Skip Graphs

James Aspnes* Gauri Shah†

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 elements are stored in separate nodes that may fail at any time. They are designed for use in searching peer-to-peer networks, 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, constructing, inserting new elements into, searching a skip graph and detecting and repairing errors in the data structure introduced by node failures can be done using simple and straightforward algorithms.

1 Introduction

Peer-to-peer networks are distributed systems without any central authority that 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 peer-to-peer research yields a slew of desirable features for a peer-to-peer network 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 systems, such as Napster [NAP], Gnutella [GNU] and Freenet [FRE], did not support most of these features and were 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 systems like CAN [RFH+01], Chord [SMK+01], Pastry [RD01], Tapestry [JKZ01] and Viceroy [MNR02] use a **distributed hash table** (DHT) approach to overcome scalability problems. To ensure scalability, they hash the key of a resource to determine which node it will be stored at and balance out the load on the nodes in the network. The main operation in these systems is to retrieve the identity of the node which stores the resource, from any other node in the system. To this end, there is an overlay graph in which the location of the 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 systems by using different routing algorithms. Pastry and Tapestry uses Plaxton's algorithm [PRR97], 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 m bits placed around a *modulo* 2^m *identifier circle* and does greedy routing to the farthest possible node stored in the routing table. Most of these systems use $O(\log n)$ space and time for routing and $O(\log^2 n)$ time for node insertion. Because hashing destroys the ordering on keys, DHT systems do not support queries that seek near matches to a key or keys within a given range.

Some of these systems try to optimize performance by taking locality into account. Pastry [RD01, CDHR02] and Tapestry [JKZ01, ZJK02] exploit geographical proximity by choosing the physically closest node out of all the possible nodes with an appropriate identifier prefix. In CAN [RFH+01], each node measures its round-trip delay to a set of landmark nodes and accordingly places itself in the co-ordinate space to facilitate routing with respect to network proximity. This last method is not fully self-organizing

*Department of Computer Science, Yale University, New Haven, CT 06520-8285, USA. Email: aspnes@cs.yale.edu. Supported by NSF grants CCR-9820888 and CCR-0098078.

†Department of Computer Science, Yale University, New Haven, CT 06520-8285, USA. Email: shah@cs.yale.edu. Supported by NSF grants CCR-9820888 and CCR-0098078.

and may cause imbalance in the distribution of nodes leading to hotspots. Some methods to solve the nearest neighbor problem for overlay networks can be seen in [HKRZ02] and [KR02].

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 [FS02] present a system which is resilient to adversarial deletion of a constant fraction of the nodes; some extensions of this result can be seen in [Dat02]. However, they do not give efficient methods to dynamically maintain such a system.

TerraDir [SBK02] is a recent system that provides locality and maintains a hierarchical data structure using caching and replication. There are as 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 the near-uniform distribution of the output of the hash function. So the costs of constructing, maintaining, and searching these data structures is closer to the $\Theta(\log n)$ costs of tree operations than 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 web pages, enhanced browsing and efficient searching. Skip graphs also support **complex queries** such as range queries, i.e. locating resources whose keys lie within a certain specified range. There has been some interest in supporting complex queries in peer-to-peer-systems [HHH+02], and designing a system that supports range queries has been posed as an open question. 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 $O(1/\log n)$ 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.

The rest of the paper is organized as follows: we describe skip graphs and algorithms for them in detail in Section 2. Sections 3 and 4 describe the repair mechanism and fault-tolerance properties for a skip graph. Contention analysis and load balancing results are described in Section 5. Finally, we conclude in Section 6.

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**, i.e., there is a fixed upper bound (time-out) on the transmission delay of a message. Processes can **crash**, i.e., halt prematurely, and crashes are permanent. We use the term *node* to represent a process that is running on a particular machine. We assume that each message takes at most unit time to be delivered and any internal processing at a machine takes no time.

2 Skip graphs

A **skip list** [Pug90] 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. The 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 level. 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 $O\left(\log n \frac{1}{(1-p)\log\frac{1}{p}}\right)$. Skip lists have been extensively studied [Pug90, PMP90, Dev92, KP94, KMP95] and because they require no global balancing operations are particularly useful in parallel systems [GMM93, GMM96, GM97].

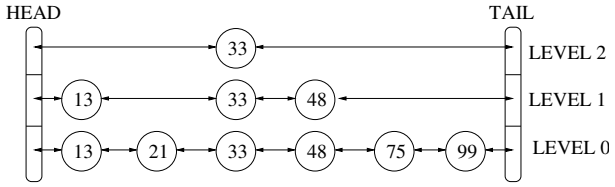


Figure 1: A skip list.

We would like to use a data structure similar to a skip list to support typical binary tree operations on a sequence whose elements 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 contention. 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 their fellows 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 node 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 in Lemma 2.1 that algorithms for skip lists can be applied directly to skip graphs, as a skip graph is equivalent to a collection of up to 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 no node is isolated. In Section 4 we observe that skip graphs are resilient to node failures and have an expansion ratio of $\Omega(1/\log n)$ with n nodes in the graph.

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 whose distribution satisfies certain symmetry properties [ADS02]. While this lower bound requires some independence assumptions that are not satisfied by skip graphs, there is enough similarity between skip graphs and the class of models considered in [ADS02] that an $\Omega(\log n)$ average degree is not surprising.

We now give a formal definition of a skip graph. Precisely which lists an element x belongs to 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 $m(x)$ needs to be generated on average. The idea of the membership vector is that every doubly-linked list in the skip graph is labeled by some finite word w, and an element x is in the list labeled by w if and only if w is a prefix of $m(x)$.

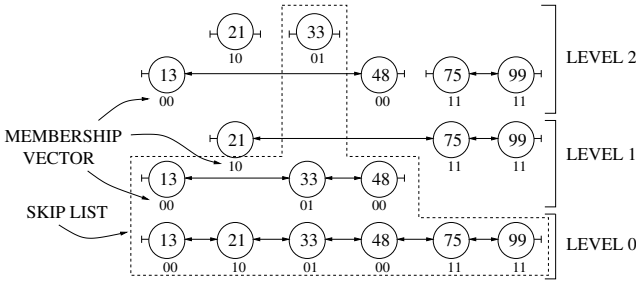


Figure 2: A skip graph with $\lceil \log N \rceil = 3$ levels.

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^ω 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 $w \in \Sigma^\omega$. If $|w| \geq i$, write $w \restriction i$ for the prefix of w of length i. Write ϵ for the empty word.

Returning to skip graphs, the bottom level is always a doubly-linked list S_ϵ consisting of all the elements in order. In general, for each w 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 elements 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 its right neighbor, and in general will think of L_i and R_i as composable operators, to allow writing expressions like $xR_iR_{i-1}^2$ etc.

An alternative view of a skip graph is a **trie** [dlB59, Fre60, Knu73] 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:

LEMMA 2.1. *Let $\{S_w\}$ be a skip graph with alphabet Σ. For any $z \in \Sigma^\omega$, the sequence $S_0, S_1, S_2, \ldots$, where each $S_i = S_{z\restriction i}$, is a skip list with $p = |\Sigma|^{-1}$.*

Proof: By induction on i. The list S_0 equals S_ϵ, which is just the base list of all elements. An element 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 element in S_i appears in S_{i+1} with independent probability p, and $S_0, S_1, \ldots$ form a skip list. ∎

For a peer-to-peer system, each resource will be a node in a skip graph and the nodes are sorted according to the resource key. Each node stores the addresses and the keys of two neighbors at each of the $O(\log n)$ levels. In addition, each node also needs $O(\log n)$ bits of space for its membership vector.

2.1 Algorithms for a skip graph We describe the `search` and `insert` operations for a skip graph but omit the description of `delete`, which is fairly straightforward, to save space.

2.1.1 The `search` operation The `search` operation (Algorithm 1) is exactly the same as in the case of 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 it proceeds along the same level without overshooting the key, continuing at a lower level if required, until it reaches level 0. Either the address of the node storing the search key, if it exists, or the address of the node storing the key closest to the search key is returned.

Algorithm 1: `search` for node n

```
upon receiving ⟨searchOp, startNode, searchKey, level⟩:
if n.key = searchKey then
  send ⟨searchOp, n⟩ to startNode
if n.key < searchKey then
  while level ≥ 0 do
    if (nR_level).key < searchKey then
      send ⟨searchOp, startNode, searchKey, level⟩ to nR_level
      break
    else level←level-1
else
  while level ≥ 0 do
    if (nL_level).key > searchKey then
      send ⟨searchOp, startNode, searchKey, level⟩ to nL_level
      break
    else level←level-1
if level < 0 then
  send ⟨searchOp, n⟩ to startNode
```

LEMMA 2.2. *The `search` operation in a skip graph S with n nodes takes expected $O(\log n)$ time and $O(\log n)$ messages.*

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 $[x \ldots, y]$, 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), and then broadcast the query through the m nodes in the interval by flooding (which takes $O(\log m)$ time and $O(m \log n)$ messages). If the originator of the query is capable of processing m simultaneous responses, the entire operation still takes $O(\log n)$ time.

2.1.2 The `insert` operation A new node n' inserts itself in some list at each level till it finds itself alone in a list at any level (Algorithms 2 and 3). At level 0, n' will link to a node with a key closest to its own key. At each level i, $i \geq 1$, n' will try to find the closest node x in level $i-1$ with $m(x) \restriction i = m(n') \restriction i$ and link to x at level i. Each existing node can delay determining $m(x)_i$ until a new node shows up asking for its value; thus at any given time only a finite prefix of any membership vector has to be generated.

Inserts can be trickier when we have to deal with concurrent node joins. Before n' links to any neighbors, it verifies that its join will not violate the skip graph properties. So if any new nodes have joined the skip graph between n' and its predetermined neighbor, n' will advance over the new nodes if required before linking in the correct place.

Algorithm 2: `insert` for new node n'

if *introducer* $= n'$ **then**
  $n'L_0 = \perp$
  $n'R_0 = \perp$
else
  if *introducer.key* $<$ *n'.key* **then** side = R
  else side = L
  send ⟨`searchOp`, n', n'.key, 0⟩ to introducer
  upon receiving ⟨`searchOp`, neighbor⟩:
  send ⟨`linkOp`, n', side, 0⟩ to neighbor
  level← 1
  while *true* **do**
    if $n'L_{level-1} \neq \perp$ **then**
      send ⟨`buddyOp`, n', level, $m(n')_{\text{level}}$⟩ to $n'L_{\text{level}-1}$
      upon receiving ⟨`buddyOp`, newBuddy, level⟩:
      if *newBuddy* $\neq \perp$ **then**
        send ⟨`linkOp`, n', R, level⟩ to newBuddy
    else if $(n'R_{level-1} \neq \perp) \wedge$ *(newBuddy* $= \perp$*)* **then**
      send ⟨`buddyOp`, n', level, $m(n')_{\text{level}}$⟩ to $n'R_{\text{level}-1}$
      upon receiving ⟨`buddyOp`, newBuddy, level⟩:
      if *newBuddy* $\neq \perp$ **then**
        send ⟨`linkOp`, n', L, level⟩ to newBuddy
      else break
    else break
    level←level+1
    $n'L_{level} = \perp$
    $n'R_{level} = \perp$

LEMMA 2.3. *The* `insert` *operation in a skip graph S with n nodes takes expected $O(\log n)$ time and $O(\log n)$ messages.*

3 Repair Mechanism

In this section, we describe a self-stabilization mechanism that repairs the skip graph in the event of node and link failures. We first characterize the constraints for an ideal skip graph. Let x be any node in the skip graph; then for any level i:

1. If $xR_i \neq \perp$, $xR_i > x$.
2. If $xL_i \neq \perp$, $xL_i < x$.

Algorithm 3: `Insert` for existing node n

upon receiving ⟨`linkOp`, n', side, level⟩:
if *side* $= R$ **then** cmp $=<$
else cmp $=>$
if *(n* $side_{level}$*).key cmp n'.key* **then**
  send ⟨`linkOp`, n', side, level⟩ to n $side_{\text{level}}$
else
  adjust links to add n' as side neighbor
  send ⟨`linkOp`, n', otherSide, level⟩ to n $side_{level}$
upon receiving ⟨`buddyOp`, n', level, val⟩ from side L(R):
if $m(n)_{level} = \perp$ **then**
  $m(n)_{\text{level}}$ = getCoin()
  $nL_{level} = \perp$
  $nR_{level} = \perp$
if $m(n)_{level} = val$ **then**
  send ⟨`buddyOp`, n, level⟩ to n'
else
  if $nR_{level}(L_{level}) \neq \perp$ **then**
    send ⟨`buddyOp`, n', val, level⟩ to nR_{level} (nL_{level})
  else
    send ⟨`buddyOp`, $\perp$, level⟩ to n'

3. If $xL_i \neq \perp$, $xL_iR_i = x$.
4. If $xR_i \neq \perp$, $xR_iL_i = x$.
5. If $i > 0$, $m(x) \restriction i = m(xR^l_{i-1}) \restriction i$ and $\nexists k, k < l, m(x) \restriction i = m(xR^k_{i-1}) \restriction i$, then $xR_i = xR^l_{i-1}$.
6. If $i > 0$, $m(x) \restriction i = m(xL^l_{i-1}) \restriction i$ and $\nexists k, k < l, m(x) \restriction i = m(xL^k_{i-1}) \restriction i$, then $xL_i = xL^l_{i-1}$.

THEOREM 3.1. *Every connected component of the data structure is a skip graph if and only if conditions $1-6$ are satisfied.*

3.1 Maintaining the invariant Define $\perp L_i = \perp R_i = \perp$. We define conditions $1-4$ as an **invariant** for a skip graph as they hold in all states with no undelivered messages, even in the presence of failures. Conditions $5-6$ may fail to hold with failures, but they can be restored by the repair mechanism. We shall call conditions 5 and 6 as the L and R **successor** conditions respectively.

THEOREM 3.2. *With no undelivered messages, the invariant is maintained for a skip graph with node insertions, deletions and node failures.*

3.2 Restoring skip graph constraints The successor conditions get violated during insert and

delete operations as well as when a node or a link fails.

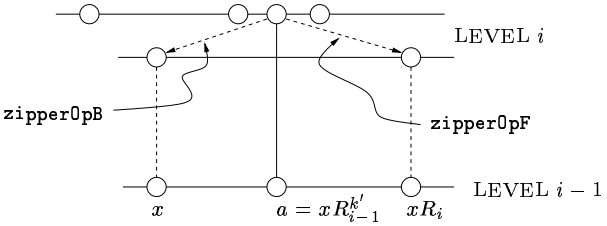
Although the skip graph constraints may get violated during an insert or a delete operation, once no messages are pending and provided no additional inserts, deletes or failures occur, the successor conditions are satisfied. Thus we see that the repair mechanism is required to restore the successor conditions only in case of node or link failures. We consider the possible cases in which the successor conditions can be violated and provide a repair mechanism for the each of those cases. We will concentrate on the repair mechanism for the R links and fixing the L links is symmetric. It may be possible to combine the two mechanisms to improve the performance but we will treat them separately for simplicity.

There are two cases when the R successor condition is violated:

1. $xR_i = xR_{i-1}^k$ but $\exists a = xR_{i-1}^{k'}$, $k' < k$, $m(x) \restriction i = m(a) \restriction i$. This case occurs when two nodes are connected to each other at levels $i-1$ and i, and a new node in inserted between them at level $i-1$ but is pending to be inserted between them at level i. If the left neighbor of the new node checks its R successor condition at level i before the insert of the new node at level i is completed, it will detect a discrepancy.
2. $xR_i \neq xR_{i-1}^k$, for any k. This case occurs with the failure of any node or link in an ideal skip graph.

We consider each case in detail and propose a repair mechanism for each violation.

Case 1: $xR_i = xR_{i-1}^k$, but $\exists a = xR_{i-1}^{k'}, k' < k, m(x) \restriction i = m(a) \restriction i$.

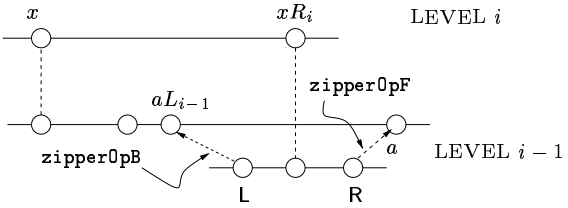


Node a should be inserted into level i and this is done by sending the following messages[1]:

- Send $\langle$zipperOpF, xR_i, $i\rangle$ to a.
- Send $\langle$zipperOpB, x, $i\rangle$ to a.

Case 2: $xR_i \neq xR_{i-1}^k$, for any k. There are three ways to repair this violation depending on what other nodes are present at level $i-1$.

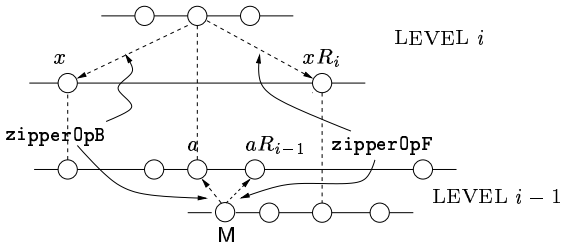
Case 2a: $\exists a = xR_{i-1}^k > xR_i$ and $\nexists b = xR_{i-1}^+ < a$ such that $m(b) \restriction i = m(x) \restriction i$.



The nodes connected to a and xR_i at level $i-1$ have to be merged together into one ring by sending the following messages:

- Probe level $i-1$ to find largest $xR_iR_{i-1}^{k'} = \mathsf{R} < a$.
- Send $\langle$zipperOpF, a, $i-1\rangle$ to R.
- Probe level $i-1$ to find smallest $xR_iL_{i-1}^{k''} = \mathsf{L} > aL_{i-1}$.
- Send $\langle$zipperOpB, aL_{i-1}, $i-1\rangle$ to L.

Case 2b: $\exists a = xR_{i-1}^k < xR_i, m(a) \restriction i = m(x) \restriction i$ and $xR_{i-1}^+ \neq xR_i$.

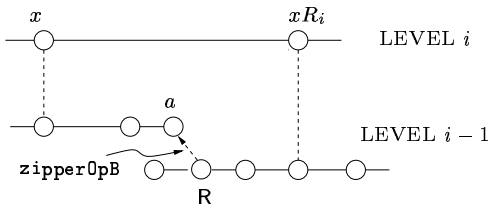


The nodes connected to a and xR_i have to be merged at levels i and $i-1$ respectively by sending the following messages:

- Probe level $i-1$ to find smallest $xR_iL_{i-1}^{k'} = \mathsf{M} > a$.
- Send $\langle$zipperOpB, a, $i-1\rangle$ to M.
- Send $\langle$zipperOpF, aR_{i-1}, $i-1\rangle$ to M.
- Send $\langle$zipperOpB, x, $i\rangle$ to a.
- Send $\langle$zipperOpF, xR_i, $i\rangle$ to a.

[1] Details of the zipperOp algorithm are given in Algorithms 4 and 5.

Case 2c: $\exists a < xR_i, aR_i = \perp$.



The nodes connected to a and xR_i at level $i-1$ have to be merged by sending the following messages:

- Probe level $i-1$ to find smallest $xR_iL^k_{i-1} = \mathsf{R} > a$.
- Send $\langle$`zipperOpB`, a, $i-1\rangle$ to R.

Algorithm 4: `zipperOpB` for node n

```
upon receiving ⟨zipperOpB, x, ℓ⟩:
if nL_ℓ > x.key then
  send ⟨zipperOpB, x, ℓ⟩ to nL_ℓ
else
  tmp = nL_ℓ
1 nL_ℓ = x
2 xR_ℓ = n
  if tmp ≠ ⊥ then
    send ⟨zipperOpB, tmp, ℓ⟩ to x
```

Algorithm 5: `zipperOpF` for node n

```
upon receiving ⟨zipperOpF, x, ℓ⟩:
if nR_ℓ <x.key then
  send ⟨zipperOpF, x, ℓ⟩ to nR_ℓ
else
  tmp = nR_ℓ
1 xL_ℓ = n
2 nR_ℓ = x
  if tmp ≠ ⊥ then
    send ⟨zipperOpF, tmp, ℓ⟩ to x
```

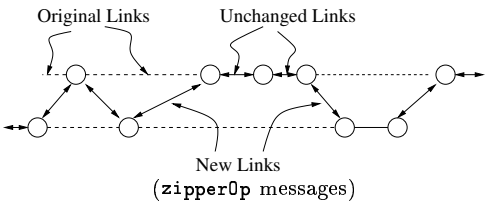


Figure 3: `zipperOp` operation to merge nodes on the same level.

THEOREM 3.3. *In the absence of new failures, the repair mechanism described in Section 3.2 will eventually restore the violated constraints of a skip graph, without losing existing connectivity.*

4 Fault Tolerance

In this section, we describe some of the fault tolerance properties of a skip graph. Fault tolerance of related data structures, such as augmented versions of linked lists and binary trees, has been well-studied and some results can be seen in [MP84, AB96]. The main question is how many nodes can be separated from the primary component by the failure of other nodes, as this determines the size of the surviving skip graph after the repair mechanism finishes.

We show first that even a worst-case choice of failures by an adversary that can observe the structure of the skip graph can do only limited damage. With high probability, a skip graph with n nodes has an $\Omega(1/\log n)$ expansion ratio, implying that at most $O(f \log n)$ nodes can be separated from the primary component by f failures. These results are described in Section 4.1

For random failures, the situation appears even more promising; our experimental results, presented in Section 4.2, 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 failure even without using the repair mechanism by use of redundant links.

4.1 Adversarial failures Given a subset A of the nodes of a skip graph, define δA as the set of all nodes that are not in A but that are adjacent to A. Further define $\delta_h A$ as the set of all nodes that are not in A but are joined to a node in A by an edge at level h. Clearly $\delta A = \bigcup_h \delta_h A$ and $|\delta A| \geq \max_h |\delta_h A|$.

The expansion ratio of a set A is $|\delta A|/|A|$. The expansion ratio of a graph is the minimum expansion ratio of any set A for which $1 \leq |A| \leq n/2$. The expansion ratio determines the resilience of a skip 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 $\Omega(1/\log n)$ expansion ratios with high probability, implying that only $O(f \log n)$ 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 $\delta_h A$ for some particular h chosen based on the size of A. We begin by counting the number of sets A of a given size that have small $\delta_0 A$.

LEMMA 4.1. *In an n-node skip graph, the number of sets A, where $|A| = m < n$ and $|\delta_0 A| < s$, is less than $\sum_{r=1}^{s-1} \binom{m+1}{r}\binom{n-m-1}{r-1}$.*

Sketch of proof: Represent each A as a bit-vector where 1 indicates a member of the set and 0 a non-member. Then $|\delta_0 A|$ is at least the number of intervals of zeroes in this bit-vector. The bound in the lemma is then obtained by bounding the number of length n bit-vectors with m ones and at most s intervals of zeroes. ∎

LEMMA 4.2. *Let A be a subset of $m \leq n/2$ nodes of an n-node skip graph S. Then for any h, $\Pr\left[|\delta_h A| \leq \frac{1}{3} \cdot 2^h\right] < 2\binom{2^h}{\lfloor \frac{2}{3} \cdot 2^h \rfloor}(2/3)^m$.*

Sketch of proof: The key observation is that for each b in $\{0,1\}^h$, each skip list S_b that contains a member of both A and its complement contributes at least one distinct element to $\delta_h A$. We then show that at least a third of the S_b are likely to do so by bounding the probability that either A or $S - A$ are represented in less than two-thirds of the S_b. ∎

THEOREM 4.1. *Let $c \geq 6$. Then a skip graph with n nodes has an expansion ratio of at least $\frac{1}{c \log_{3/2} n}$ with probability $1 - O(n^{5-c})$, where the constant factor does not depend on c.*

Sketch of proof: The probability bound is obtained by summing the probability of having $\delta_h A$ too small over all A for which $\delta_0 A$ is too small. For each set A of size m, h is chosen so that the $\frac{1}{3} \cdot 2^h$ bound of Lemma 4.2 exceeds m times the expansion ratio. The probabilities derived from Lemma 4.2 are then summed over all sets A of a fixed size m using Lemma 4.1, and the result of this process is summed over all $m > c \log_{3/2} n$ to obtain the final bound. ∎

4.2 Random failures In our experiments, skip graphs appear to be highly resilient against random failures. As shown in Figure 4, nearly all nodes remain in the primary component even as the probability of individual node failure exceeds 0.6, and we suspect that most of the lost nodes at this stage become isolated only 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 that $O(\log n)$. By detecting failures locally and using additional redundant edges, we can make searches highly tolerant to small numbers of random faults; some experimental results are shown in Figure 5. In these experiments, each node x has extra links to its five nearest neighbors on each side, at every level that it is a member of. In general, we cannot make as strong guarantees as those provided by data structures based on explicit use of expanders [FS02, Dat02], but we believe that this is compensated for by the simplicity of skip graphs and the existence of good distributed mechanisms for constructing and repairing them.

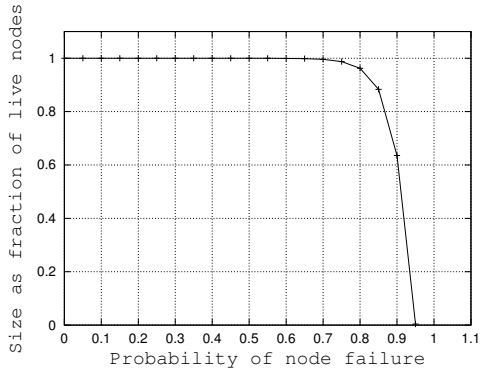


Figure 4: Size of the largest connected component as a fraction of the surviving nodes with 131072 nodes.

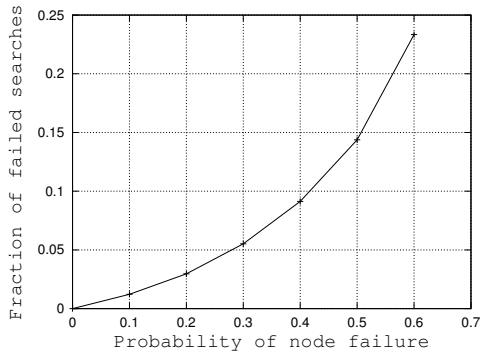


Figure 5: Failed Searches with 131072 nodes and 10000 messages.

5 Load balancing

In addition to fault-tolerance, a skip graph provides a limited form of load balancing, by smoothing out hot spots caused by popular search targets. The guarantees that a skip graph makes in this case are similar to the guarantees made for survivability. Just

as an element stored at a particular node will not survive the loss of that node or its neighbors in the graph, many searches directed at a particular element will lead to high load on the node that stores it and on nodes likely to be on a search path. However, we can show that this effect drops off rapidly with distance; elements that are far away from a popular target in the bottom-level list produce little additional load on average.

We give two characterizations of this result. The first shows that the probability that a particular search uses a node between the source and target drops off inversely with the distance from the node to the 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. Our second characterization addresses this issue by showing that most of the load-spreading effects are the result of assuming a random membership vector for the source of the search.

THEOREM 5.1. *Let S be a skip graph with alphabet $\{0,1\}$, and consider a search from s to t in S. Let u be node with $s < u < t$ in the key ordering 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 $\frac{2}{d+1}$.*

Theorem 5.1 is of small consolation to some node that draws a $\frac{2}{d+1}$ straw and participates in every search. Fortunately, such things do not happen often. Define the **average load** L_{tu} imposed by a search for t on a node u in a given skip graph S as the probability that an $s-t$ search hits u conditioned on the membership vectors of all nodes in the interval $[u,t]$, where $s < u < t$. 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 5.2. *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 $\alpha \geq 0$, $\Pr[L_{ut} > \alpha] \leq 2e^{-\alpha d/2}$.*

6 Conclusion

We have defined a new data structure, the skip graph, for distributed data stores that has several desirable properties. Constructing, inserting new nodes into and searching in a skip graph can be done in logarithmic time. 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 and clustering of nodes with similar keys.

This data structure gives rise to a class of random graphs whose properties we have only begun to examine: some open problems remain regarding the reliability of these graphs. Also, skip graphs do not exploit geographical proximity in location of resources and it would be interesting to study performance benefits in that direction, perhaps by using multi-dimensional skip graphs. Finally, while the theoretical properties and relative simplicity of skip graphs make them a good candidate for implementation, the ultimate test of their usefulness will be their performance in practice. This is an issue that we hope to study soon.

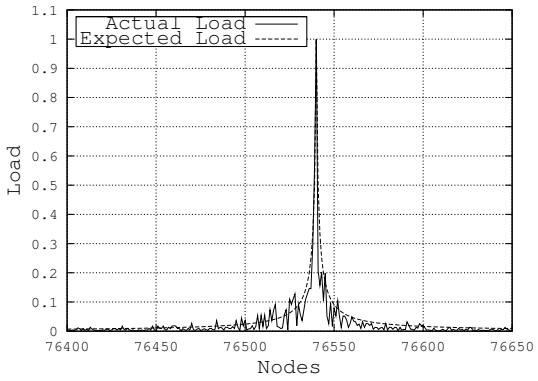


Figure 6: Actual and expected load in a skip graph with 131072 nodes with the target=76539. Messages were delivered from each node to the target and the actual load on each node was measured. The expected load is computed using Theorem 5.1.

References

[AB96] Yonatan Aumann and Michael A. Bender. Fault tolerant data structures. In *Thirty-Seventh Annual Symposium on Foundations of Computer Science*, pages 580–589, Burlington, VT, USA, October 1996.

[ADS02] James Aspnes, Zoë Diamadi, and Gauri Shah. Fault-tolerant routing in peer-to-peer systems. In *Twenty-First ACM Symposium on Principles of Distributed Computing*, pages 223–232, Monterey, MA, USA, July 2002.

[CDHR02] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Anthony Rowstron. Exploiting network proximity in peer-to-peer overlay networks. In *International Workshop on Future Directions in Distributed Computing*, Bertinoro, Italy, June 2002. [Longer version submitted for publication].

[Dat02] Mayur Datar. Butterflies and peer-to-peer networks. In *Proceedings of the 10th European Symposium on Algorithms*, Rome, Italy, September 2002.

[Dev92] L. Devroye. A limit theory for random skip lists. *The Annals of Applied Probability*, 2(3):597–609, 1992.

[dlB59] Rene de la Briandais. File searching using variable length keys. In *Western Joint Computer Conference*, volume 15, pages 295–298, Montvale, NJ, USA, 1959. AFIPS Press.

[FRE] FREENET. http://www.freenet.sourceforge.net.

[Fre60] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.

[FS02] Amos Fiat and Jared Saia. Censorship resistant peer-to-peer content addressable networks. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, CA, USA, January 2002.

[GM97] J. Gabarró and X. Messeguer. A unified approach to concurrent and parallel algorithms on balanced data structures. In *XVII International Conference of the Chilean Computer Society*, 1997.

[GMM93] J. Gabarró, C. Martínez, and X. Messeguer. Parallel update and search in skip lists. In *13th International Conference of the Chilean Computer Society*, 1993.

[GMM96] J. Gabarró, C. Martínez, and X. Messeguer. A top-down design of a parallel dictionary using skip lists. *Theoretical Computer Science*, 158(1–2):1–33, May 1996.

[GNU] GNUTELLA. http://gnutella.wego.com.

[HHH+02] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. Complex queries in DHT-based peer-to-peer networks. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, USA, March 2002.

[HKRZ02] Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed object location in a dynamic network. In *Fourteenth ACM Symposium on Parallel Algorithms and Architectures*, Winnipeg, Manitoba, Canada, August 2002.

[JKZ01] Anthony D. Joseph, John Kubiatowicz, and Ben Y. Zhao. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California, Berkeley, Apr 2001.

[KMP95] P. Kirschenhofer, C. Martínez, and H. Prodinger. Analysis of an optimized search algorithm for skip lists. *Theoretical Computer Science*, 144(1–2):119–220, 26 June 1995.

[Knu73] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 1973.

[KP94] P. Kirschenhofer and H. Prodinger. The path length of random skip lists. *Acta Informatica*, 31(8):775–792, 1994.

[KR02] David Karger and Matthias Ruhl. Finding nearest neighbors in growth-restricted metrics. In *Thirty-Fourth ACM Symposium on Theory of Computing*, pages 741–750, Montreal, Canada, May 2002.

[MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalabale and dynamic emulation of the butterfly. In *Twenty-First ACM Symposium on Principles of Distributed Computing*, pages 183–192, Monterey, CA, USA, July 2002.

[MP84] J. Ian Munro and Patricio V. Poblete. Fault tolerance and storage reduction in binary search trees. *Information and Control*, 62(2/3):210–218, August 1984.

[NAP] NAPSTER. Formerly, http://www.napster.com.

[PMP90] T. Papadakis, J.I. Munro, and P.V. Poblete. Analysis of the expected search cost in skip lists. In J. R. Gilbert and R. G. Karlsson, editors, *SWAT 90, 2nd Scandinavian Workshop on Algorithm Theory*, volume 447 of *Lecture Notes in Computer Science*, pages 160–172, Bergen, Norway, 11–14 July 1990. Springer.

[PRR97] C. Plaxton, R. Rajaram, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1997.

[Pug90] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM*, pages 161–170, 2001.

[SBK02] Bujor Silaghi, Bobby Bhattacharjee, and Pete Keleher. Query routing in the terradir distributed directory. In *SPIE ITCOM 2002*, August 2002.

[SMK+01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishna. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM 2001*, pages 149–160, 2001.

[ZJK02] Ben Y. Zhao, Anthony D. Joseph, and John D. Kubiatowicz. Locality-aware mechanisms for large-scale networks. In *Workshop on Future Directions in Distributed Computing*, Bertinoro, Italy, June 2002.