Skip Graphs

JAMES ASPNES

Yale

AND

GAURI SHAH

Google

Abstract. Skip graphs are a novel distributed data structure, based on skip lists, that provide the full functionality of a balanced tree in a distributed system where resources are stored in separate nodes that may fail at any time. They are designed for use in searching peer-to-peer systems, and by providing the ability to perform queries based on key ordering, they improve on existing search tools that provide only hash table functionality. Unlike skip lists or other tree data structures, skip graphs are highly resilient, tolerating a large fraction of failed nodes without losing connectivity. In addition, simple and straightforward algorithms can be used to construct a skip graph, insert new nodes into it, search it, and detect and repair errors within it introduced due to node failures.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*Distributed data structures*

General Terms: Algorithms

Additional Key Words and Phrases: Peer-to-peer, skip lists, overlay networks

ACM Reference Format:

Aspnes, J. and Shah, G. 2007. Skip graphs. ACM Trans. Algor. 3, 4, Article 37 (November 2007), 25 pages. DOI = 10.1145/1290672.1290674 http://doi.acm.org/10.1145/1290672.1290674

1. *Introduction*

Peer-to-peer networks are distributed systems without any central authority, which are used for efficient location of shared resources. Such systems have become very popular for Internet applications in a short period of time. A survey of recent

This research was supported in part by NSF Grants CCR-9820888 and CCR-0098078 for the first author and NSF Grants CCR-9820888 and CCR-0098078 for the second author. Much of the work described in this article was performed at Yale University.

Authors' addresses: J. Aspnes, Department of Computer Science, Yale University, New Haven, CT 06520-8285, e-mail: aspnes@cs.yale.edu; G. Shah, Google Inc., 1600 Amphitheatre Pkwy., Mountain View, CA 94043, e-mail: gauri.shah@aya.yale.edu.

peer-to-peer research yields a slew of desirable features for peer-to-peer systems, such as decentralization, scalability, fault tolerance, self-stabilization, data availability, load balancing, dynamic addition and deletion of peer nodes, efficient and complex query searching, incorporating geography in searches, and exploiting spatial as well as temporal locality in searches. The initial approaches, such as those used by Napster [2003], Gnutella [2007], and Freenet [Clarke et al. 2000], do not support most of these features and are clearly unscalable, either due to the use of a central server (Napster) or due to high message complexity from performing searches by flooding the network (Gnutella). The performance of Freenet is difficult to evaluate, but it provides no provable guarantee on the search latency and permits accessible data to be missed.

Recent peer-to-peer systems like CAN [Ratnasamy et al. 2001], Chord [Stoica et al. 2003], Pastry [Rowstron and Druschel 2001], Tapestry [Zhao et al. 2001] and Viceroy [Malkhi et al. 2002] use a *distributed hash table* (DHT) approach to overcome scalability problems. To ensure scalability, they hash the key of a resource to determine at which node it will be stored and balance out the load on the nodes in the network. The main operation in these networks is to retrieve the identity of the node which stores the resource, from any other node in the network. To this end, there is an overlay graph in which the location of nodes and resources is determined by the hashed values of their identities and keys, respectively. Resource location using the overlay graph is done in these various networks by using different routing algorithms. Pastry and Tapestry use the algorithm of Plaxton et al. [1999], which is based on hypercube routing: The message is forwarded deterministically to a neighbor whose identifier is one digit closer to the target identifier. CAN partitions a d-dimensional coordinate space into *zones* that are owned by nodes which store keys mapped to their zone. Routing is done by greedily forwarding messages to the neighbor closest to the target zone. Chord maps nodes and resources to identities of b bits placed around a *modulo* 2^b *identifier circle*, each node maintaining links to distances $2^0, 2^1 \ldots$ for greedy routing. With m machines in the system, most of these networks use $O(\log m)$ space and time for routing, and $O(\log m)$ time for node insertion (with the exception of Chord, which takes $O(\log^2 m)$ time). Because hashing destroys the ordering on keys, DHT systems do not support queries that seek near matches to a key, or to keys within a given range.

Some of these systems try to optimize performance by taking topology into account. Pastry [Rowstron and Druschel 2001; Castro et al. 2002] and Tapestry [Zhao et al. 2002; 2001] exploit geographical proximity by choosing the physically closest node out of all possible nodes with an appropriate identifier prefix. In CAN [Ratnasamy et al. 2001], each node measures its round-trip delay to a set of landmark nodes, and accordingly places itself in the coordinate space to facilitate routing with respect to geographic proximity. This last method is not fully self-organizing and may cause imbalance in the distribution of nodes, leading to hot spots. Some methods to solve the nearest neighbor problem for overlay networks can be seen in Hildrum et al. [2002] and Karger and Ruhl [2002].

Some of these systems are partly resilient to random node failures, but their performance may be badly impaired by adversarial deletion of nodes. Fiat and Saia [2002] present a network which is resilient to adversarial deletion of a constant fraction of nodes; some extensions of this result can be seen in Saia et al. [2002] and Datar [2002]. However, they do not give efficient methods to dynamically maintain such a network.

TerraDir [Silaghi et al. 2002] is a recent system that provides locality and maintains a hierarchical data structure using caching and replication. There are as of yet no provable guarantees on load balancing and fault tolerance for this system.

1.1. OUR APPROACH. The underlying structure of Chord, CAN, and similar DHTs resembles a balanced tree in which balancing depends on near-uniform distribution of the output of the hash function. So, the costs of constructing, maintaining, and searching these data structures are closer to the $\Theta(\log n)$ costs of tree operations than to the $\Theta(1)$ costs of traditional hash tables. But because keys are hashed, DHTs can provide only hash table functionality. Our approach is to exploit the underlying tree structure to give tree functionality, while applying a simple distributed balancing scheme to preserve balance and distribute load.

We describe a new model for a peer-to-peer network based on a distributed data structure that we call a *skip graph*. This distributed data structure has several benefits: Resource location and dynamic node addition and deletion can be done in logarithmic time, and each node in a skip graph requires only logarithmic space to store information about its neighbors. More importantly, there is no hashing of the resource keys, so related resources are present near each other in a skip graph. This may be useful for certain applications such as prefetching of webpages, enhanced browsing, and efficient searching. Skip graphs also support *complex queries* such as range queries, namely, locating resources whose keys lie within a certain specified range.[1] There has been some interest in supporting complex queries in peer-to-peer-systems, and designing a system that supports range queries was posed as an open question [Harren et al. 2002]. Skip graphs are resilient to node failures: A skip graph tolerates removal of a large fraction of its nodes chosen at random without becoming disconnected, and even the loss of an $O(\frac{1}{\log n})$ fraction of the nodes chosen by an adversary still leaves most of the nodes in the largest surviving component. Skip graphs can also be constructed without knowledge of the total number of nodes in advance. In contrast, DHT systems such as Pastry and Chord require a priori knowledge about the size of the system or its keyspace. While the size of the key (2^{160} bits) in these systems is sufficient for all practical purposes, it is still an interesting theoretical property that skip graphs do not need to know the size of the keyspace in advance.

The rest of the article is organized as follows: We describe skip graphs and algorithms for them in detail in Sections 2 and 4. Section 3 covers some implementation issues. We describe the fault-tolerance properties of a skip graph in Section 5 and the contention properties Section 6. Finally, we discuss some recent related work in Section 7 and conclude in Section 8.

1.2. MODEL. We briefly describe the model for our algorithms. We assume a *message passing* environment in which all processes communicate with each other by sending messages over a communication channel. The system is *partially synchronous*, that is, there is a fixed upper bound (time-out) on the transmission delay of a message. Processes can *crash*, that is, halt prematurely, and crashes are permanent. We assume that each message takes at most

[1] Skip graphs support complex queries along a single dimension, that is, for one attribute of the resource, for example, its name key.

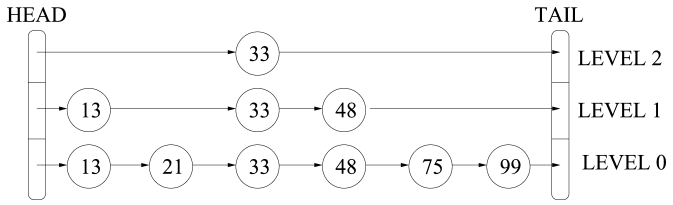

FIG. 1. A skip list with $n = 6$ nodes and $\lceil \log n \rceil = 3$ levels.

unit time to be delivered and any internal processing at a machine takes no time.

2. *Skip Graphs*

A *skip list*, introduced by Pugh [1990], is a randomized balanced tree data structure organized as a tower of increasingly sparse linked lists. Level 0 of a skip list is a linked list of all nodes in increasing order by key. For each i greater than 0, each node in level $i-1$ appears in level i independently with some fixed probability p. In a doubly-linked skip list, each node stores a predecessor pointer and a successor pointer for each list in which it appears, for an average of $\frac{2}{1-p}$ pointers per node. Lists at the higher level act as "express lanes" that allow the sequence of nodes to be traversed quickly. Searching for a node with a particular key involves searching first in the highest level, and repeatedly dropping down a level whenever it becomes clear that the node is not in the current one. Considering the search path in reverse shows that no more than $\frac{1}{1-p}$ nodes are searched on average per level, giving an average search time of $O(\log n \frac{1}{(1-p)\log \frac{1}{p}})$ with n nodes at level 0. Skip lists have been extensively studied [Pugh 1990; Papadakis et al. 1990; Devroye 1992; Kirschenhofer and Prodinger 1994; Kirschenhofer et al. 1995], and, because they require no global balancing operations, are particularly useful in parallel systems [Gabarró et al. 1996; Gabarró and Messeguer 1997].

We would like to use a data structure similar to a skip list to support typical binary tree operations on a sequence whose nodes are stored at separate locations in a highly distributed system subject to unpredictable failures. A skip list alone is not enough for our purposes because it lacks redundancy and is thus vulnerable to both failures and congestion. Since only a few nodes appear in the highest-level list, each such node acts as a single point of failure whose removal partitions the list, and forms a hot spot that must process a constant fraction of all search operations. Skip lists also offer few guarantees that individual nodes are not separated from the rest, even with occasional random failures. Since each node is connected on average to only $O(1)$ other nodes, even a constant probability of node failures will isolate a large fraction of the surviving nodes.

Our solution is to define a generalization of a skip list that we call a *skip graph*. As in a skip list, each of the n nodes in a skip graph is a member of multiple linked lists. The level-0 list consists of all nodes in sequence. Where a skip graph is distinguished from a skip list is that there may be many lists at level i, and every node participates in one of these lists, until the nodes are splintered into singletons after $O(\log n)$ levels, on average. A skip graph supports search, insert, and delete operations analogous to the corresponding operations for skip lists; indeed, we show

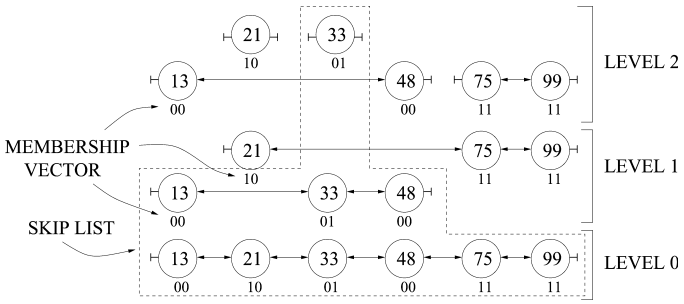

FIG. 2. A skip graph with $n = 6$ nodes and $\lceil \log n \rceil = 3$ levels.

in Lemma 1 that algorithms for skip lists can be applied directly to skip graphs, as a skip graph is equivalent to a collection of n skip lists that happen to share some of their lower levels.

Because there are many lists at each level, the chances that any individual node participates in some search is small, eliminating both single points of failure and hot spots. Furthermore, each node has $\Theta(\log n)$ neighbors on average, and with high probability ($1 - [1/p]^{\log n}$), where p is the probability of node failure), no node is isolated. In Section 5 we observe that skip graphs are resilient to node failures and have an expansion ratio of $\Omega(\frac{1}{\log n})$ with n nodes in the graph.[2]

In addition to providing fault tolerance, having an $\Omega(\log n)$ degree to support $O(\log n)$ search time appears to be necessary for distributed data structures based on nodes in a one-dimensional space linked by random connections satisfying certain uniformity conditions [Aspnes et al. 2002]. While this lower bound requires some independence assumptions that are not satisfied by skip graphs, there is enough similarity between the latter and the class of models considered in the bound that an $\Omega(\log n)$ average degree is not surprising.

We now give a formal definition of a skip graph. Precisely to which lists a node x belongs is controlled by a *membership vector* $m(x)$. We think of $m(x)$ as an infinite random word over some fixed alphabet, although in practice, only an $O(\log n)$-length prefix of $m(x)$ need be generated on average. The idea of the membership vector is that every linked list in the skip graph is labeled by some finite word w, and a node x is in the list labeled by w if and only if w is a prefix of $m(x)$.

To reason about this structure formally, we will need some notation. Let Σ be a finite alphabet, let Σ^* be the set of all finite words consisting of characters in Σ, and let Σ^∞ consist of all infinite words. We use subscripts to refer to individual characters of a word, starting with subscript 0; a word w is equal to $w_0 w_1 w_2 \ldots$. Let $|w|$ be the length of w, with $|w| = \infty$ if $w \in \Sigma^\infty$. If $|w| \geq i$, write $w \restriction i$ for the prefix of w of length i. Write ϵ for the empty word. If v and w are both words, write $v \preceq w$ if v is a prefix of w, namely, if $w \restriction |v| = v$. Write w_i for the ith character of the word w. Write $w_1 \wedge w_2$ for the common prefix (possibly empty) of the words w_1 and w_2.

Returning to skip graphs, the bottom level is always a doubly-linked list S_ϵ consisting of all the nodes in order, as shown in Figure 2. In general, for each w

[2] Since the initial publication of these results, the bound on the expansion has been improved to $\Omega(1)$ by Aspnes and Wieder [2005].

in Σ^*, the doubly-linked list S_w contains all x for which w is a prefix of $m(x)$, in increasing order. We say that a particular list S_w is part of level i if $|w| = i$. This gives an infinite family of doubly-linked lists; in an actual implementation, only those S_w with at least two nodes are represented. A skip graph is precisely a family $\{S_w\}$ of doubly-linked lists generated in this fashion. Note that because the membership vectors are random variables, each S_w is also a random variable.

We can also think of a skip graph as a random graph where there is an edge between x and y whenever x and y are adjacent in some S_w. Define x's left and right neighbors at level i as its immediate predecessor and successor, respectively, in $S_{m(x)\restriction i}$, or $\perp$ if no such nodes exist. We will write xL_i for x's left neighbor at level i and xR_i for x's right neighbor, and in general will think of the R_i's as forming a family of associative composable operators to allow writing expressions like $xR_iR_{i-1}^2$, etc. We write x.maxLevel for the first level ℓ at which x is in a singleton list, that is, x has at least one neighbor at level $\ell - 1$.

An alternative view of a skip graph is a *trie* [de la Briandais 1959; Fredkin 1960; Knuth 1973] of skip lists that share their lower levels. If we think of a skip list formally as a sequence of random variables $S_0, S_1, S_2, \ldots$, where the value of S_i is the level-i list, then we have the following lemma.

LEMMA 1. *Let $\{S_w\}$ be a skip graph with alphabet Σ. For any $z \in \Sigma^\infty$, the sequence $S_0, S_1, S_2, \ldots$, where each $S_i = S_{z\restriction i}$, is a skip list with parameter $p = |\Sigma|^{-1}$.*

PROOF. By induction on i. The list S_0 equals S_ϵ, which is just the base list of all nodes. A node x appears in S_i if $m(x) \restriction i = z \restriction i$; conditioned on this event occurring, the probability that x also appears in S_{i+1} is just the probability that $m(x)_{i+i} = z_{i+1}$. This event occurs with probability $p = |\Sigma|^{-1}$, and it is easy to see that it is independent of the corresponding event for any other x' in S_i. Thus, each node in S_i appears in S_{i+1} with independent probability p, and $S_0, S_1, \ldots$ form a skip list. □

For a node x with membership vector $m(x)$, let the skip list $S_{m(x)}$ be called the *skip-list restriction* of node x.

3. *Implementation Issues*

In an actual implementation of a peer-to-peer system using a skip graph, each node in a skip graph will be a resource. The resources are sorted in increasing lexicographic order of their keys. Mapping these keys to actual physical machines can be done in two ways: In the first approach, we make every machine responsible for the resources that it hosts. Alternatively, we use a DHT approach where we hash node identifiers and resource keys to determine which nodes will be responsible for which keys. The first approach gives security and manageability, whereas the second gives good load balancing. For now, we treat nodes in the skip graph as representing resources, and present our results without committing to how these resources are distributed across machines. Each node in a skip graph stores the address and key of its successor and predecessor at each of the $O(\log n)$ levels. In addition, each node also needs $O(\log n)$ bits of space for its membership vector.

In both of the aforementioned approaches, with n resources in the network, each machine is responsible for maintaining $O(\log n)$ links for *each* resource that it hosts,

TABLE I. LIST OF ALL THE VARIABLES STORED AT EACH NODE

Variable	Type
key	Resource key
neighbor[R]	Array of successor pointers
neighbor[L]	Array of predecessor pointers
m	Membership vector
maxLevel	Integer
deleteFlag	Boolean

for a total of $O(n \log n)$ links in the entire network. This is a much higher storage requirement than the potentially-maximum $O(m \log m)$ links for DHTs, where m is the number of machines in the system. Further, this may lead to high message traffic if each machine periodically checks to see that its links are functional, as common in many P2P implementations. Recently, a new data structure called a family tree [Harvey and Zatloukal 2004] was introduced which overcomes these limitations. A family tree supports locality queries using a constant number of pointers per node, and expected search/update time logarithmic in the number of nodes.

A central issue in implementing a skip graph is managing the many doubly-linked lists. We do not attempt to solve here the problem of constructing reliable concurrent doubly-linked lists, but instead assume use of some existing concurrent doubly-linked list implementation such as that of Johnson and Colbrook [1994] as used in Abraham et al. [2005]. For our basic algorithms we assume no failures, and rely on a separately-invoked repair mechanism as described in Section 5.3 to recover from the loss of individual nodes. For more sophisticated approaches to the issues involved in implementing distributed rings with concurrency and failures, see Li et al. [2004] and Shaker and Reeves [2005].

4. *Algorithms for a Skip Graph*

In this section, we describe the search, insert, and delete operations for a skip graph. For simplicity, we refer to the key of a node (e.g., x.key) with the same notation (e.g., x) as the node itself. It will be clear from the context whether we refer to a node or its key. In the algorithms, we denote the pointer to x's successor and predecessor at level ℓ as x.neighbor[R][ℓ] and x.neighbor[L][ℓ], respectively. We define xR_ℓ formally to be the value of x.neighbor[R][ℓ], if x.neighbor[R][ℓ] is a nonnil pointer to a nonfaulty node, and $\perp$ otherwise. We define xL_ℓ similarly. We summarize the variables stored at each node in Table I.

We assume that the pointer variables are maintained by an underlying deadlock-free implementation of a distributed sorted doubly-linked list, as in Johnson and Colbrook [1994]. We assume that operations on this doubly-linked list can be treated as atomic, and that it supports search, insert, and delete operations, each taking $O(k)$ time and $O(k)$ messages starting from a node k hops away from its target. We also assume that operations on different doubly-linked lists can be performed in parallel without interference.

In Sections 4.1 to 4.3, we give the algorithms for skip graph operations and analyze their performance. We defer proofs of correctness under concurrency to Section 4.4.

4.1. THE SEARCH OPERATION. The search operation (Algorithm 1) is identical to the search in a skip list, with only minor adaptations to run in a distributed system. The search is started at the topmost level of the node seeking a key and proceeds along each level without overshooting the key, continuing at a lower level if required, until reaching level 0. Either the address of the node storing the search key, if such exists, or that of the node storing the largest key less than (or smallest key greater than) the search key is returned.

Algorithm 1. search for node v

```
1  upon receiving ⟨searchOp, startNode, searchKey, level⟩:
     // If the current key is the search key, return
2  if (v.key = searchKey) then
3  |  send ⟨foundOp, v⟩ to startNode
     // Coming from left, current key is less than search key
4  if (v.key < searchKey) then
   |  // Go one level down till a right neighbor with key smaller than the search key is reached
5  |  while level ≥ 0 do
6  |  |  if ((v.neighbor[R][ level].key ≤ searchKey) then
7  |  |  |  send ⟨searchOp, startNode, searchKey, level⟩ to v.neighbor[R][ level]
8  |  |  |  break
9  |  |  else level←level-1
     // Coming from right, current key is greater than search key
10 else
11 |  while level ≥ 0 do
   |  |  // Go one level down till a left neighbor with key larger than the search key is reached
12 |  |  if ((v.neighbor[L][level]).key ≥ searchKey) then
13 |  |  |  send ⟨searchOp, startNode, searchKey, level⟩ to v.neighbor[L][ level]
14 |  |  |  break
15 |  |  else level←level-1
     // Reached the bottom-most level, key is not found
16 if (level < 0) then
17 |  send ⟨notFoundOp, v⟩ to startNode
```

LEMMA 2. *The search operation in a skip graph S with n nodes takes expected $O(\log n)$ messages and $O(\log n)$ time.*

PROOF. Let Σ be the alphabet for the membership vectors of nodes in the skip graph S, and z be the node at which the search starts. By Lemma 1, the sequence $S_{m(z)} = S_0, S_1, S_2, \ldots$, where each $S_i = S_{m(z)\restriction i}$, is a skip list. A search that starts at z in the skip graph will follow the same path in S as in $S_{m(z)}$. Thus, we can directly apply the skip-list search analysis given in Pugh [1990], to analyze the search in S. This analysis shows that with n nodes, on average there will be $O(\log n \frac{1}{\log(1/p)})$ levels, for $p = |\Sigma|^{-1}$. At most $\frac{1}{1-p}$ nodes are searched on average at each level, for a total of $O(\log n \frac{1}{(1-p)\log(1/p)})$ expected messages and $O(\log n \frac{1}{(1-p)\log(1/p)})$ expected time. Thus, with fixed p, the search operation takes expected $O(\log n)$ messages and $O(\log n)$ time. □

The network performance depends on the value of $p = |\Sigma|^{-1}$. As p increases, the search time decreases but the number of levels increase, so each node has to maintain neighbors at more levels. Thus we get a tradeoff between search time and storage requirements at each node.

The performance shown in Lemma 2 is comparable to that of distributed hash tables, for example, Chord [Stoica et al. 2003]. With n resources in the system,

a skip graph takes $O(\log n)$ time for one search operation. In comparison, Chord takes $O(\log m)$ time, where m is the number of machines in the system. As long as n is polynomial in m, we get the same asymptotic performance from both DHTs and skip graphs for search operations.

Skip graphs can support *range queries* in which one is asked to find a key $\geq x$, a key $\leq x$, the largest key $< x$, the least key $> x$, some key in the interval $[x, y]$, all keys in $[x, y]$, and so forth. For most of these queries, the procedure is an obvious modification of Algorithm 1 and runs in $O(\log n)$ time with $O(\log n)$ messages. For finding all nodes in an interval, we can use a modified Algorithm 1 to find a single element of the interval (which takes $O(\log n)$ time and $O(\log n)$ messages). With r nodes in the interval, we can then broadcast the query through all the nodes (which takes $O(\log r)$ time and $O(r \log n)$ messages). If the originator of the query is capable of processing r simultaneous responses, the entire operation still takes $O(\log n)$ time.

4.2. THE INSERT OPERATION. A new node u knows some *introducing* node v in the network that will help it to join the network. Node u inserts itself in one linked list at each level until it finds itself in a singleton list at the topmost level. The insert operation consists of two stages, as described next.

(1) Node u starts a search for itself from v to find its neighbors at level 0, and links to them.

(2) Node u finds the closest nodes s and y at each level $\ell \geq 0$, $s < u < y$, such that $m(u) \restriction (\ell + 1) = m(s) \restriction (\ell + 1) = m(y) \restriction (\ell + 1)$, if they exist, and links to them at level $\ell + 1$.

Because each existing node v does not require $m(v)_{\ell+1}$ unless there exists another node u such that $m(v) \restriction (\ell + 1) = m(u) \restriction (\ell + 1)$, it can delay determining its value until a new node arrives asking for its value; thus, at any given time only a finite prefix of the membership vector of any node need be generated. Detailed pseudocode for the insert operation is given in Algorithm 2.

Algorithm 2. insert for new node u

```
1  execute search from introducer to find max s < u
2  ℓ ← 0
3  while true do
4  |  insert u in list at level ℓ starting from s
5  |  scan backwards at level ℓ to find s′ such that m(s′) ↾ (ℓ + 1) = m(u) ↾ (ℓ + 1)
6  |  if no such s′ exists then
7  |  |  exit
8  |  else
9  |  |  s ← s′
10 |  |  ℓ ← ℓ + 1
```

LEMMA 3. *In the absence of concurrency, the insert operation in a skip graph S with n nodes takes expected $O(\log n)$ messages and $O(\log n)$ time.*

PROOF. Fix $p = |\Sigma|$. With n nodes, there will be an average of $O(\log n)$ levels in the skip graph. To link at level 0, a new node u performs one search operation. From Lemma 2, this takes $O(\log n)$ expected messages and $O(\log n)$ expected time.

At each level ℓ, $\ell \geq 0$, u communicates with an average of $1/p$ nodes in line 5 before it finds its level $\ell + 1$ predecessor s'; this requires an additional $O(1/p) = O(1)$ expected time and messages. The insert operation is assumed to take $O(1)$ time from this point. Summing over all levels and including the cost of the initial search in line 1 gives $O(\log n)$ total time and messages. □

With concurrent inserts, the cost of a particular insert may be arbitrarily large. In addition to any costs imposed by the underlying doubly-linked list implementation, there is the possibility of starvation in line 5, as new nodes are inserted faster than u can skip over them. We do not expect this problem to be common in practice.

With m machines and n resources in the system, most DHTs such as CAN, Pastry, and Tapestry take $O(\log m)$ time for insertion; an exception is Chord, which takes $O(\log^2 m)$ time. An $O(\log m)$ time bound improves on the $O(\log n)$ bound for skip graphs when m is much smaller than n. However, the cost of this improvement is in terms of losing support for complex queries and spatial locality, and the improvement itself is only a constant factor unless some machines store a superpolynomial number of resources.

4.3. THE DELETE OPERATION. The delete operation is very simple. When node u wants to leave the network, it deletes itself in parallel from all lists above level 0 and then deletes itself from level 0.

Algorithm 3. delete for existing node u

```
1 for ℓ ← 1 to u.maxLevel in parallel do
2 |  delete u from list at level ℓ
3 delete u from list at level 0
```

LEMMA 4. *In the absence of concurrency, the delete operation in a skip graph S with n nodes takes expected $O(\log n)$ messages and $O(1)$ time.*

PROOF. We have assumed that each linked-list delete operation takes $O(1)$ messages and $O(1)$ time starting from u; since all but one of these operations proceed in parallel, the total time is $O(1)$ while the messages total (summing over all $O(\log n)$ expected levels) at $O(\log n)$. □

The performance of a delete operation in the presence of concurrency depends on the costs of the underlying linked-list delete operation.

4.4. CORRECTNESS UNDER CONCURRENCY. In this section, we prove the correctness of the search, insert, and delete algorithms given in Section 4. We show in particular that both insert and delete maintain the following simple invariant, and then we use this invariant to argue that search operations eventually find their target node or correctly report that it is not present in the skip graph.

LEMMA 5. *At any time during the execution of any number of concurrent insert or delete operations, the set of nodes in any level-ℓ list for any $\ell > 0$ is a subset of the set of nodes in the level-0 list.*

PROOF. Immediate by assumption of atomicity for doubly-linked list operations and inspection of Algorithms 2 and 3. □

COROLLARY 6. *Consider a search operation with target v. If the search operation returns a node with key v, then some such node existed in the graph at some time during the execution of the search. If the search operation returns* ⟨`notFoundOp`⟩*, then there is a time during its execution at which no such node is present.*

PROOF. From Lemma 5, we have that v is present if and only if it appears in the level-0 list. In order to return ⟨`notFoundOp`⟩, the search algorithm must reach the bottom level without finding v; thus at the time when it queries v's level-0 predecessor and finds a successor $v' > v$ (or vice versa), v is not present in the graph.

If the search operation finds v, it must be because v sends a ⟨`foundOp`⟩ back to the source node. This can only occur if v has previously received ⟨`searchOp`⟩, which must have been sent by v's successor or predecessor in some list. At the time this message is sent, v is still an element of that list and thus present in the graph. □

Corollary 6 implies that any search operation can be linearized with respect to inserts and deletes. In effect, the skip graph inherits the atomicity properties of its bottom layer, with upper layers serving only to provide increased efficiency.

5. *Fault Tolerance*

In this section, we describe some of the fault-tolerance properties of a skip graph with alphabet {0, 1}. We are interested in the number of nodes that can be separated from the primary component by the failure of other nodes, as this determines the size of the surviving skip graph. This also allows comparison with the fault tolerance of related data structures, such as augmented versions of linked lists and binary trees, which have been well studied: Some results can be seen in Munro and Poblete [1984] and Aumann and Bender [1996].

Note that if multiple nodes are stored on a single machine, when that machine crashes, all of its nodes vanish simultaneously. Our results are stated in terms of the fraction of nodes that are lost; if the nodes are roughly balanced across machines, this will be proportional to the fraction of machine failures. Nonetheless, it would be useful to have a better understanding of fault tolerance when the mapping of resources to machines is taken into account; this may in fact dramatically improve fault tolerance, as nodes stored on surviving machines can always find other nodes stored on the same machine, and so need not be lost even if all of their neighbors in the skip graph are lost.

We consider two fault models: a random failure model in which an adversary chooses random nodes to fail, and a worst-case failure model in which an adversary chooses specific nodes to fail after observing the structure of the skip graph. For a random failure pattern, experimental results, presented in Section 5.1, show that for a reasonably large skip graph nearly all nodes remain in the primary component until about two-thirds of the nodes fail, and that it is possible to make searches highly resilient to failures, even without using the repair mechanism, by the use of redundant links. For a worst-case failure pattern, theoretical results, presented in Section 5.2, show that even a worst-case choice of failures causes limited damage. With high probability, a skip graph with n nodes has an $\Omega(\frac{1}{\log n})$ expansion ratio, implying that at most $O(f \cdot \log n)$ nodes can be separated from the primary

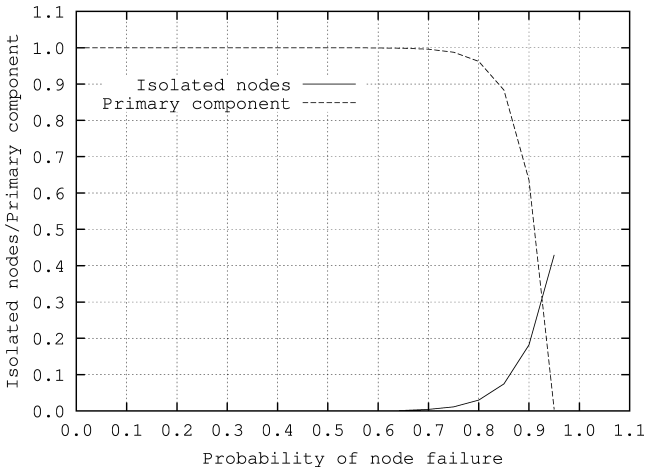

FIG. 3. The number of isolated nodes and the size of the primary component as a fraction of the surviving nodes in a skip graph with 131,072 nodes.

component by f failures. We do not give experimental results for adversarial failures, as experiments may not be able to identify the worst-case failure pattern.

The result of node failures will not be a skip graph, as many internal pointers will be replaced by dangling $\perp$ values. Restoring the skip graph structure will require a *repair mechanism*. Given current methods, the simplest and most efficient repair mechanism appears to be that of rebuilding the skip graph from scratch. We discuss this issue further in Section 5.3.

5.1. RANDOM FAILURES. In our simulations, skip graphs appear to be highly resilient to random failures. We constructed a skip graph of 131, 072 nodes, where each node had a unique label from [1, 131072]. We progressively increased the probability of node failure and measured the size of the largest connected component of the live nodes, as well as the number of isolated nodes as a fraction of the total number of nodes in the graph. As shown in Figure 3, nearly all nodes remain in the primary component, even as the probability of individual node failure exceeds 0.6. We also see that a lot of nodes are isolated as the failure probability increases because all of their immediate neighbors die.

For searches, the fact that the average search involves only $O(\log n)$ nodes establishes trivially that most searches succeed as long as the proportion of failed nodes is substantially less than $O(\frac{1}{\log n})$. By detecting failures locally and using additional redundant edges, we can make searches highly tolerant to small numbers of random faults.

Some further experimental results are shown in Figure 4. In these experiments, each node had additional links to up to five nearest successors at every level. A total of 10,000 messages were sent between randomly chosen source and destination nodes, and the fraction of failed searches was measured. We see that skip graphs are quite resilient to random failures. This plot appears to contradict the one shown in Figure 3, because we would expect all the searches to succeed as long as all live nodes are in the same connected component. However, once the source and target nodes are fixed, there is a fixed, deterministic path along which the search proceeds and if any node on this path fails, the search fails. So there may be *some*

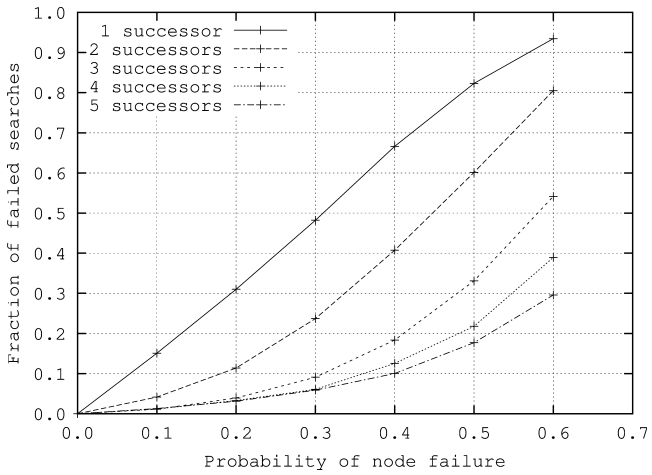

FIG. 4. Fraction of failed searches in a skip graph with 131,072 nodes and 10,000 messages. Each node has up to five successors at each level.

path between the source and destination nodes, putting them in the same connected component, but the path used by the search algorithm may be broken, foiling the search. This suggests that if we use smarter search techniques, such as jumping between the different skip lists to which a node belongs, we can get much better search performance even in the presence of failures.

In general, skip graphs do not provide as strong guarantees as those provided by data structures based on explicit use of expanders such as censorship-resistant networks [Fiat and Saia 2002; Saia et al. 2002; Datar 2002]. Nonetheless, we believe that this is compensated by the simplicity of skip graphs and the existence of good distributed mechanisms for constructing and repairing them.

5.2. ADVERSARIAL FAILURES. In addition to considering random failures, we are also interested in analyzing the performance of a skip graph when an adversary can observe the data structure and choose specific nodes to fail. Experimental results may not even be able to identify these worst-case failure patterns. So in this section, we look at the expansion ratio of a skip graph, as this gives us the number of nodes that can be separated from the primary component even with adversarial failures.

Let G be a graph. Recall that the expansion ratio of a set of nodes A in G is $|\delta A|/|A|$, where $|\delta A|$ is the number of nodes that are not in A but are adjacent to some node in A. The expansion ratio of the graph G is the minimum expansion ratio for any set A, for which $1 \leq |A| \leq n/2$. The expansion ratio determines the resilience of a graph in the presence of adversarial failures because separating a set A from the primary component requires all nodes in δA to fail. We will show that skip graphs have $\Omega(\frac{1}{\log n})$ expansion ratio with high probability, implying that only $O(f \cdot \log n)$ nodes can be separated by f failures, even if the failures are carefully targeted.

Our strategy for showing a lower bound on the expansion ratio of a skip graph will be to show that with high probability, all sets A either have large $\delta_0 A$ (i.e., many neighbors at the bottom level of the skip graph) or have large $\delta_\ell A$ for some particular ℓ chosen based on the size of A. Formally, we define $\delta_\ell A$ as the set of all nodes that are not in A but are joined to a node in A by an edge at level ℓ. Our

result is based on the observation that $\delta A = \bigcup_\ell \delta_\ell A$ and $|\delta A| \geq \max_\ell |\delta_\ell A|$. We begin by counting the number of sets A of a given size that have small $\delta_0 A$.

LEMMA 7. *In a n-node skip graph with alphabet $\{0, 1\}$, the number of sets A, where $|A| = m < n$ and $|\delta_0 A| < s$, is less than $\sum_{r=1}^{s-1} \binom{m+1}{r}\binom{n-m-1}{r-1}$.*

PROOF. Without loss of generality, assume that the nodes of the skip graph are numbered from 1 to n. Given a subset A of these nodes, define a corresponding bit-vector x by letting $x_i = 1$ if and only if node i is in A. Then $\delta_0 A$ corresponds to all zeroes in x that are adjacent to a one.

Consider the extended bit-vector $x' = 1x1$ obtained by appending a one to each end of x. Because x' starts and ends with a one, it can be divided into alternating intervals of ones and zeroes, of which $r+1$ intervals will consist of ones and r will consist of zeroes for some r, where $r > 0$ since x contains at least one zero. Observe that each interval of zeroes contributes at least one and at most two of its endpoints to $\delta_0 A$. It follows that $r \leq |\delta_0 A| \leq 2r$, and thus any A for which $|\delta_0 A| < s$ corresponds to an x for which x' contains $r \leq |\delta_0 A| < s$ intervals of zeroes.

Since there is at least one A with $r < s$ but $|\delta_0 A| \geq s$, the number of sets A with $|\delta_0 A| < s$ is strictly less than the number of sets A with $r < s$. By counting the latter quantity, we get a strict upper bound on the former.

We now count, for each r, the number of bit-vectors x' with $n - m$ zeroes consisting of $r + 1$ intervals of ones and r intervals of zeroes. Observe that we can characterize such a bit-vector completely by specifying the nonzero length of each of the $r+1$ all-one intervals, together with the nonzero length of each of the r all-zero intervals. There are $m+2$ ones that must be distributed among the $r+1$ all-one intervals, and there are $\binom{m+2-1}{r+1-1} = \binom{m+1}{r}$ ways to do so. Similarly, there are $n-m$ zeroes to distribute among the r all-zero intervals, and there are $\binom{n-m-1}{r-1}$ ways to do so. Since these two distributions are independent, the total count is exactly $\binom{m+1}{r}\binom{n-m-1}{r-1}$.

Summing over all $r < s$ then gives the upper bound $\sum_{r=1}^{s-1} \binom{m+1}{r}\binom{n-m-1}{r-1}$. □

For levels $\ell > 0$, we show with a probabilistic argument that $|\delta_\ell A|$ is only rarely small.

LEMMA 8. *Let A be a subset of $m \leq n/2$ nodes of an n-node skip graph S with alphabet $\{0, 1\}$. Then for any ℓ, $\Pr[|\delta_\ell A| \leq \frac{1}{3} \cdot 2^\ell] < 2\binom{2^\ell}{\lfloor \frac{2}{3} \cdot 2^\ell \rfloor}(2/3)^m$.*

PROOF. The key observation is that for each b in $\{0, 1\}^\ell$, if A contains a node u with $m(u) \restriction \ell = b$ and A's complement $S - A$ contains a node v with $m(v) \restriction \ell = b$, then there exist nodes $u' \in A$ and $v' \in S - A$ along the path from u to v in S_b, such that u' and v' are adjacent in S_b. Furthermore, since such pairs are distinct for distinct b, we get a lower bound on $\delta_\ell A$ by computing a lower bound on the number of distinct b's for which both A and $S - A$ contain at least one node in S_b.

Let $T(A)$ be the set of $b \in \{0, 1\}^\ell$ for which A contains a node of S_b, and similarly for $T(S - A)$. Then

$$\begin{aligned}\Pr\left[|T(A)| < \frac{2}{3} \cdot 2^\ell\right] &\leq \sum_{B \subset \{0,1\}^\ell, |B| = \lfloor \frac{2}{3} \cdot 2^\ell \rfloor} \Pr[T(A) \subseteq B] \\ &\leq \binom{2^\ell}{\lfloor \frac{2}{3} \cdot 2^\ell \rfloor}(2/3)^{|A|},\end{aligned}$$

and by the same reasoning,

$$\Pr\left[|T(S-A)| < \frac{2}{3}\cdot 2^\ell\right] \le \binom{2^\ell}{\lfloor \frac{2}{3}\cdot 2^\ell \rfloor}(2/3)^{|S-A|}.$$

But if both $T(A)$ and $T(S-A)$ hit at least two-thirds of the b, then their intersection must hit at least one-third. Thus, the probability that $|T(A)\cap T(S-A)| < \frac{1}{3}\cdot 2^\ell$ is at most $\binom{2^\ell}{\lfloor \frac{2}{3}\cdot 2^\ell \rfloor}((2/3)^{|A|}+(2/3)^{|S-A|})$, which is in turn bounded by $2\binom{2^\ell}{\lfloor \frac{2}{3}\cdot 2^\ell \rfloor}(2/3)^{|A|}$ under the assumption that $|A| \le |S-A|$. □

We can now get the full result by arguing that there are not enough sets A with small $|\delta_0 A|$ (Lemma 7) to get a nonnegligible probability that any one of them has small $|\delta_\ell A|$ for an appropriately chosen ℓ (Lemma 8). Details are given in the proof of Theorem 9 next.

THEOREM 9. *Let $c \ge 6$. Then a skip graph with n nodes and alphabet $\{0,1\}$, has an expansion ratio of at least $\frac{1}{c\log_{3/2} n}$ with probability at least $1-\alpha n^{5-c}$, where the constant factor α does not depend on c.*

PROOF. We will show that the probability that a skip graph S with n nodes does *not* have the given expansion ratio is at most αn^{5-c}, where $\alpha = 31$. The particular value of $\alpha = 31$ may be an artifact of our proof; the actual constant may be smaller.

Consider some subset A of S with $|A| = m \le n/2$. Let $s = \frac{m}{c\log_{3/2} n} = \frac{m\lg(3/2)}{c\lg n}$, and let $s_1 = \lceil s \rceil$. We wish to show that with high probability all A of size m have $|\delta A| \ge s_1$. We will do so by counting the expected number of sets A with smaller expansion. Any such set must have both $|\delta_0 A| < s$ and $|\delta_\ell A| < s$ for any ℓ; our strategy will be to show first that there are few sets A in the first category, and that each set that does have small $\delta_0 A$ is very likely to have large $\delta_\ell A$ for a suitable ℓ.

By Lemma 7, there are at most $\sum_{r=1}^{s_1-1}\binom{m+1}{r}\binom{n-m-1}{r-1}$ sets A of size m for which $|\delta_0 A| < s_1$. It is not hard to show that the largest term in this sum dominates. Indeed, for $r < s_1$, the ratio between adjacent terms $\binom{m+1}{r}\binom{n-m-1}{r-1}/\binom{m+1}{r+1}\binom{n-m-1}{r}$ equals $\frac{r}{m+1-r}\cdot\frac{r+1}{n-m-r}$, and since $r < \frac{1}{6}m$ and $m \le \frac{n}{2}$, this product is easily seen to be less than $\frac{1}{2}$. It follows that

$$\begin{aligned}\sum_{r=1}^{s_1-1}\binom{m+1}{r}\binom{n-m-1}{r-1} &< \binom{m+1}{s_1-1}\binom{n-m-1}{s_1-2}\sum_{i=0}^{\infty}2^{-i}\\ &= 2\binom{m+1}{s_1-1}\binom{n-m-1}{s_1-2}\\ &< n^{2(s_1-1)}\\ &< n^{2s} = n^{\frac{2m}{c\log_{3/2} n}} = 2^{\frac{2m\lg(3/2)}{c}}.\end{aligned}$$

Now let $\ell = \lceil \lg s + \lg 3 \rceil$, so that $s \le \frac{1}{3}2^\ell \le 2s$.

Applying Lemma 8, we have

$$\begin{aligned}\Pr[|\delta_\ell A| < s|] &\le \Pr\left[|\delta_\ell A| \le \frac{1}{3}\cdot 2^\ell\right]\\ &< 2\binom{2^\ell}{\lfloor \frac{2}{3}\cdot 2^\ell \rfloor}(2/3)^m\end{aligned}$$

$$\begin{aligned} &\leq 2\binom{6s}{2s+1}(2/3)^m \\ &< 2\cdot(6s)^{2s+1}(2/3)^m, \end{aligned}$$

and thus

$$\sum_{A\subset S,|A|=m,|\delta_0 A|<s} \Pr\left[|\delta_\ell A| < s|\right] < 2^{\frac{2m\lg(3/2)}{c}}\cdot 2\cdot(6s)^{2s+1}(2/3)^m. \tag{1}$$

Taking the base-2 logarithm of the righthand side gives

$$\begin{aligned} &\frac{2m\lg(3/2)}{c} + 1 + (2s+1)\lg(6s) - m\lg(3/2) \\ &< \frac{2m\lg(3/2)}{c} + 1 + \left(\frac{3m\lg(3/2)}{c\lg n}\right)\lg n - m\lg(3/2) \\ &= 1 + m\left(\frac{5}{c} - 1\right)\lg(3/2). \end{aligned}$$

It follows that the probability that there exists an A of size m, for which both $|\delta_0 A|$ and $|\delta_\ell A|$ are less than s, is at most 2 to the preceding quantity, which we can write as $2b^m$, where $b = (3/2)^{(5/c-1)}$.

To compute the probability that any set has a small neighborhood, we sum over m. By definition of the expansion ratio, we need only consider values of m that are less than or equal to $n/2$; however, because every proper subset A of S has at least one neighbor, we need to consider only $m > c\log_{3/2} n$. So we have

$$\begin{aligned} \Pr\left[S \text{ has expansion ratio less than } \frac{1}{c\log_{3/2} n}\right] &< \sum_{m=\lceil c\log_{3/2} n\rceil}^{n/2} 2b^m \\ &< 2\left(\sum_{i=0}^{\infty} b^i\right)\left(b^{\lceil c\log_{3/2} n\rceil}\right) \\ &\leq \frac{2}{1-b}\cdot b^{c\log_{3/2} n} \\ &= \frac{2}{1-b}\cdot n^{c\log_{3/2} b} \\ &= \frac{2}{1-b}\cdot n^{c\left(\frac{5}{c}-1\right)} \\ &= \frac{2}{1-b}\cdot n^{5-c} \\ &< 31\cdot n^{5-c}, \end{aligned}$$

where the last inequality follows from the assumption that $c \geq 6$. □

5.3. REPAIRING A SKIP GRAPH. We have shown in preceding sections that such crashes are unlikely to disconnect the skip graph. However, this leaves the short-term problem of performing searches in a damaged skip graph and the long-term problem of regenerating the missing links. We would like to have a

repair mechanism that reconstitutes a damaged skip graph from its surviving fragments.

In a previous version of the present work [Aspnes and Shah 2003], we described an elaborate mechanism for repairing a skip graph concurrently with insertions and deletions. This repair mechanism left much to be desired; its worst-case running time was linear in size of the skip graph, and under certain patterns of operations it could permanently disconnect nodes from the graph or fail to converge. Fortunately, better solutions are possible.

The simplest is a generational approach in which the skip graph is periodically rebuilt from scratch, with all nodes migrating to the new skip graph once it has been reconstructed. This can be done either by having some initiator node recruit the remaining nodes using the standard insertion algorithm of Section 4.2, or by some faster parallel mechanism along the lines of Angluin et al. [2005]. The method of Angluin et al. [2005] can, in principle, reconstruct a skip graph from an arbitrary surviving fragment in $O(\log^2 n)$ time, under certain plausible assumptions about key length.

6. *Congestion*

In addition to fault tolerance, a skip graph provides a limited form of congestion control by smoothing out hot spots caused by popular search targets. The guarantees that a skip graph makes in this case are similar to those made for survivability. Just as a node's continued connectivity depends on the survival of its neighbors, its message load depends on the popularity of its neighbors as search targets. However, we can show that this effect drops off rapidly with distance; nodes that are far away from a popular target in the bottom-level list of a skip graph get little increased message load on average.

We give two versions of this result. The first version, given in Section 6.1, shows that the probability that a particular search uses a node between the source and target drops off inversely with the distance from node to target. This fact is not necessarily reassuring to heavily-loaded nodes. Since the probability averages over all choices of membership vectors, it may be that some particularly unlucky node finds itself with a membership vector that puts it on nearly every search path to some very popular target. The second version, given in Section 6.2, shows that our average-case bounds hold with high probability. While it is still possible that a spectacularly unlucky node is hit by most searches, such a situation only occurs for very low-probability choices of membership vectors. It follows that most skip graphs alleviate congestion well. For our results, we consider skip graphs with alphabet $\{0, 1\}$.

6.1. Average Congestion for a Single Search.

Our argument that the average congestion is inversely proportional to distance is based on the observation that a node only appears on a search path in a skip list S if it is among the tallest nodes between itself and the target. We will need a small technical lemma that counts the expected number of such tallest nodes. Consider a set-valued Markov process $A_0 \supseteq A_1 \supseteq A_2 \ldots$, where A_0 is some nonempty initial set and each element of A_t appears in A_{t+1} with independent probability $\frac{1}{2}$. Let τ be the largest index for which A_τ is not empty. We will now show that $\mathrm{E}[|A_\tau|]$ is small, regardless of the size of the initial set A_0.

LEMMA 10. *Let $A_0, A_1, \ldots, A_\tau$ be defined as earlier. Then* $\mathrm{E}[|A_\tau|] < 2$.

PROOF. The bound on $\mathrm{E}[|A_\tau|]$ will follow from a surprising connection between $\mathrm{E}[|A_\tau|]$ and $\Pr[|A_\tau| = 1]$. We begin by obtaining a recurrence for $\Pr[|A_\tau| = 1]$.

Let $P(n) = \Pr[|A_\tau| = 1 : |A_0| = n]$. Clearly $P(0) = 0$ and $P(1) = 1 \geq \frac{2}{3}$. For larger n, summing over all $k = |A_1|$ gives $P(n) = 2^{-n} \sum_{k=0}^{n} \binom{n}{k} P(k)$, which can be rewritten as $P(n) = \frac{1}{2^n - 1} \sum_{k=1}^{n-1} \binom{n}{k} P(k)$. The solution to this recurrence goes asymptotically to $\frac{1}{2 \ln 2} \pm 10^{-5} \approx 0.7213\ldots$ [Sedgewick and Flajolet 1996, Theorem 7.9]; however, we will use the much simpler property that when $n \geq 2$, $P(n) = \Pr[|A_\tau| = 1]$ is the probability of an event that does not always occur, and is thus less than 1.

Let $E(n) = \mathrm{E}[|A_\tau| : |A_0| = n]$. Then $E(n) = 2^{-n}(n + \sum_{k=1}^{n} \binom{n}{k} E(k))$, and eliminating the $E(n)$ term on the righthand side gives $E(n) = \frac{1}{2^n - 1}(n + \sum_{k=1}^{n-1} \binom{n}{k} E(k))$.

For $n = 1$, $E(1) = 1$ by definition. Recall that $P(1)$ is also equal to 1. We will now show that $E(n) = 2P(n)$ for all $n > 1$. Suppose that $E(k) = 2P(k)$ for $1 < k < n$. Let $n = 2$, then $E(k) = 2P(k)$ for all $1 < k < n$ (an empty set of k). Then,

$$\begin{aligned}
E(n) &= \frac{1}{2^n - 1}\left(n + \sum_{k=1}^{n-1} \binom{n}{k} E(k)\right) \\
&= \frac{1}{2^n - 1}\left(n + \binom{n}{1} E(1) + 2\sum_{k=2}^{n-1} \binom{n}{k} P(k)\right) \\
&= \frac{1}{2^n - 1}\left(2\binom{n}{1} P(1) + 2\sum_{k=2}^{n-1} \binom{n}{k} P(k)\right) \\
&= 2 \cdot \frac{1}{2^n - 1} \sum_{k=1}^{n-1} \binom{n}{k} P(k) \\
&= 2P(n).
\end{aligned}$$

Since $P(n) < 1$ for $n > 1$, we immediately get $E(n) < 2$ for all n. For large n this is an overestimate: Given the asymptotic behavior of $P(n)$, $E(n)$ approaches $\frac{1}{\ln 2} \pm 2 \times 10^{-5} \approx 1.4427\ldots$. However, it is close enough for our purposes. □

THEOREM 11. *Let S be a skip graph with alphabet $\{0, 1\}$, and consider a search from s to t in S. Let u be a node with $s < u < t$ in the key ordering (the case $s > u > t$ is symmetric), and let d be the distance from u to t, defined as the number of nodes v with $u < v \leq t$. Then the probability that a search from s to t passes through u is less than $\frac{2}{d+1}$.*

PROOF. Let $S_{m(s)}$ be the skip-list restriction of s whose existence is shown by Lemma 1. From Lemma 2, we know that searches in S follow searches in $S_{m(s)}$. Observe that for u to appear in the search path from s to t in $S_{m(s)}$, there must be no node v with $u < v \leq t$ whose height in $S_{m(s)}$ is higher than u's. It follows that u can appear in the search path only if it is among the tallest nodes in the interval $[u, t]$, namely, if $|m(s) \wedge m(u)| \geq |m(s) \wedge m(v)|$ for all v with $u < v \leq$

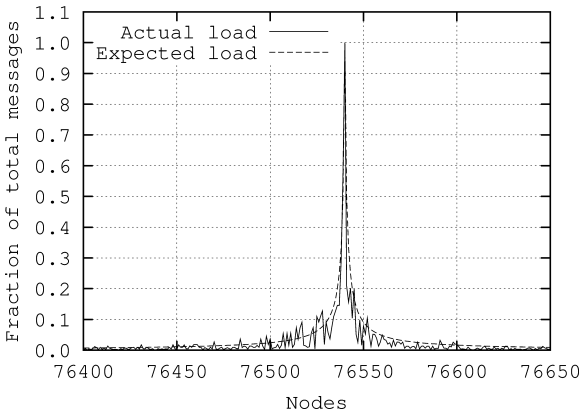

FIG. 5. Actual and expected congestion in a skip graph with 131,072 nodes with the target = 76,539. Messages were delivered from each node to the target and the actual number of messages through each node was measured. The bound on the expected congestion is computed using Theorem 11. Note that this bound may overestimate the actual expected congestion.

t. Recall that $m(s) \wedge m(v)$ is the common prefix (possibly empty) of $m(s)$ and $m(v)$.

There are $d+1$ nodes in this interval. By symmetry, if there are k tallest nodes, then the probability that u is among them is $\frac{k}{d+1}$. Let T be the random variable representing the set of tallest nodes in the interval. Then

$$\begin{aligned}\Pr[u \in T] &= \sum_{k=1}^{d+1} \Pr[|T| = k] \frac{k}{d+1} \\ &= \frac{\mathrm{E}[|T|]}{d+1}.\end{aligned}$$

What is the expected size of T? All $d+1$ nodes have height at least 0, and in general each node with height at least k has height at least $k+1$ with independent probability $\frac{1}{2}$. The set T consists of the nodes that are left at the last level before all nodes vanish. It is thus equal to A_τ in the process defined in Lemma 10, and we have $\mathrm{E}[|T|] < 2$ and thus $\Pr[u \in T] < \frac{2}{d+1}$. □

For comparison, experimental data for the congestion in a skip graph with 131, 072 nodes, together with the theoretical average predicted by Theorem 11, is shown in Figure 5.

6.2. DISTRIBUTION OF THE AVERAGE CONGESTION. Theorem 11 is of small consolation to some node that draws a probability $\frac{2}{d+1}$ straw and participates in every search. Fortunately, such disasters do not happen often. Define the *average congestion* L_{tu} imposed by a search for t on a node u as the probability that an $s - t$ search hits u conditioned on the membership vectors of all nodes in the interval $[u, t]$, where $s < u < t$, or, equivalently, $s > u > t$.[3] Note that since the

[3] It is immediate from the proof of Theorem 11 that L_{tu} does not depend on the choice of s.

conditioning does not include the membership vector of s, the definition in effect assumes that $m(s)$ is chosen randomly. This approximates the situation in a fixed skip graph where a particular target t is used for many searches that may hit u, but the sources of these searches are chosen randomly from the other nodes in the graph.

Theorem 11 implies that the expected value of L_{tu} is no more than $\frac{2}{d+1}$. In the following theorem, we show that the distribution of L_{tu} declines exponentially beyond this point.

THEOREM 12. *Let S be a skip graph with alphabet $\{0, 1\}$. Fix nodes t and u, where $u < t$ and $|\{v : u < v \leq t\}| = d$. Then for any integer $\ell \geq 0$, $\Pr[L_{tu} > 2^{-\ell}] \leq 2e^{-2^{-\ell}d}$.*

PROOF. Let $V = \{v : u < v \leq t\}$ and let $m(V) = \{m(v) : v \in V\}$. As in the proof of Theorem 11, we will use the fact that u is on the path from s to t if and only if u's height in $S_{m(s)}$ is not exceeded by the height of any node v in V.

To simplify notation, let us assume without loss of generality that $m(u)$ is the all-zero vector. Then the height of u in $S_{m(s)}$ is equal to the length of the initial prefix of zeroes in $m(s)$, and u has height at least ℓ with probability $2^{-\ell}$. Whether this is enough to raise it to the level of the tallest nodes in V will depend on what membership vectors appear in $m(V)$.

Let $m(s) = 0^i 1 \ldots$. Then u has height exactly i, and is hit by an $s - t$ search unless there is some $v \in V$ with $m(v) = 0^i 1 \ldots$. We will argue that when $d = |V|$ is sufficiently large, then there is high probability that all initial prefixes $0^i 1$ appear in $m(V)$ for $i < \ell$. In this case, u can only appear in the $s - t$ path if its height is at least ℓ, which occurs with probability only $2^{-\ell}$. So if $0^i 1$ appears as a prefix of some $m(v)$ for all $i < \ell$, then $L_{tu} \leq 2^{-\ell}$. Conversely, if $L_{tu} > 2^{-\ell}$, then $0^i 1$ does *not* appear as a prefix of some $m(v)$ for some $i < \ell$.

Now let us calculate the probability that not all such prefixes appear in $m(V)$. We are going to show that this probability is at most $2e^{-2^{-\ell}d}$, and so we need only consider the case where $e^{-2^{-\ell}d} \leq \frac{1}{2}$; this bound is used in steps (2) and (3) given next. We have

$$\begin{aligned}
\Pr[L_{tu} > 2^{-\ell}] &\leq \Pr[\neg(\forall i < \ell : \exists v \in V : 0^i 1 \preceq m(v))] \\
&= \Pr[\exists i < \ell : \forall v \in V : 0^i 1 \not\preceq m(v)] \\
&\leq \sum_{i=0}^{\ell-1} \Pr[\forall v \in V : 0^i 1 \not\preceq m(v)] \\
&= \sum_{i=0}^{\ell-1} (1 - 2^{-i-1})^d \\
&\leq \sum_{i=0}^{\ell-1} e^{-2^{-i-1}d} \\
&= \sum_{j=0}^{\ell-1} e^{-2^{-\ell+j}d}
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{j=0}^{\ell-1} (e^{-2^{-\ell}d})^{2^j} \\
&\leq \sum_{j=0}^{\ell-1} (e^{-2^{-\ell}d})^{j+1} \qquad (2) \\
&\leq \sum_{j=0}^{\infty} (e^{-2^{-\ell}d})^{j+1} \\
&= \frac{e^{-2^{-\ell}d}}{1 - e^{-2^{-\ell}d}} \\
&\leq 2e^{-2^{-\ell}d}. \qquad (3)
\end{aligned}
$$

□

7. *Related Work*

SkipNet is a system very similar to skip graphs that was independently developed by Harvey et al. [2003]. SkipNet builds a trie of circular, singly-linked skip lists to link the *machines* in the system. Machine names are sorted using the domain in which they are located (e.g., `www.yale.edu`). In addition to the pointers between all machines in all domains that are structured like a skip graph, within each individual domain, the machines are also linked using a DHT, and the resources are uniformly distributed over all machines using hashing. A search consists of two stages: First, the search locates the domain in which a resource lies by using a search operation similar to a skip graph. Second, once the search reaches some machine inside a particular domain, it uses greedy routing as in DHTs to locate the resource within that domain. SkipNet has been successfully implemented, and this shows that a skip-graph-like structure can be used to build real systems.

The SkipNet design ensures *path locality*, that is, the traffic within a domain traverses other nodes only within the same domain. Further, each domain gets to host it own data, which provides *content locality* and inherent security. Finally, using the hybrid storage and search scheme provides *constrained load balancing* within a given domain. However, as the name of the data item includes the domain in which it is located, transparent remapping of resources to other domains is not possible, thus giving a very limited form of load balancing. Another drawback of this design is that it does not give full-fledged spatial locality. For example, if the resources are document files, sorting according to the domain on which they are served gives no advantage in searching for related files, as compared to DHTs.

Zhang et al. [2003] and Awerbuch and Scheideler [2003] have both independently suggested designs for peer-to-peer systems using separate data structures for resources and the machines that store them. The main idea is to build a data structure D over the resources, which are distributed uniformly among all the machines using hashing, and to build a separate DHT over all machines in the system. Each resource maintains the keys of its neighboring resources in D, and each machine maintains the addresses of its neighboring machines as per the DHT network.

To access a neighbor b of resource a, a initiates a DHT search for the hash value of b. One pointer access in D is converted to a search operation in the DHT, so if any operation in D takes time t, the same operation takes $O(t \log m)$ time with m machines in this hybrid system. Zhang et al. [2003] focus on implementing a tree of the resources, in which each node in the tree is responsible for some fixed range of the keyspace for which its parent is responsible. Awerbuch and Scheideler [2003] propose building a skip graph of the resources on top of the machines in the DHT.

This design approach is interesting because it allows building any data structure using the resources, while providing uniform load balancing. In particular, both these systems support complex queries as in skip graphs, and uniform load balancing as in DHTs. We believe that distributing resources uniformly among all nodes (as described in Section 3) will also have the same properties as these two approaches.

However, the Awerbuch and Scheideler approach and our uniform resource distribution method suffer from the same problems of high storage requirements and high volume of repair-mechanism message traffic as a skip graph. With m machines and n resources in the system, in the Awerbuch and Scheideler technique, each machine has to store $O(\log m)$ pointers (for the DHT links) and $O(\log n)$ keys for *each* resource that it hosts (for the skip-graph pointers). Further, the repair mechanism has to repair the broken DHT links as well as inconsistent skip-graph keys. Finally, the search performance is degraded to $O(\log^2 m)$ compared to $O(\log m)$ in DHTs and $O(\log n)$ in skip graphs. In comparison, in our approach of uniform resource distribution, each machine has to store $O(\log n)$ pointers for each resource that it hosts, repair is required only for the skip-graph links, and the search time is $O(\log n)$, as in skip graphs.

In the Zhang et al. approach, each machine has to store k keys for the k children of each tree node that it hosts, and $O(\log m)$ pointers for the DHT links. Repair involves fixing broken tree keys as well as broken DHT links. This scheme suffers from other problems of tree data structures, such as increased traffic on nodes higher up in the tree, and vulnerability to failures of these nodes. Further, unlike skip graphs, it requires a priori knowledge about the keyspace in order to assign specific ranges to the tree nodes. Thus, it is an open problem to design a system that efficiently supports both uniform load balancing and complex queries.

8. *Conclusion*

We have defined a new data structure for distributed data stores, namely, the skip graph, that has several desirable properties. Constructing, inserting new nodes, and searching in a skip graph can be done using simple and straightforward algorithms. Skip graphs are highly resilient, tolerating a large fraction of failed nodes without losing connectivity. Using the repair mechanism, disruptions to the data structure can be repaired in the absence of additional faults. Skip graphs also support range queries, which allows, for example, searching for a copy of a resource near a particular location by using the location as low-order field in the key, as well as the clustering of nodes with similar keys.

As explained in Section 7, one issue that remains to be addressed is the large number of pointers per machine in the system. It would be interesting to design

a peer-to-peer system that maintains fewer pointers per machine and yet supports spatial locality. Also, skip graphs do not exploit network locality (such as latency along transmission paths) in locating resources and it would be interesting to study the performance benefits in that direction, perhaps by using multidimensional skip graphs. As with other overlay networks, it would be interesting to see how the network performs in the presence of Byzantine failures. Finally, it would be useful to develop a self-stabilizing repair mechanism for defective skip graphs.

ACKNOWLEDGMENTS: We would like to thank Arvind Krishnamurthy for helpful discussions.

REFERENCES

ABRAHAM, I., ASPNES, J., AND YUAN, J. 2005. Skip B-trees. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, 284–295.

ANGLUIN, D., ASPNES, J., CHEN, J., WU, Y., AND YIN, Y. 2005. Fast construction of overlay networks. In *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 145–154.

ASPNES, J., DIAMADI, Z., AND SHAH, G. 2002. Fault-Tolerant routing in peer-to-peer systems. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC)*, Monterey, CA, 223–232. `http://arXiv.org/abs/cs/0302022`.

ASPNES, J., AND SHAH, G. 2003. Skip graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 384–393.

ASPNES, J., AND WIEDER, U. 2005. The expansion and mixing time of skip graphs with applications. In *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 126–134.

AUMANN, Y., AND BENDER, M. A. 1996. Fault tolerant data structures. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS)*, Burlington, VT, 580–589.

AWERBUCH, B., AND SCHEIDELER, C. 2003. Peer-to-Peer systems for prefix search. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC)*, Boston. to appear.

CASTRO, M., DRUSCHEL, P., HU, Y. C., AND ROWSTRON, A. 2002. Topology-Aware routing in structured peer-to-peer overlay networks. In *Proceedings of the International Workshop on Future Directions in Distributed Computing (FuDiCo)*, Bertinoro, Italy, A. Schiper, et al., eds. Lecture Notes in Computer Science, vol. 2584. Springer, 103–107.

CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. 2000. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, H. Federrath, ed. Lecture Notes in Computer Science, vol. 2009. Springer, 46–66. `http://www.freenet.sourceforge.net`.

DATAR, M. 2002. Butterflies and peer-to-peer networks. In *Proceedings of the 10th European Symposium on Algorithms (ESA)*, Rome, R. Möhring and R. Raman, eds. Lecture Notes in Computer Science, vol. 2461. Springer, 310–322.

DE LA BRIANDAIS, R. 1959. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference*, vol. 15, Montvale, NJ, 295–298.

DEVROYE, L. 1992. A limit theory for random skip lists. *Ann. Appl. Probab. 2,* 3, 597–609.

FIAT, A., AND SAIA, J. 2002. Censorship resistant peer-to-peer content addressable networks. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, San Francisco, CA, 94–103. Submitted to a special issue of *J. Alg.* dedicated to select papers of SODA 2002.

FREDKIN, E. 1960. Trie memory. *Commun. ACM 3,* 9 (Sept.), 490–499.

GABARRÓ, J., MARTÍNEZ, C., AND MESSEGUER, X. 1996. A top-down design of a parallel dictionary using skip lists. *Theor. Comput. Sci. 158,* 1–2 (May), 1–33.

GABARRÓ, J., AND MESSEGUER, X. 1997. A unified approach to concurrent and parallel algorithms on balanced data structures. In *Proceedings of the 17th International Conference of the Chilean Computer Society (SCCC)*, Valparáiso, Chile, 78–92.

GNUTELLA. 2007. Gnutella website. `http://gnutella.wego.com`.

HARREN, M., HELLERSTEIN, J. M., HUEBSCH, R., LOO, B. T., SHENKER, S., AND STOICA, I. 2002. Complex queries in DHT-based peer-to-peer networks. In *Proceedings of the 1st International Workshop on*

Peer-to-Peer Systems (IPTPS), Cambridge, MA, P. Druschel, et al., eds. Lecture Notes in Computer Science, vol. 2429, Springer, 242–250.

HARVEY, N. J. A., JONES, M. B., SAROIU, S., THEIMER, M., AND WOLMAN, A. 2003. SkipNet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, Seattle, WA, 113–126.

HARVEY, N. J. A., AND ZATLOUKAL, K. C. 2004. Family trees. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, New Orleans, LA, 308–317.

HILDRUM, K., KUBIATOWICZ, J. D., RAO, S., AND ZHAO, B. Y. 2002. Distributed object location in a dynamic network. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Winnipeg, Manitoba, Canada, 41–52.

JOHNSON, T., AND COLBROOK, A. 1994. A distributed, replicated, data-balanced search structure. *Int. J. High Speed Comput. 6,* 4 (Dec.), 475–500.

KARGER, D., AND RUHL, M. 2002. Finding nearest neighbors in growth-restricted metrics. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*, Montreal, Canada, 741–750.

KIRSCHENHOFER, P., MARTÍNEZ, C., AND PRODINGER, H. 1995. Analysis of an optimized search algorithm for skip lists. *Theor. Comput. Sci. 144,* 1–2 (Jun.), 199–220.

KIRSCHENHOFER, P., AND PRODINGER, H. 1994. The path length of random skip lists. *Acta Informatica 31,* 8, 775–792.

KNUTH, D. E. 1973. *The Art of Computer Programming: Sorting and Searching*, vol. 3. Addison-Wesley, Reading, MA.

LI, X., MISRA, J., AND PLAXTON, C. G. 2004. Active and concurrent topology maintenance. In *Proceedings of the 18th International Conference on Distributed Computing (DISC)*, Amsterdam, The Netherlands, R. Guerraoui, ed. Lecture Notes in Computer Science, vol. 3274. Springer, 320–334.

MALKHI, D., NAOR, M., AND RATAJCZAK, D. 2002. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC)*, Monterey, CA, 183–192.

MUNRO, J. I., AND POBLETE, P. V. 1984. Fault tolerance and storage reduction in binary search trees. *Inf. Control 62,* 2/3 (Aug.), 210–218.

NAPSTER. 2003. Napster website. `http://www.napster.com`.

PAPADAKIS, T., MUNRO, J. I., AND POBLETE, P. V. 1990. Analysis of the expected search cost in skip lists. In *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, Bergen, Norway, J. R. Gilbert and R. G. Karlsson, eds. Lecture Notes in Computer Science, vol. 447, Springer, 160–172.

PLAXTON, C. G., RAJARAMAN, R., AND RICHA., A. W. 1999. Accessing nearby copies of replicated objects in a distributed environment. *Theory Comput. Syst. 32,* 3, 241–280.

PUGH, W. 1990. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM 33,* 6 (Jun.), 668–676.

RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. 2001. A scalable content-addressable network. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols (SIGCOMM)*, San Diego, CA, 161–172.

ROWSTRON, A., AND DRUSCHEL, P. 2001. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, 329–350.

SAIA, J., FIAT, A., GRIBBLE, S., KARLIN, A., AND SAROIU, S. 2002. Dynamically fault-tolerant content addressable networks. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA. P. Druschel, et al., eds. Lecture Notes in Computer Science, vol. 2429, Springer, 270–279.

SEDGEWICK, R., AND FLAJOLET, P. 1996. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, Reading, MA.

SHAKER, A., AND REEVES, D. S. 2005. Self-Stabilizing structured ring topology P2P systems. In *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing (P2P)*, Konstanz, Germany. IEEE Computer Society, 39–46.

SILAGHI, B., BHATTACHARJEE, B., AND KELEHER, P. 2002. Query Routing in the TerraDir distributed directory. In *Proceedings of the SPIE ITCOM*, vol. 4868, V. Firoiu and Z.-L. Zhang, eds. SPIE, Boston, 299–309.

STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. 2003. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE/ACM Trans. Netw. 11*, 17–32.

ZHANG, Z., SHI, S., AND ZHU, J. 2003. SOMO: Self-Organized metadata overlay for resource management in P2P DHT. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA.

ZHAO, B. Y., JOSEPH, A. D., AND KUBIATOWICZ, J. D. 2002. Locality-Aware mechanisms for large-scale networks. In *Proceedings of the Workshop on Future Directions in Distributed Computing (FuDiCo)*, Bertinoro, Italy, 80–83.

ZHAO, B. Y., KUBIATOWICZ, J., AND JOSEPH, A. D. 2001. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, University of California, Berkeley, Berkeley, CA.

RECEIVED JUNE 2003; ACCEPTED JUNE 2006