 maintain a “free list” of buckets, and can therefore also classify each bucket as active or free.

We next partition the list of active buckets into groups of 2 or 3, maintaining the invariant that every closed bucket is adjacent to an open bucket. The possible patterns we might see are therefore as follows, 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 accordingly. 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 these algorithms are described below.

4.3.2 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 $C_1$:

1. $C_1$-O2 $\Rightarrow$ C1-O'-C2
2. $C_1$-O2-C3 $\Rightarrow$ C1-O2 $\mid$ C3-O'

In Case 1, the new key is inserted into $C_1$, causing the transfer of a key from $C_1$ 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 $C_1$, causing the transfer of a key from $C_1$ to O2. In turn, O2 transfers a key to C3, and then becomes closed. C3 must then transfer a key to the empty bucket, O'.

4.3.3 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 $\Rightarrow$ C1-O3
2. C1-O2 | C3-O* $\Rightarrow$ C1-O2-C3
3. C1-O2-C3 | C4-O* $\Rightarrow$ 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. If O2 is empty, a key can be shifted from C1 to O2, and then from O2 to C3, resulting in a single 3-group. In Case 3, similar shifting can occur to form two 2-groups.

4.3.4 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 processor owns the entire bucket, at most two processors 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.

The real problem with this load-balancing approach, however, is that it doesn’t actually balance load! Since deletions can occur from any bucket, the system is unable to prevent the scenario in which there are heavily-used, closed buckets adjacent to empty, open buckets. Such a scenario might resemble the following, in which each $O^*$ represents an empty, open bucket which has not been returned to the free list:

$$C1-O^* | C2-O^* | C3-O^* \ldots$$

Therefore, while the average load across each group might be roughly the same, one processor might be doing considerably more work than another. We hope to address this issue in future research by having buckets dynamically change their thresholds according to some estimate of the size of the free list.

Another limitation of the load-balancing mechanism presented is that, while it is effective for relatively fixed-size loads, where the number of keys in the system remains within a given range, it is not currently able to efficiently handle dynamic loads. 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. Therefore, such a scheme threatens the locality that we gained by grouping keys into buckets in the first place!

This fact leads us to consider the second approach, in which the number of keys per bucket changes dynamically. When the system reaches a certain ratio of buckets to keys, the threshold number of keys required before a bucket becomes closed could be adjusted. It remains to be seen, however, exactly when and how the threshold should shift. One problem that arises is that, if the threshold increases, every closed bucket in the system becomes open. While one can imagine a mechanism by which keys are transferred appropriately to restore the 2-3 structure to the bucket level, such an operation would be expensive. Clearly, some hysteresis would also have to be introduced into the system to prevent this expensive operation from repeatedly taking place if the number of keys in the system was to oscillate around the new threshold. Thus, we have seen that integrating good locality and system-wide load-balancing, \textit{without} using a hashing mechanism that would destroy the ability to perform range queries, proves to be very difficult. In the next section, we describe the software used to analyze our load-balancing mechanism.

5 The Simulator

We have written a simulator to test both the skip graph and the underlying load-balancing mechanism. The simulator is written in the C programming language. What follows is an overview of the code, including suggestions for possible optimizations.

5.1 The Top-level Skip Graph

The skip graph is maintained in three primary data structures. Each key in the system is encapsulated by a \texttt{KeyType} structure, containing the value of the key itself, and any information associated with this key, such as the address of the owning processor. For
the purposes of this simulator, all keys have a non-negative integer type, generalized by the 
ValueType typedef. Keys in the system are also bounded by a 
celling value of MAX_KEY, which can be set arbitrarily high, provided it is less than the maximum al-
lowable integer for the given machine. Each KeyType 
is encapsulated by a GraphNode structure, which 
contains the membership vector associated with this 
key, and pointers to this key’s neighbors in the skip 
graph. Finally, the SkipGraph encapsulates the en-
tire top level, maintaining pointers to all GraphN-
odes, and other statistical data, such as the number 
of keys currently in the system. It currently stores 
these handles in an array, and always begins a search 
from the first non-empty GraphNode in this array. 
This could be optimized by storing these handles in 
a tree structure capable of performing an ith element 
look-up. This would extend the functionality of the 
simulator by deciding where an operation should be-
gin.

The top-level skip graph supports the following 
operations:

- SGCreate(): Returns a pointer to a fresh 
  SkipGraph.

- SGSearch(k): Returns the GraphNode corre-
sponding to the closest key not greater than k.

- SGInsert(k): Inserts k into the SkipGraph. 
  For simplicity, it rejects the attempted inser-
tion of duplicate keys.

- SGDelete(k): Deletes k from the SkipGraph, 
  if present.

These operations follow the algorithms described in 
[1] and do not warrant any special explanation.

5.2 The Bucket Level

The bucket level introduces two new data structures. 
The first is the Bucket, which stores a linked-list of 
KeyTypes. This could be optimized to a tree struc-
ture, especially if, for example, the middle key of 
every bucket is to become the representative in the skip 
graph. Each bucket is assigned a state:

- DEFAULT: The Bucket is on the “free list” of 
  empty Buckets.

- OPEN: The Bucket is part of the linked-list 
of active Buckets, but contains fewer than the 
  threshold number of keys.

- CLOSED: The Bucket contains exactly thresh-
  old keys.

Each Bucket is also assigned a type from the set:

\{T2_1, T2_2, T3_1, T3_2, T3_3\}

These types indicate whether the Bucket is on the 
left, middle, or right of a group of two or three. Cur-
rently, all keys appear in the top-level skip graph with 
a probability of \(\frac{1}{3}\). A future version of this simulator 
could adjust this probability dynamically.

We also introduce the System structure, which 
capsulates the entire system by storing pointers to 
the top-level SkipGraph, and the head of the 
list of Buckets. The result is that the user in-
terface supports the operations SystemCreate(), 
SystemSearch(k), SystemInsert(k), and Sys-
ystemDelete(k), corresponding to their counterpart 
functions in the top-level skip graph.

5.3 The Level Interface

One of the attractive features of the two-level scheme 
is that there is a minimal amount of interaction be-	ween the top-level skip graph and the underlying 
bucket-level. The only modification made to the top 
level is that each GraphNode contains a pointer to the 
bucket in which it is located. This pointer then be-
comes the handle by which the two levels can interact. 
A fixed number of Buckets are allocated when the 
System is created. These are represented by GraphN-
odes, with unique, negative key values, in the top-
level skip graph. These GraphNodes contain point-
ers to the empty Buckets. Thus, to obtain an empty 
Bucket from the free list, we choose a random positive 
value modulo MAX_KEY, and negate it. We then 
perform an ordinary SGSearch with this value. If the 
GraphNode returns contains a pointer to a Bucket 
with a state of DEFAULT, we know it is empty, and 
adjust pointers as needed. If not, we try a new ran-
don value. The number of allowable iterations is es-
lished by the constant, FIND_BUCKET_LIMIT.

5.4 An Example

While the algorithms for the two-level scheme trans-
late directly from the description given in Section 4.3, 
we provide a portion of the SystemInsert() algorithm 
to make the type of operations required more con-
crete. In this case, a 3-group might need to be split 
two 2-groups:
Algorithm 1 Insert(Value, Info) into Bucket B of Type T3.1

\[
\begin{align*}
& v \leftarrow \text{SystemSearch}(Value) \\
& \text{if } v = \text{Value} \text{ then} \\
& \quad \text{Print(“Value already present!”)} \\
& \text{else} \\
& \quad k \leftarrow \text{KeyCreate(Value, Info)} \\
& \quad \text{KeyInsert}(k, B) \\
& \quad \text{if } \text{B.KeyTail appears in top-level skip graph then} \\
& \quad \quad \text{Adjust GraphNode containing B.KeyTail to point to B.next} \\
& \quad \quad \text{Move}(\text{B.KeyTail}, \text{B.next}) \{\text{Transfer tail of 3.1 to open 3.2}\} \\
& \quad \text{end if} \\
& \quad \text{if } \text{B.next now has THRESHOLD keys then} \\
& \quad \quad B' \leftarrow \text{FindBlankBucket()} \\
& \quad \quad \text{if } \text{B.next.KeyTail appears in top-level skip graph then} \\
& \quad \quad \quad \text{Adjust GraphNode containing B.next.KeyTail to point to B.next.next} \\
& \quad \quad \quad \text{Move}(\text{B.next.KeyTail}, \text{B.next.next}) \{\text{Transfer tail of 3.2 to closed 3.3}\} \\
& \quad \quad \text{end if} \\
& \quad \quad \text{if } \text{B.next.next.KeyTail appears in top-level skip graph then} \\
& \quad \quad \quad \text{Adjust GraphNode containing B.next.next.KeyTail to point to B'} \\
& \quad \quad \quad \text{Move}(\text{B.next.next.KeyTail}, B') \{\text{Transfer tail of 3.3 to new bucket}\} \\
& \quad \text{end if} \\
& \quad \text{end if} \\
& \quad \text{Update states and types of all 4 buckets} \\
& \quad \text{if FlipCoin() then} \\
& \quad \quad \text{newNode} \leftarrow \text{GNCInsert(Value)} \\
& \quad \quad \text{SGInsert(newNode)} \\
& \quad \text{end if} \\
& \text{end if} \\
\end{align*}
\]

6 The Distributed Prototype

We now describe our implementation of the top-level skip graph in a distributed environment. The current prototype supports the standard skip graph operations described in this paper, as well as operations by which a processor can join or exit the system. While there are many optimizations that can be made to the prototype, it provides evidence of the basic functionality of the skip graph.

6.1 An Overview

The distributed prototype is divided into two programs: \texttt{bootstrap} and \texttt{skipClient}. The \texttt{bootstrap} program acts as a server by which processors can gain entry into the skip graph data structure. We refer to the processor running this program as the \texttt{bootstrap node}. A processor running the \texttt{skipClient} program contacts the \texttt{bootstrap} node at a known location, and on a known port. It obtains a unique identifier, and then inserts a handle to itself into the skip graph by communicating with the bootstrap node. It is important to note that, while the bootstrap node acts as a server (something we would like to avoid in P2P systems) in this prototype, there is no loss of generality in having it do so. One can easily imagine a system in which there are multiple bootstrap nodes, meaning that it will not represent a single point of failure. Processors could also choose their own unique identifiers by using an SHA-1 hash of their IP addresses. Once a processor has placed its handle into the system, it can perform the standard operations of search, insertion, and deletion.

6.2 Implementation Details

The distributed prototype is written using the C programming language, and makes use of TCP sockets. There are two important data structures used, namely the \texttt{GraphNode} and the \texttt{Neighbor}. The
former encapsulates the notion of a key in the system. It contains information such as the value of the key, the address of the owning processor, and a membership vector. The latter encapsulates the notion of a distributed pointer. It contains the address of the processor owning the neighboring GraphNode in the skip graph, as well as the value of the key in that GraphNode. We also make use of a List data structure, which is a linked-list of Nodes, each containing a GraphNode. More intuitively, the keys owned by a processor are maintained in a linked-list format, for simplicity. This could be optimized to a data structure supporting more efficient look-up, such as a balanced tree structure.

Both the bootstrap and skipClient programs are multi-threaded. For the most part, connections are not persistent, meaning that once a processor completes one of the protocols described below, it closes the connection. Certainly there is a tradeoff between the cost of opening a new connection, and storing (and maintaining) file descriptors for each distinct processor to whom a given processor has connected. In either case, both programs require one thread which listens for incoming connections. The main thread processes incoming messages sequentially, reducing the potential for race conditions, but limiting concurrency. For all processors other than the bootstrap node, there is also a thread to process user commands.

6.3 Message Types

The following is a description of the communication taking place between processors in the system. The type of each message, and the size of the incoming message, are written to the sockets as integers. The messages themselves are then sent as space-delimited strings. The terminology used below requires some explanation. Unless otherwise specified, the fields names are from the perspective of the sending processor. The term 
val refers to the value of the key contained in this processor’s Neighbor structure. It is used by the recipient to index into its list of keys. The side field is written by the sending processor in such a way that the receiving processor creates a link on the proper side. P refers to the sending processor, while B refers to the bootstrap node.

JOIN_OP:
To join the system, P simply sends a request to B, who replies with a unique identifier. P then inserts a new GraphNode containing this identifier into the skip graph by sending a search request to B. B replies with the address of a potential neighbor, allowing insertion to continue as it would for any other key.

SEARCH_OP: startNode searchKey level 
val
Searches are carried out from the perspective of P's handle into the skip graph. To initiate a search, P sends its own address, along with the search key, to its neighbor on the proper side. The search is to begin at level, which represents the highest level of which P's handle is a part.

SEARCH_REPLY: closestKey owningProcessor
When a search has been completed, the processor owning the closest key sends the key's value and its own address to the startNode obtained from the original SEARCH_OP.

LINK_OP: myAddress myKey side level 
val
P sends a message to its potential neighbors at a given level, indicating that it would like to link in to the neighbor at this level. The recipient thus needs to know the address and value of P so that it can create a new Neighbor structure. If another processor has linked to the recipient already, the message is forwarded accordingly.

LINK_REPLY: myAddress myKey side level yourKey fixLink
After successfully linking to another processor, P sends back its own address and key, so that the link requester will know to create a Neighbor structure containing those values. P also sends the 
val of the link requester so that it can index into its list of keys. Normally, upon receiving a LINK_REPLY, a processor will send a message to its former neighbor, with information regarding how to link to the new shared neighbor. The fixLink flag is used to indicate that the receiving processor should only update (or fix) its own links, and not send another LINK_OP to its former neighbor. This prevents messages from being bounced back and forth without end.

BUDDY_OP:
startNode level memVectorValue 
val side
To find a neighbor for Level i, P sends a message containing its own address and the ith bit of its membership vector. This message is for-
warded recursively to neighboring nodes if no match is found.

BUDDY_REPLY : myAddress myKey
Upon receiving a BUDDY_OP, if a processor matches on the corresponding bit of the membership vector, it sends a message to the startNode with its information. This information is then used by startNode to construct a LINK_OP.

DELETE_OP : neighbor value side level val
To delete a key, P sends messages to its left and right neighbors, with the address and key values of the opposite side. For example, P sends information about its right neighbor to its left neighbor. This is then used to set up the correct Neighbor structures.

6.4 Current Limitations
The prototype described above is essentially a barebones implementation of the top-level skip graph. Future versions of the code will install a load-balancing mechanism. They will also address the important issue of data structure repair. If nodes fail in the system, the skip graph can still maintain a high level of connectivity by fixing its links. Aspnes and Shah refer to this as the “zipper” mechanism [1], and it is essential to making the skip graph applicable to practical use. With these additional features in place, we would then like to run tests, using the PlanetLab network, to assess the performance capabilities of the prototype.

7 Future Work
Throughout this paper, we have noted many issues which require further research. We now list several issues which we have not yet touched upon. One such issue relates to data replication in the skip graph (and in the bucket layer). Currently, while the failure of a process might not result in a partitioned top-level skip graph, it does result in a loss of data availability for those keys owned by the failing processor. The challenge becomes determining which data to replicate, and how to do so without greatly increasing the amount of state needed by the system. Similarly, it is not yet clear how processors could route around failures in the bucket level, since it is currently a simple doubly-linked list. Finally, the shortcomings of our load-balancing approach suggests that a probabilistic scheme, which imposes less structure on the buckets, might be able to give us the flexibility we need to find a reasonable balance between geographic locality and load-balance.

8 Conclusions
This paper has attempted to address the major challenges currently surrounding the skip graph data structure. While there is still much research to be done, it is worth noting that no feature in the skip graph presents an obvious barrier to achieving the ideal system we have outlined. By creating two orthogonal data structures, we will be able to focus our research on specific optimizations to particular aspects of the system, without sacrificing the beneficial properties already present. Therefore, we feel that the skip graph, with the correct set of modifications, can potentially overcome its current shortcomings, and find practical use in the modern computing world.

Special thanks to Arvind Krishnamurthy for his help and guidance through every stage of this project, and to James Aspnes for allowing me to hack away at his data structure.

References