SkipNet: A Scalable Overlay Network with Practical Locality Properties

Nicholas J.A. Harvey[‡] John Dunagan[*] Michael B. Jones[*] Stefan Saroiu[†]
Marvin Theimer[*] Alec Wolman[*]

Abstract: *Scalable overlay networks such as Chord, CAN, Pastry, and Tapestry have recently emerged as flexible infrastructure for building large peer-to-peer systems. In practice, such systems have two disadvantages: They provide no control over where data is stored and no guarantee that routing paths remain within an administrative domain whenever possible. SkipNet is a scalable overlay network that provides controlled data placement and guaranteed routing locality by organizing data primarily by string names. SkipNet allows for both fine-grained and coarse-grained control over data placement: Content can be placed either on a pre-determined node or distributed uniformly across the nodes of a hierarchical naming subtree. An additional useful consequence of SkipNet's locality properties is that partition failures, in which an entire organization disconnects from the rest of the system, can result in two disjoint, but well-connected overlay networks. Furthermore, SkipNet can efficiently re-merge these disjoint networks when the partition heals.*

Categories and Subject Descriptors: E.1 [**Data Structures**]: Graphs and networks; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns (Peer-to-Peer)*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Network topology*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Routing protocols* ; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Store and forward networks*; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*.

General Terms: Overlay Network, Peer-to-Peer, Locality, Scalablity, Routing, Performance, Fault-Tolerance, Distributed Systems.

1 Introduction

Scalable overlay networks, such as Chord [38], CAN [32], Pastry [34], and Tapestry [44], have recently emerged as flexible infrastructure for building large peer-to-peer systems. A key function that these networks enable is a distributed hash table (DHT), which allows data to be uniformly diffused over all the participants in the peer-to-peer system.

While DHTs provide nice load balancing properties, they do so at the price of controlling where data is stored. This has at least two disadvantages: Data may be stored far from its users and it may be stored outside the administrative domain to which it belongs. This paper introduces SkipNet [16, 17], a distributed generalization of Skip Lists [30], adapted to meet the goals of peer-to-peer systems. SkipNet is a scalable overlay network that supports traditional overlay functionality as well as two locality properties that we refer to as *content locality* and *path locality*.

Content locality refers to the ability to either explicitly place data on specific overlay nodes or distribute it across nodes within a given organization. Path locality refers to the ability to guarantee that message traffic between two overlay nodes within the same organization is routed within that organization only.

Content and path locality provide a number of advantages for data retrieval, including improved availability, performance, manageability, and security. For example, nodes can store important data within their organization (content locality) and nodes will be able to reach their data through the overlay even if the organization becomes disconnected from the rest of the Internet (path locality). Storing data near the clients that use it also yields performance benefits. Placing content onto a specific overlay node—or a well-defined set of overlay nodes—enables provisioning of those nodes to reflect demand. Content placement also allows administrative control over issues such as scheduling maintenance for machines storing important data, thus improving manageability.

Content locality can improve security by allowing one to control the administrative domain in which data resides. Even when encrypted and digitally signed, data stored on an arbitrary overlay node outside the organization is susceptible to denial of service (DoS) attacks as well as traffic analysis. Although other techniques

[*]Microsoft Research, Microsoft Corporation, Redmond, WA 98052, {jdunagan, mbj, theimer, alecw}@microsoft.com

[†]Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, tzoompy@cs.washington.edu

[‡]MIT Computer Science and Artificial Intelligence Lab, Cambridge, MA 02139, nickh@mit.edu

for improving the resiliency of DHTs to DoS attacks exist [3], content locality is a simple, zero-overhead technique.

Path locality provides additional security benefits to an overlay that supports content locality. Although some overlay designs [4] are likely to keep routing messages within an organization most of the time, none *guarantee* path locality. For example, without such a guarantee the route from *explorer.ford.com* to *mustang.ford.com* could pass through *camaro.gm.com*, a scenario that people at *ford.com* might prefer to prevent. With path locality, nodes requesting data within their organization traverse a path that never leaves the organization. This example also illustrates that path locality can be desirable even in a scenario where no content is being placed on nodes.

Controlling content placement is in direct tension with the goal of a DHT, which is to uniformly distribute data across a system in an automated fashion. A significant contribution of this paper is the concept of *constrained load balancing*, which is a generalization that combines these two notions: Data is uniformly distributed across a well-defined subset of the nodes in a system, such as all nodes in a single organization, all nodes residing within a given building, or all nodes residing within one or more data centers.

SkipNet supports efficient message routing between overlay nodes, content placement, path locality, and constrained load balancing. It does so by employing two separate, but related address spaces: a string name ID space as well as a numeric ID space. Node names and content identifier strings are mapped directly into the name ID space, while hashes of the node names and content identifiers are mapped into the numeric ID space. A single set of routing pointers on each overlay node enables efficient routing in either address space and a combination of routing in both address spaces provides the ability to do constrained load balancing.

A useful consequence of SkipNet's locality properties is resiliency against a common form of Internet failure. Because SkipNet clusters nodes according to their name ID ordering, nodes within a single organization gracefully survive failures that disconnect the organization from the rest of the Internet. Furthermore, the organization's SkipNet segment can be efficiently remerged with the external SkipNet when connectivity is restored. In the case of uncorrelated, independent failures, SkipNet has similar resiliency to previous overlay networks [34, 38].

The basic SkipNet design, not including its enhancements to support constrained load balancing, network proximity-aware routing, reduced overhead for virtual nodes, or merge algorithms, has been concurrently and independently invented by Aspnes and Shah [1]. As described in Section 2, their work has a substantially different focus than our work and the two efforts are complementary to each other while still starting from the same underlying inspiration.

The rest of this paper is organized as follows: Section 2 describes related work, Section 3 describes SkipNet's basic design, Section 4 discusses SkipNet's locality properties, Section 5 presents enhancements to the basic design, Section 6 presents the ring merge algorithms, Section 7 discusses design alternatives to SkipNet, Section 8 presents a theoretical analysis of SkipNet, Section 9 presents an experimental evaluation, and Section 10 concludes the paper.

2 Related Work

A large number of peer-to-peer overlay network designs have been proposed recently, such as CAN [32], Chord [38], Freenet [7], Gnutella [14], Kademlia [26], Pastry [34], Salad [11], Tapestry [44], and Viceroy [25]. SkipNet is designed to provide the same functionality as existing peer-to-peer overlay networks, and additionally to provide improved content availability through explicit control over content placement.

One key feature provided by systems such as CAN, Chord, Pastry, and Tapestry is scalable routing performance while maintaining a scalable amount of routing state at each node. By scalable routing paths we mean that the expected number of forwarding hops between any two communicating nodes is small with respect to the total number of nodes in the system. Chord, Pastry, and Tapestry scale with $\log N$, where N is the system size, while maintaining $\log N$ routing state at each overlay node. CAN scales with $D \cdot N^{1/D}$, where D is a dimensionality parameter with a typical value of 6, while maintaining an amount of per-node routing state proportional to D.

A second key feature of these systems is that they are able to route to destination addresses that do not equal the address of any existing node. Each message is routed to the node whose address is "closest" to that specified in the destination field of a message; we interchangeably use the terms "route" and "search" to mean routing to the closest node to the specified destination. This feature enables implementation of a distributed hash table (DHT) [15], in which content is stored at an overlay node whose node ID is closest to the result of applying a collision-resistant hash function to that content's name (i.e. consistent hashing [21]).

Distributed hash tables have been used, for instance, in constructing the PAST [35] and CFS [9] distributed filesystems, the Overlook [41] scalable name service, the Squirrel [19] cooperative web cache, and scalable application-level multicast [6, 36, 33]. For most of these systems, if not all of them, the overlay network on which they were designed can easily be substituted with SkipNet.

SkipNet has a fundamental philosophical difference from existing overlay networks, such as Chord and Pastry, whose goal is to implement a DHT. The basic philosophy of systems like Chord and Pastry is to diffuse content randomly throughout an overlay in order to obtain uniform, load-balanced, peer-to-peer behavior. The basic philosophy of SkipNet is to enable systems to preserve useful content and path locality, while still enabling load balancing over constrained subsets of participating nodes.

This paper is not the first to observe that locality properties are important in peer-to-peer systems. Keleher et al. [22] make two main points: locality is a good thing, and DHTs destroy locality. Vahdat et al. [42] raises the locality issue as well. SkipNet addresses this problem directly: By using names rather than hashed identifiers to order nodes in the overlay, natural locality based on the names of objects is preserved. Furthermore, by arranging content in name order rather than dispersing it, efficient operations on ranges of names are possible in SkipNet, enabling, among other things, constrained load balancing.

Aspnes and Shah [1] have independently invented the same basic data structure that defines a SkipNet, which they call a Skip Graph. Beyond that, they investigate questions that are mostly orthogonal to those addressed in this paper. In particular, they describe and analyze different search and insertion algorithms and they focus on formal characterization of Skip Graph invariants. In contrast, our work is focused primarily on the content and path locality properties of the design, and we describe several extensions that are important in building a practical system: network proximity-aware routing is obtained by means of two auxiliary routing tables; constrained load balancing is supported through a combination of searches in both the string name and numeric address spaces that SkipNet defines; efficient algorithms are used to re-merge disjoint SkipNet segments that result from network partitions; and multiple virtual nodes can be hosted on a single physical node with substantially less overhead than the schemes described in previous work.

3 Basic SkipNet Structure

In this section, we introduce the basic design of SkipNet. We present the SkipNet architecture, including how to route in SkipNet, and how to join and leave a SkipNet.

3.1 Analogy to Skip Lists

A Skip List, first described in Pugh [30], is a dictionary data structure typically stored in-memory. A Skip List is a sorted linked list in which some nodes are supplemented with pointers that skip over many list elements. A "perfect" Skip List is one where the height of the i^{th} node is the exponent of the largest power-of-two that divides i. Figure 1 depicts a perfect Skip List. Note that pointers at level h have length 2^h (i.e., they traverse 2^h nodes). A perfect Skip List supports searches in $O(\log N)$ time.

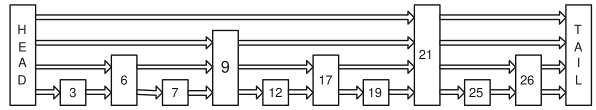

Figure 1. *A perfect Skip List.*

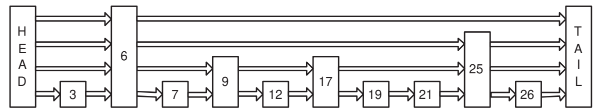

Figure 2. *A probabilistic Skip List.*

Because it is prohibitively expensive to perform insertions and deletions in a perfect Skip List, Pugh suggests a probabilistic scheme for determining node heights while maintaining $O(\log N)$ searches with high probability. Briefly, each node chooses a height such that the probability of choosing height h is $1/2^h$. Thus, with probability $1/2$ a node has height 1, with probability $1/4$ it has height 2, etc. Figure 2 depicts a probabilistic Skip List.

Whereas Skip Lists are an in-memory data structure that is traversed from its head node, we desire a data structure that links together distributed computer nodes and supports traversals that may start from any node in the system. Furthermore, because peers should have uniform roles and responsibilities in a peer-to-peer system, we desire that the state and processing overhead of all nodes be roughly the same. In contrast, Skip Lists maintain a highly variable number of pointers per data record and experience a substantially different amount of traversal traffic at each data record.

3.2 The SkipNet Structure

The key idea we take from Skip Lists is the notion of maintaining a sorted list of all data records as well as pointers that "skip" over varying numbers of records. We transform the concept of a Skip List to a distributed system setting by replacing data records with computer nodes, using the string *name IDs* of the nodes as the data record keys, and forming a ring instead of a list. The ring must be doubly-linked to enable path locality, as is explained in Section 3.3.

Rather than having nodes store a highly variable number of pointers, as in Skip Lists, each SkipNet node stores $2 \log N$ pointers, where N is the number of nodes in the overlay system. Each node's set of pointers is called

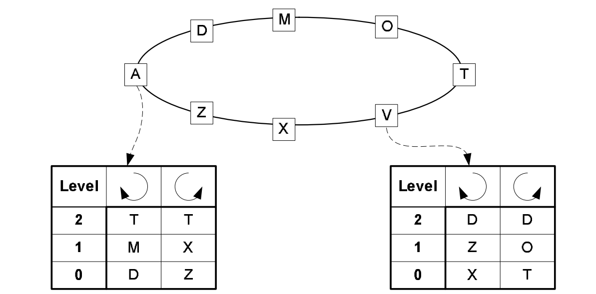

Figure 3. *SkipNet nodes ordered by name ID. Routing tables of nodes A and V are shown.*

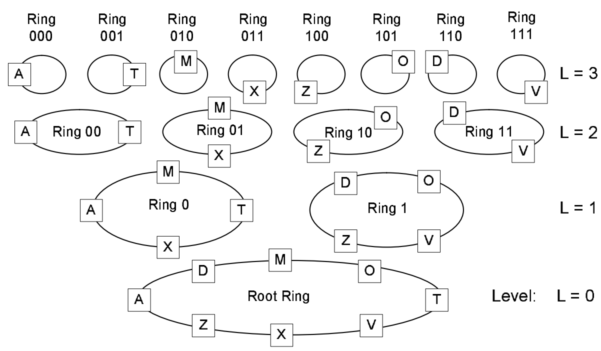

Figure 4. *The full SkipNet routing infrastructure for an 8 node system, including the ring labels.*

its *routing table*, or R-Table, since the pointers are used to route message traffic between nodes. The pointers at level h of a given node's routing table point to nodes that are roughly 2^h nodes to the left and right of the given node. Figure 3 depicts a SkipNet containing eight nodes and shows the routing table pointers that nodes A and V maintain.

The SkipNet in Figure 3 is a "perfect" SkipNet: each level h pointer traverses exactly 2^h nodes. Figure 4 depicts the same SkipNet of Figure 3, arranged to show all node interconnections at every level simultaneously. All nodes are connected by the *root ring* formed by each node's pointers at level 0. The pointers at level 1 point to nodes that are 2 nodes away and hence the overlay nodes are implicitly divided into two disjoint rings. Similarly, pointers at level 2 form four disjoint rings of nodes, and so forth. Note that rings at level $h+1$ are obtained by splitting a ring at level h into two disjoint sets, each ring containing every second member of the level h ring.

Maintaining a perfect SkipNet in the presence of insertions and deletions is impractical, as is the case with perfect Skip Lists. To facilitate efficient insertions and deletions, we derive a probabilistic SkipNet design. Each ring at level h is split into two rings at level $h+1$ by having each node randomly and uniformly choose to which of the two rings it belongs. With this probabilistic scheme, insertion/deletion of a node only affects two other nodes in each ring to which the node has randomly chosen to belong. Furthermore, a pointer at level h still skips over 2^h nodes in expectation, and routing is possible in $O(\log N)$ forwarding hops with high probability.

Each node's random choice of ring memberships can be encoded as a unique binary number, which we refer to as the node's *numeric ID*. As illustrated in Figure 4, the first h bits of the number determine ring membership at level h. For example, node X's numeric ID is 011 and its membership at level 2 is determined by taking the first 2 bits of 011, which designate Ring 01. As described in [38], there are advantages to using a collision-resistant hash (such as SHA-1) of the node's DNS name as the numeric ID. The SkipNet design does not require the use of hashing to generate nodes' numeric IDs; we only require that numeric IDs are random and unique.

Because the numeric IDs of nodes are unique they can be thought of as a second address space that is maintained by the same SkipNet data structure. Whereas SkipNet's string address space is populated by node name IDs that are *not* uniformly distributed throughout the space, SkipNet's numeric address space is populated by node numeric IDs that *are* uniformly distributed. The uniform distribution of numeric IDs in the numeric space is what ensures that our routing table construction yields routing table entries that skip over the appropriate number of nodes.

Readers familiar with Chord may have observed that SkipNet's routing pointers are exponentially distributed in a manner similar to Chord's: The pointer at level h hops over 2^h nodes in expectation. The fundamental difference is that Chord's routing pointers skip over 2^h nodes in the numeric space. In contrast SkipNet's pointers, when considered from level 0 upward, skip over 2^h nodes in the name ID space and, when considered from the top level downward, skip over 2^h nodes in the numeric ID space. Chord guarantees $O(\log N)$ routing and node insertion performance by uniformly distributing node identifiers in its numeric address space. SkipNet guarantees $O(\log N)$ performance of node insertion and routing in both the name ID and numeric ID spaces by uniformly distributing numeric IDs and leveraging the sorted order of name IDs.

3.3 Routing by Name ID

Routing/searching by name ID in SkipNet is based on the same basic principle as searching in Skip Lists: Follow pointers that route closest to the intended destination. At each node, a message will be routed along the highest-level pointer that does not point past the destination value. Routing terminates when the message arrives at a node whose name ID is closest to the destination.

```
SendMsg(nameID, msg) {
  if( LongestPrefix(nameID,localNode.nameID)==0 )
    msg.dir = RandomDirection();
  else if( nameID<localNode.nameID )
    msg.dir = counterClockwise;
  else
    msg.dir = clockwise;
  msg.nameID = nameID;
  RouteByNameID(msg);
}

// Invoked at all nodes (including the source and
// destination nodes) along the routing path.
RouteByNameID(msg) {
  // Forward along the longest pointer
  // that is between us and msg.nameID.
  h = localNode.maxHeight;
  while (h >= 0) {
    nbr = localNode.RouteTable[msg.dir][h];
    if (LiesBetween(localNode.nameID, nbr.nameID,
                    msg.nameID, msg.dir)) {
      SendToNode(msg, nbr);
      return;
    }
    h = h - 1;
  }
  // h<0 implies we are the closest node.
  DeliverMessage(msg.msg);
}
```

Figure 5. *Algorithm for routing by name ID in SkipNet.*

Figure 5 presents this algorithm in pseudocode.

Since nodes are ordered by name ID along each ring and a message is never forwarded past its destination, all nodes encountered during routing have name IDs between the source and the destination. Thus, when a message originates at a node whose name ID shares a common prefix with the destination, all nodes traversed by the message have name IDs that share that same prefix. Because rings are doubly-linked, this scheme can route using either right or left pointers depending upon whether the source's name ID is smaller or greater than the destination's. The key observation of this scheme is that routing by name ID traverses only nodes whose name IDs share a non-decreasing prefix with the destination ID. Section 8.5 proves that node stress is well-balanced even under this scheme.

If the source name ID and the destination name ID share no common prefix, a message can be routed in either direction. For the sake of fairness, we randomly pick a direction so that nodes whose name IDs are near the middle of the sorted ordering do not get a disproportionately large share of the forwarding traffic.

The number of message hops when routing by name ID is $O(\log N)$ with high probability. For a proof see Section 8.1.

3.3.1 Generality of the Name ID Space

In this paper we use examples where the name IDs for each node are chosen based on the name of the organization that the node belongs to. However, SkipNet has many potential uses beyond this particular naming scheme. Because there is no requirement that the names be uniformly distributed along the root ring in order to achieve $O(\log N)$ routing performance, there are no constraints placed on the strategy one uses to select names for nodes and data.

```
// Invoked at all nodes (including the source and
// destination nodes) along the routing path.
// Initially:
//   msg.ringLvl = -1
//   msg.startNode = msg.bestNode = null
//   msg.finalDestination = false
RouteByNumericID(msg) {
  if (msg.numID == localNode.numID ||
      msg.finalDestination) {
    DeliverMessage(msg.msg);
    return;
  }

  if (localNode == msg.startNode) {
    // Done traversing current ring.
    msg.finalDestination = true;
    SendToNode(msg.bestNode);
    return;
  }

  h = CommonPrefixLen(msg.numID, localNode.numID);
  if (h > msg.ringLvl) {
    // Found a higher ring.
    msg.ringLvl = h;
    msg.startNode = msg.bestNode = localNode;
  } else if ( abs(localNode.numID - msg.numID) <
              abs(msg.bestNode.numID - msg.numID)) {
    // Found a better candidate for current ring.
    msg.bestNode = localNode;
  }

  // Forward along current ring.
  nbr = localNode.RouteTable[clockWise][msg.ringLvl];
  SendToNode(nbr);
}
```

Figure 6. *Algorithm to route by numeric ID in SkipNet*

3.4 Routing by Numeric ID

It is also possible to route messages efficiently to a given numeric ID. In brief, the routing operation begins by examining nodes in the level 0 (root) ring until a node is found whose numeric ID matches the destination numeric ID in the first digit. At this point the routing operation jumps up to this node's level 1 ring, which also contains the destination node. The routing operation then examines nodes in this level 1 ring until a node is found whose numeric ID matches the destination numeric ID in the second digit. As before, we know that this node's level 2 ring must also contain the destination node, and thus the routing operation proceeds in this level 2 ring.

This procedure repeats until we cannot make any more progress — we have reached a ring at some level h such that none of the nodes in that ring share $h + 1$ digits with the destination numeric ID. We must now deterministically choose one of the nodes in this ring to be the destination node. Our algorithm defines the destination node to be the node whose numeric ID is numerically closest to destination numeric ID amongst all nodes in this highest ring. Figure 6 presents this algorithm in pseudocode.

As an example, imagine that the numeric IDs in Figure 4 are 4 bits long and that node Z's ID is 1000 and node O's ID is 1001. If we want to route a message from node A to destination 1011 then A will first forward the message to node D because D is in ring 1. D will then forward the message to node O because O is in ring 10. O will forward the message to Z because it is not in ring 101. Z will forward the message onward around the ring (and hence back) to O for the same reason. Since none of the members of ring 10 belong to ring 101, node O

will be picked as the final message destination because its numeric ID is closest to 1011 of all ring 10 members.

The number of message hops when routing by numeric ID is $O(\log N)$ with high probability. For a proof see Section 8.3.

Some intuition for why SkipNet can support efficient routing by both name ID and numeric ID with the same data structure is illustrated in Figure 4. Note that the root ring, at the bottom, is sorted by name ID and, collectively, the top-level rings are sorted by numeric ID. For any given node, the SkipNet rings to which it belongs precisely form a Skip List. Thus efficient searches by name ID are possible. Furthermore, if you construct a trie on all nodes' numeric IDs, the nodes of the resulting trie would be in one-to-one correspondence with the SkipNet rings. This suggests that efficient searches by numeric ID are also possible.

3.5 Node Join and Departure

To join a SkipNet, a newcomer must first find the top-level ring that corresponds to the newcomer's numeric ID. This amounts to routing a message to the newcomer's numeric ID, as described in Section 3.4.

The newcomer then finds its neighbors in this top-level ring, using a search by name ID within this ring only. Starting from one of these neighbors, the newcomer searches for its name ID at the next lower level and thus finds its neighbors at this lower level. This process is repeated for each level until the newcomer reaches the root ring. For correctness, the existing nodes only point to the newcomer after it has joined the root ring; the newcomer then notifies its neighbors in each ring that it should be inserted next to them. Figure 7 presents this algorithm in pseudocode.

As an example, imagine inserting node O into the SkipNet of Figure 4. Node O initiates a search by numeric ID for its own ID (101) and the resulting insertion message ends up at node Z in ring 10 since that is the highest non-empty ring that shares a prefix with node O's numeric ID. Since Z is the only node in ring 10, Z concludes that it is both the clockwise and counter-clockwise neighbor of node O in this ring.

In order to find node O's neighbors in the next lower ring (ring 1), node Z forwards the insertion message to node D. Node D then concludes that D and V are the neighbors of node O in ring 1. Similarly, node D forwards the insertion message to node M in the root ring, who concludes that node O's level 0 neighbors must be M and T. The insertion message is returned to node O, who then instructs all of its neighbors to insert it into the rings.

The key observation for this algorithm's efficiency is that a newcomer searches for its neighbors at a certain level only after finding its neighbors at all higher levels.

```
InsertNode(nameID, numID) {
  msg = new JoinMessage();
  msg.operation = findTopLevelRing;
  RouteByNumericID(numID, msg);
}

DeliverMessage(msg) {
  ...
  else if (msg.operation == findTopLevelRing) {
    msg.ringLvl =
        CommonPrefix(localNode.numID, msg.numID);
    msg.ringNbrClockWise = new Node[msg.ringLvl];
    msg.ringNbrCClockWise = new Node[msg.ringLvl];
    msg.doInsertions = false;
    CollectRingInsertionNeighbors(msg);
  }
  else ...
}

// Invoked at every intermediate routing hop.
CollectRingInsertionNeighbors(msg) {
  if (msg.doInsertions) {
    InsertIntoRings(msg.ringNbrClockWise,
                    msg.ringNbrCClockWise);
    return;
  }

  while (msg.ringLvl >= 0) {
    nbr = localNode.RouteTable[clockWise][msg.ringLvl];
    if (LiesBetween(localNode.nameID, msg.nameID,
                    nbr.nameID, clockWise)) {
      // Found an insertion neighbor.
      msg.ringNbrClockWise[msg.ringLvl] = nbr;
      msg.ringNbrCClockWise[msg.ringLvl] = localNode;
      msg.ringLvl = msg.ringLvl-1;
    } else {
      // Keep looking
      SendToNode(msg, nbr);
      return;
    }
  }

  msg.doInsertions = true;
  SendToNode(msg, msg.joiningNode);
}
```

Figure 7. *Algorithm to insert a SkipNet node.*

As a result, the search by name ID will traverse only a few nodes within each ring to be joined: The range of nodes traversed at each level is limited to the range between the newcomer's neighbors at the next higher level. Therefore, with high probability, a node join in SkipNet will traverse $O(\log N)$ hops. For a proof see Section 8.4.

The basic observation in handling node departures is that SkipNet can route correctly as long as the root ring is maintained. All pointers but the root ring ones can be regarded as routing optimization hints, and thus are not necessary to maintain routing protocol correctness. Therefore, like Chord and Pastry, SkipNet maintains and repairs the upper-level ring memberships by means of a background repair process. In addition, when a node voluntarily departs from the SkipNet, it can proactively notify all of its neighbors to repair their pointers immediately.

To maintain the root ring correctly, each SkipNet node maintains a leaf set that points to additional nodes along the root ring, for redundancy. We describe the leaf set next.

3.6 Leaf Set

Every SkipNet node maintains a set of pointers to the $L/2$ nodes closest in name ID on the left side and similarly on the right side. We call this set of pointers a *leaf set*. Several previous peer-to-peer systems [34] incorporate a similar architectural feature; in Chord [39] they refer to this as a successor list.

These additional pointers in the root ring provide two benefits. First, the leaf set increases fault tolerance. If a search operation encounters a failed node, a node adjacent to the failed node will contain a leaf set pointer with a destination on the other side of the failed node, and so the search will eventually move past the failed node. Repair is also facilitated by repairing the root ring first, and recursively relying on the accuracy of lower rings to repair higher rings. Without a leaf set, it is not clear that higher level pointers (that point past a failed node) sufficiently enable repair. If two nodes fail, it may be that some node in the middle of them becomes invisible to other nodes looking for it using only higher level pointers. Additionally, in the node failure scenario of an organizational disconnect, the leaf set pointers on most nodes are more likely to remain intact than higher level pointers. The resiliency to node failure that leaf sets provide (with the exception of the organizational disconnect scenario) was also noted by [39].

A second benefit of the leaf set is to increase search performance by subtracting a noticeable additive constant from the required number of search hops. When a search message is within $L/2$ of its destination, the search message will be immediately forwarded to the destination. In our current implementation we use a leaf set of size $L = 16$, just as Pastry does.

3.7 Background Repair

SkipNet uses the leaf set to ensure with good probability that the neighbor pointers in the root ring point to the correct node. As is the case in Chord [38], this is all that is required to guarantee correct, if possibly inefficient, routing by name ID. For an intuitive argument of why this is true, suppose that some higher-level pointer does not point to the correct node, and that the search algorithm tries to use this pointer. There are two cases. In the first case, the incorrect pointer points further around the ring than the routing destination. In this case the pointer will not be used, as it goes past the destination. In the second case, the incorrect pointer points to a location between the current location and the destination. In this case the pointer can be safely followed and routing will proceed from wherever it points. The only potential loss is routing efficiency. In the worst case, correct routing will occur using the root ring.

Nonetheless, for *efficient* routing, it is important to ensure as much as possible that the other pointers are correct. SkipNet employs two background algorithms to detect and repair incorrect ring pointers.

The first of these algorithms builds upon the invariant that a correct set of ring pointers at level h can be used to build a correct set of pointers in the ring above it at level $h + 1$. Each node periodically routes a message a short distance around each ring that it belongs to, starting at level 0, verifying that the pointers in the ring above it point to the correct node and adjusting them if necessary. Once the pointers at level h have been verified, this algorithm iteratively verifies and repairs the pointers one level higher. At each level, verification and repair of a pointer requires only a constant amount of work in expectation.

The second of these algorithms performs local repairs to rings whose nodes may have been inconsistently inserted or whose members may have disappeared. In this algorithm nodes periodically contact their neighbors at each level saying "I believe that I am your left(right) neighbor at level h". If the neighbor agrees with this information no reply is necessary. If it doesn't, the neighbor replies saying who he believes his left(right) neighbor is, and a reconciliation is performed based upon this information to correct any local ring inconsistencies discovered.

4 Useful Locality Properties of SkipNet

In this section we discuss some of the useful locality properties that SkipNet is able to provide, and their consequences.

4.1 Content and Routing Path Locality

Given the basic structure of SkipNet, describing how SkipNet supports content and path locality is straightforward. Incorporating a node's name ID into a content name guarantees that the content will be hosted on that node. As an example, to store a document *doc-name* on the node *john.microsoft.com*, naming it *john.microsoft.com/doc-name* is sufficient.

SkipNet is oblivious to the naming convention used for nodes' name IDs. Our simulations and deployments of SkipNet use DNS names for name IDs, after suitably reversing the components of the DNS name. In this scheme, *john.microsoft.com* becomes *com.microsoft.john*, and thus all nodes within *microsoft.com* share the *com.microsoft* prefix in their name IDs. This yields path locality for organizations in which all nodes share a single DNS suffix (and hence share a single name ID prefix).

4.2 Constrained Load Balancing

As mentioned in the Introduction, SkipNet supports Constrained Load Balancing (CLB). To implement CLB, we divide a data object's name into two parts: a part that specifies the set of nodes over which DHT load balancing should be performed (the *CLB domain*) and a part that is used as input to the DHT's hash function (the *CLB suffix*). In SkipNet the special character '!' is used as a delimiter between the two parts of the name.

For example, suppose we stored a document using the name *msn.com/DataCenter!TopStories.html*. The CLB domain indicates that load balancing should occur over all nodes whose names begin with the prefix *msn.com/DataCenter*. The CLB suffix, *TopStories.html*, is used as input to the DHT hash function, and this determines the specific node within *msn.com/DataCenter* on which the document will be placed. Note that storing a document with CLB results in the document being placed on precisely one node within the CLB domain (although it would be possible to store replicas on other nodes). If numerous other documents were also stored in the *msn.com/DataCenter* CLB domain, then the documents would be uniformly distributed across all nodes in that domain.

To search for a data object that has been stored using CLB, we first search for any node within the CLB domain using search by name ID. To find the specific node within the domain that stores the data object, we perform a search by numeric ID within the CLB domain for the hash of the CLB suffix.

The search by name ID is unmodified from the description in Section 3.3, and takes $O(\log N)$ message hops. The search by numeric ID is constrained by a name ID prefix and thus at any level must effectively step through a doubly-linked list rather than a ring. Upon encountering the right boundary of the list (as determined by the name ID prefix boundary), the search must reverse direction in order to ensure that no node is overlooked. Reversing directions in this manner affects the performance of the search by numeric ID by at most a factor of two, and thus $O(\log N)$ message hops are required in total.

Note that both traditional system-wide DHT semantics as well as explicit content placement are special cases of constrained load balancing: system-wide DHT semantics are obtained by placing the '!' hashing delimiter at the beginning of a document name. Omission of the hashing delimiter and choosing the name of a data object to have a prefix that matches the name of a particular SkipNet node will result in that data object being placed on that SkipNet node.

Constrained load balancing can be performed over any naming subtree of the SkipNet but not over an arbitrary subset of the nodes of the overlay network. Another limitation is that CLB domain is encoded in the name of a data object. Thus, transparent remapping to a different load balancing domain is not possible.

4.3 Fault Tolerance

Previous studies [24, 28] indicate that network connectivity failures in the Internet today are due primarily to Border Gateway Protocol (BGP) misconfigurations and faults. Other hardware, software and human failures play a lesser role. As a result, node failures in overlay networks are not independent; instead, nodes belonging to the same organization or AS tend to fail together. We consider both correlated and independent failure cases in this section.

4.3.1 Independent Failures

SkipNet's tolerance to uncorrelated, independent failures is much the same as previous overlay designs' (e.g., Chord and Pastry), and is achieved through similar mechanisms. The key observation in failure recovery is that maintaining correct neighbor pointers in the root ring is enough to ensure correct functioning of the overlay. Since each node maintains a leaf set of 16 neighbors at level 0, the root ring pointers can be repaired by replacing them with the leaf set entries that point to the nearest live nodes following the failed node. The live nodes in the leaf set may be contacted to repopulate the leaf set fully.

As described in Section 3.7, SkipNet also employs a background stabilization mechanism that gradually updates all necessary routing table entries when a node fails. Any query to a live, reachable node will still succeed during this time; the stabilization mechanism simply restores optimal routing.

4.3.2 Failures along Organization Boundaries

In previous peer-to-peer overlay designs [32, 38, 34, 44], node placement in the overlay topology is determined by a randomly chosen numeric ID. As a result, nodes within a single organization are placed uniformly throughout the address space of the overlay. While a uniform distribution facilitates the $O(\log N)$ routing performance of the overlay it makes it difficult to control the effect of physical link failures on the overlay network. In particular, the failure of a inter-organizational network link may manifest itself as multiple, scattered link failures in the overlay. Indeed, it is possible for each node within a single organization that has lost connectivity to the Internet to become disconnected from the entire overlay and from all other nodes within the organization. Section 9.4 reports experimental results that confirm this observation.

Since SkipNet name IDs tend to encode organizational membership, and nodes with common name ID prefixes are contiguous in the overlay, failures along organization boundaries do not completely fragment the overlay, but instead result in ring segment partitions. Consequently, a significant fraction of routing table entries of nodes within the disconnected organization still point to live nodes within the same network partition. This property allows SkipNet to gracefully survive failures along organization boundaries. Furthermore, the disconnected organization's SkipNet segment can be efficiently re-merged with the external SkipNet when connectivity is restored, as described in Section 6.

4.4 Security

In this section, we discuss some security consequences of SkipNet's content and path locality properties. Recent work [3] on improving the security of peer-to-peer systems has focused on certification of node identifiers and the use of redundant routing paths. The security advantages of content and path locality depend on an access control mechanism for creation of name IDs. SkipNet does not directly provide this mechanism but rather assumes that it is provided at another layer. Our use of DNS names for name IDs does provide this mechanism: Arbitrary nodes cannot create global DNS names containing the suffix of a registered organization without its permission.

Path locality allows SkipNet to guarantee that messages between two machines within a single administrative domain that uses a single name ID prefix will never leave the administrative domain. Thus, these messages are not susceptible to traffic analysis or denial-of-service attacks by machines located outside of the administrative domain. Furthermore, traffic that is internal to an organization is not susceptible to a Sybil attack [10] originating from a foreign organization: Creating an unbounded number of nodes outside *microsoft.com* will not allow the attacker to see any traffic internal to *microsoft.com*, nor allow the attacker to usurp control over documents placed specifically within *microsoft.com*.

In Chord, the nodes belonging to an administrative domain (for example, *microsoft.com*) are uniformly dispersed throughout the overlay. Thus, intercepting a significant portion of the traffic to *microsoft.com* may require that an attacker create a large number of nodes. In SkipNet, the nodes belonging to an administrative domain form a contiguous segment of the overlay. Thus, an attacker might attempt to target *microsoft.com* by creating nodes (for example, *microsofa.com*) that are adjacent to the target domain. Thus a security disadvantage of SkipNet is that it may be possible to target traffic between an administrative domain and the outside world with fewer attacking nodes than would be necessary in systems such as Chord. We believe that susceptibility to these kinds of attacks is a small price to pay in return for the benefits provided by path and content locality.

4.5 Range Queries

Since SkipNet's design is based on and inspired by Skip Lists, it inherits their functionality and flexibility in supporting efficient range queries. In particular, since nodes and data are stored in name ID order, documents sharing common prefixes are stored over contiguous ring segments. Performing range queries in SkipNet is therefore equivalent to routing along the corresponding ring segment. Because our current focus is on SkipNet's architecture and locality properties, we do not discuss the use of range queries for implementing various higher-level data query operators further in this paper.

5 SkipNet Enhancements

This section presents several optimizations and enhancements to the basic SkipNet design.

5.1 Sparse and Dense Routing Tables

The basic SkipNet design may be modified in order to improve routing performance. Thus far in our discussions, SkipNet numeric IDs consist of 128 random binary digits. However, the random digits need not be binary. Indeed, Skip Lists using non-binary random digits are well-known [30].

We can also use non-binary random digits for the numeric IDs in SkipNet, which changes the ring structure depicted in Figure 4, the number of pointers stored per node, and the expected routing cost. We denote the number of different possibilities for a digit by k; in the binary digit case, $k = 2$. If $k = 3$, the root ring of SkipNet remains a single ring, but there are now three level 1 rings, nine level 2 rings, etc. As k increases, the total number of pointers in the R-Table will decrease. Because there are fewer pointers, it will take more routing hops to get to any particular node. For increasing values of k, the number of pointers decreases to $O(\log_k n)$ while the number of hops required for search increases to $O(k \log_k n)$. We call the routing table that results from this modification a *sparse R-Table* with parameter k.

It is also possible to build a *dense R-Table* by additionally storing $k - 1$ pointers to contiguous nodes at each level of the routing table and in both directions. In this case, the expected number of search hops decreases while the expected number of pointers at a node increases.

Increasing k makes the sparse R-Table sparser and the dense R-Table denser. The density parameter k and choice of sparse or dense construction can be used to control the amount routing state used by all SkipNet routing tables, and in Section 9 we examine the relationship between routing performance and the amount of routing table state maintained.

Our density parameter, k, bears some similarity to Pastry's density parameter, b. Pastry always generates binary numeric IDs but divides bits into groups of b. This is analogous to our scheme for choosing numeric IDs with $k = 2^b$.

Implementing node join and departure in the case of sparse R-Tables requires no modification to our previous algorithms. For dense R-Tables, the node join message must traverse and gather information about at least $k-1$ nodes in both directions in every ring containing the newcomer, before descending to the next ring. As before, node departure merely requires notifying every neighbor.

If $k = 2$, the sparse and dense constructions are identical. Increasing k makes the sparse R-Table sparser and the dense R-Table denser. Any given degree of sparsity/-density can be well-approximated by appropriate choice of k and either a sparse or a dense R-Table. Our implementation chooses $k = 8$ to achieve a good balance between state per node and routing performance.

5.2 Duplicate Pointer Elimination

Two nodes that are neighbors in a ring at level h may also be neighbors in a ring at level $h + 1$. In this case, these two nodes maintain "duplicate" pointers to each other at levels h and $h + 1$. Intuitively, routing tables with more distinct pointers yield better routing performance than tables with fewer distinct pointers, and hence duplicate pointers reduce the effectiveness of a routing table. Replacing a duplicate pointer with a suitable alternative, such as the following neighbor in the higher ring, improves routing performance by a moderate amount (our experiments indicate improvements typically around 25%). Routing table entries adjusted in this fashion can only be used when routing by name ID since they violate the invariant that a node point to its closest neighbor on a ring, which is required for correct routing by numeric ID.

5.3 Incorporating Network Proximity for Routing by Name ID

In SkipNet, a node's neighbors are determined by a random choice of ring memberships (i.e., numeric IDs) and by the ordering of name IDs within those rings. Accordingly, the SkipNet overlay is constructed without direct consideration of the physical network topology, potentially hurting routing performance. For example, when sending a message from the node *saturn.com/nodeA* to the node *chrysler.com/nodeB*, both in the USA, the message might get routed through the intermediate node *jaguar.com/nodeC* in the UK. This would result in a much longer path than if the message had been routed through another intermediate node in the USA.

To deal with this problem, we introduce a second routing table called the *P-Table*, which is short for proximity table. The goal of the P-Table is to maintain routing in $O(\log N)$ hops, while also ensuring that each hop has low cost in terms of network latency. Our P-Table design is inspired by Pastry's proximity-aware routing tables [4]. To incorporate network proximity into SkipNet, the key observation is that any node that is roughly the right distance away in name ID space can be used as an acceptable routing table entry that will maintain the underlying $O(\log N)$ routing performance. For example, it doesn't matter whether a P-Table entry at level 3 points to the node that is exactly 8 nodes away or to one that is 7 or 9 nodes away; statistically the number of forwarding hops that messages will take will end up being the same. However, if the 7th or 9th node is nearby in network distance then using it as the P-Table entry can yield substantially better routing performance. In fact, the P-Table entry at level h can be anywhere between 2^h and 2^{h+1} nodes away while maintaining $O(\log N)$ routing performance.

To construct its P-Table, a node needs to locate a set of candidate nodes that are close in terms of network distance and whose name IDs are appropriately distributed around the root ring. Unlike Chord and Pastry, in SkipNet it is difficult to estimate distance along the root ring simply by looking at a candidate node's name ID. We solve this problem by observing that a node's basic routing table (the R-Table) conveniently divides the root ring into intervals of exponentially increasing size. Thus, two pointers at adjacent levels in the R-Table provide the name ID boundaries of a contiguous interval along the root ring. Given a node, we examine these intervals to determine which P-Table entry it is a candidate for. We discover candidate nodes that are nearby using a recursive process: we start at a nearby *seed node* and discover other nearby nodes by querying the P-Table of the seed node. Finally, we determine that two nodes are near each other by estimating the round-trip latency between them.

The following section provides a detailed description of the algorithm that a SkipNet node uses to construct its P-Table. After the initial P-Table is constructed, SkipNet constantly tries to improve the quality of its P-Table entries, as well as adjust to node joins and departures, by means of a periodic stabilization algorithm. The periodic stabilization algorithm is very similar to the initial construction algorithm presented below. Finally, in Section 8.8 we argue that P-Table routing performance and P-Table construction are efficient.

5.3.1 P-Table Construction

Recall that the R-Table has only two configuration parameters: the value of k and either sparse or dense construction. The P-Table inherits these parameters from the R-Table upon which it is based. In certain cases it is possible to construct a P-Table with parameters that differ from the R-Table's by first constructing a temporary R-Table with the desired parameters. For example, if the R-Table is sparse, one may construct a dense P-Table by first constructing a temporary dense R-Table to use as input to the P-Table construction algorithm.

When a node joins SkipNet it first constructs its R-Table. P-Table construction is then initiated by copying the entries of the R-Table to a separate list, where they are sorted by name ID and then duplicate entries are eliminated. Duplicates and out-of-order entries can arise in this list due to the probabilistic nature of constructing the R-Table.

The joining node then constructs a P-Table join message that contains the sorted list of endpoints: a list of j nodes defining $j - 1$ intervals. The joining node sends this P-Table join message to a *seed node* – a node that should be nearby in terms of network distance.

Every node that receives a P-Table join message uses its own P-Table entries to fill in the intervals with "candidate" nodes. As a practical consideration, we limit the maximum number of candidates per interval to 10 in order to avoid accumulating too many nodes. After filling in any possible intervals, the node checks whether any of the intervals are still empty. If so, the node must forward the join message to another node in order to fill the remaining empty intervals.

Assuming that all intervals are initially empty, the expected number of hops required to find a candidate for the j^{th} farthest interval from the joining node is $O(j)$. Thus, in order to find candidates that are close to the joining node in terms of network proximity, we use the following strategy: Nodes that receive the join message use their own P-Table entries to forward the message towards the unfilled interval that is the *farthest* from the joining node. If all the intervals have at least one candidate, the node sends the completed join message back to the original joining node. The expected total number of hops to fill all intervals is $O(\log N)$.

When the original node receives its own join message, it chooses one candidate node per interval as its P-Table entry. The choice between candidate nodes is performed by estimating the network latency to each candidate and choosing the closest node.

We now summarize a few remaining key details of P-Table construction. Since SkipNet can route either clockwise or counter-clockwise, the P-Table contains intervals that cover the address space in both directions from the joining node. Thus two join messages are sent from the same starting node.

The effectiveness of P-Table routing entries is dependent to a great extent on finding nearby nodes. The basis of this process is finding a good *seed node*. In our simulator, we implemented two strategies for locating a seed node. Our first strategy uses global knowledge from the simulator topology model to find the closest node in the entire system. The second and more realistic strategy is that we choose the seed node at random, and then run the P-Table join algorithm twice. We use the first run of the P-Table join algorithm to locate a nearby seed, and the second run to construct a better P-table based on the nearby seed. Section 9.6 summarizes a performance evaluation of these two approaches.

For a real implementation, we make the following simple proposal: The seed node should be determined by estimating the network latency to all nodes in the leaf set and choosing the closest leaf set node. Since SkipNet name IDs incorporate naming locality, a node is likely to be close in terms of network proximity to the nodes in its leaf set. Thus the closest leaf set node is likely to be an excellent choice for a seed node.

After the initial P-Table is constructed, SkipNet constantly tries to improve the quality of its P-Table entries, and adjusts to node joins and departures, by means of a periodic stabilization algorithm. The P-Table is updated periodically so that the P-Table segment endpoints accurately reflect the distribution of name IDs in the SkipNet, which may change over time. The periodic mechanism used to update P-Table entries is very similar to the initial construction algorithm presented above. One key difference between the update mechanism and the initial construction mechanism is that for update, the current P-Table entries are considered as candidate nodes in addition to the candidates returned by the P-Table join message. The other difference is that for update, the seed node is chosen as the best candidate from the existing P-Table. Finally, the P-Table entries may also be incrementally updated as node joins and departures are discovered through ordinary message traffic.

5.4 Incorporating Network Proximity for Routing by Numeric ID

We add a third routing table, the C-Table, to incorporate network proximity when searching by numeric ID. Constrained Load Balancing (CLB), because it involves searches by both name ID and numeric ID, takes advantage of both the P-Table and the C-Table. Because search by numeric ID as part of a CLB search must stay within the CLB domain, C-Table entries that step outside the domain cannot be used. When such an entry is encountered, the CLB search must revert to using the R-Table.

The C-Table has identical functionality and design to the routing table that Pastry maintains [34]. The suggested parameter choice for Pastry's routing table is $b = 4$ (i.e. $k = 16$), while our implementation chooses $k = 8$, as mentioned in Section 5.1. As is the case with searching by numeric ID using the R-Table, and as is the case with Pastry, searching by numeric ID with the C-Table requires at most $O(\log N)$ message hops.

For concreteness, we describe the C-Table in the case that $k = 8$, although this description could be inferred from [34]. At each node the C-Table consists of a set of arrays of node pointers, one array per numeric ID digit, each array having an entry for each of the eight possible digit values. Each entry of the first array points to a node whose first numeric ID digit matches the array index value. Each entry of the second array points to a node whose first digit matches the first digit of the current node and whose second digit matches the array index value. This construction is repeated until we arrive at an empty array.

5.4.1 C-Table Construction and Update

C-Tables are constructed in the same manner as Pastry's routing tables, which is described in [4]. The key idea is: For each array in the C-Table, route to a nearby node with the necessary numeric ID prefix, obtaining its C-Table entries at that level, and then populate the joining node's array with those entries. Since several candidate nodes may be available for a particular table entry, the candidate with the best network proximity is selected. Section 8.8 shows that the cost of constructing a C-Table is $O(\log N)$ in terms of message traffic. As in Pastry, the C-Table is updated lazily, by means of a background stabilization algorithm.

We report experiments in Section 9.5 showing that use of the C-Table during CLB search reduces the RDP (Relative Delay Penalty). An adaptation of the argument presented in [4] for Pastry explains why this should be the case.

5.5 Failover Nodes

Many overlay networks, including SkipNet, have the following useful property: a message routed to a particular destination will arrive at the node that is *closest* to that destination. The definition of closest depends on the routing scheme used by the particular overlay. As an example, suppose that we store a file X by routing towards the destination X. If node Y is the closest node to that destination, then the file X will be stored on node Y. Next, suppose that node Y has failed and that we try to retrieve file X. All messages routed towards destination X will now arrive at some other node, say node Z. If node Z previously received a replica of file X from node Y, then node Z can successfully handle our request for this file. We say that node Z is a *failover node* for file X.

In order to implement this automatic failover functionality, an overlay network must provide a facility to determine failover nodes. In SkipNet, the identity of the failover nodes may depend on the actual routing destination used. In our example, even if files X_1 and X_2 are both stored on node Y, the respective failover nodes may be Z_1 and Z_2. If destination X_1 uses routing by name ID, it is easy to determine the failover nodes for this destination: they are simply the nodes in node Y's leaf set. However, if destination X_2 uses routing by numeric ID or constrained load-balancing, determining the failover nodes is more complicated, as explained next.

Recall that a message routed by numeric ID will arrive at the highest-level ring matching the destination ID. Among those nodes in that ring, the message will terminate at the node whose numeric ID is numerically closest to the destination ID. However, this node is not necessarily a neighbor of the node that is second-closest to the destination ID, since the ring is sorted by name ID, not by numeric ID. This shows that the failover node for a message routed by numeric ID need not be a neighbor of the original destination. A similar argument holds for messages routed by constrained load-balancing, except that the failover node must lie within the CLB domain used by the destination ID. For simplicity, the remainder of this discussion will regard routing by numeric ID as a special case of CLB routing where the CLB domain is the empty string.

Suppose that node Y is the destination node when routing to CLB destination $A!B$, where A is the CLB domain. We now describe our algorithm for determining a list of f failover nodes for this destination. This algorithm searches all rings containing node Y, looking for other nodes in CLB domain A. The search starts in the highest ring, since it contains the fewest nodes and hence requires the least work to search. If the neighbors of node Y in this ring are not in CLB domain A, it follows that node Y is the only member of CLB domain A in this ring, so the search drops to the next lowest ring. When the algorithm reaches a ring containing another node in CLB domain A, we enumerate over all such nodes, building a list of the f nodes whose numeric IDs are numerically closest to the hash of the CLB suffix B. If f such nodes are found, the search halts. Otherwise, the search drops again to the next lowest ring and continues searching for nodes until f nodes are found in total. A straightforward argument shows that the expected number of nodes traversed by this algorithm is $O(f)$.

Each SkipNet node Y supports a function *CLBNeighbor(*D*)* that returns a list of failover neighbors for the given CLB destination D, assuming that Y is the destination node for D. A simple implementation of this function could use the algorithm described in the previous paragraph to send a message around the SkipNet and build up the desired list of neighbors. A disadvantage of this simple implementation is that it must perform network traffic for each invocation of the function. Our implementation of *CLBNeighbor()* uses a different approach to avoid this disadvantage.

Instead of finding a list of failover nodes for each given destination, each node builds one list of failover nodes for each CLB domain to which it belongs. Specifically, for a node Y and CLB domain A, we find f nodes that are in CLB domain A, whose numeric IDs match Y's in as many digits as possible, and whose numeric IDs are numerically closest to but less than Y's. We also find a symmetric list of nodes within this CLB domain whose numeric IDs are greater than Y's. Each node stores a cached copy of these lists and periodically rebuilds them to maintain their accuracy as node arrive and depart.

These lists enable node Y to determine the failover nodes for any CLB destination D for which Y is the destination node, without incurring any network traffic. Suppose that the function *CLBNeighbor(D)* is invoked, where CLB destination D is in CLB domain A. Node Y combines the two lists of failover nodes for CLB domain A, sorts the nodes in the combined list by (1) the number of numeric ID digits that match destination D, and (2) the absolute difference in numeric ID value with destination D. The first f nodes in the sorted list are precisely the f failover nodes for destination D.

5.6 CLB Routing Loop Elimination

When routing a message with CLB[1], a particular node may appear multiple times on the routing path. We refer to such an occurrence as a *routing loop*. While the presence of routing loops does not affect the correctness of the CLB routing protocol, these loops may present difficulties for applications that layer on top of SkipNet. For example, applications such as Scribe [36, 5], Overlook [41] and FUSE [12] use the SkipNet routing paths to build additional data structures, such as reverse-path forwarding trees. Each of these applications would need complicated additional logic if it were modified to build its data structure in the presence of routing loops. A preferable design is for SkipNet to either eliminate these routing loops or hide them from applications.

Our implementation of SkipNet hides CLB routing loops by modifying the routing algorithms presented earlier. Recall that the CLB routing algorithm has two phases. The first phase simply routes by name ID towards the CLB domain. In the second phase, the message is routed by numeric ID within the CLB domain. Each node primarily uses its C-Table (Section 5.4) for this second phase, as long the required entry stays within the CLB domain. If the C-Table entry steps outside the CLB domain, we must route using the R-Table instead. To hide routing loops we modify the R-Table routing algorithm (Section 3.4) and keep track of several pieces of state within the routing message:

$start$ stores the identifier of the first node encountered within this ring.

dir indicates the current routing direction within this ring, clockwise or counter-clockwise.

$mode$ is either “tentative” or “commit”. In tentative mode applications are not notified as we traverse each node, so traversing a node multiple times is permissible. In commit mode applications are notified as we traverse each node, so routing paths may not have loops.

$best$ stores the identifier of the node in the current ring whose numeric ID is closest to the destination.

The message starts at the level 0 ring, initially in tentative mode and in the clockwise direction. The message hops linearly through this ring in the current direction, updating the $best$ variable as necessary, until one of the following cases applies.

Case 1: We find a node v that matches the destination ID in one more digit. In this case, the message then returns to the $start$ node, switches to commit mode, and hops towards node v in the appropriate direction. Hopping from $start$ to node v will not create any routing loops. Upon arriving at node v, we will continue this algorithm in the ring one level higher. Before doing so, we set $start$ to node v, dir to clockwise and $mode$ to tentative.

Case 2: The variable dir is clockwise, and we find a node that is outside the CLB domain. In this case, we change dir to counter-clockwise, and the message hops back through the ring in the new direction. The message will traverse nodes that it has already traversed, thereby creating a routing loop. However, since the message is in tentative node, all applications are oblivious to this routing loop.

Case 3: The variable dir is counter-clockwise, and we find a node v that is outside the CLB domain. In this case, we may conclude that no node within the CLB domain matches the destination ID in any more digits. Therefore the message returns to the $start$ node, switches to commit mode, and hops towards the $best$ node in the appropriate direction. Hopping from $start$ to the $best$ node will not create any routing loops.

This modified CLB routing algorithm performs no more than twice as many hops as the algorithm described in Section 3.4 and Section 4.2. Furthermore, since only the second phase is affected, and primarily the C-Table is used within this phase, typically much fewer than twice as many hops are performed.

5.7 CLB Routing Requires Consistency at All R-Table Levels

All peer-to-peer overlay networks implementing a DHT attempt to preserve consistency of the address space: distinct messages with identical destination IDs should be routed to the same destination node. In realistic deployment scenarios, it is not feasible to make this

[1] As in Section 5.5, we will regard routing by numeric ID as a special case of CLB routing where the CLB domain is the empty string.

an absolute guarantee. For example, a network partition might prevent a message on one side of the partition from reaching a node on the other side. Nevertheless, consistency of the address space is considered sufficiently desirable that most overlay networks employ leafsets to help ensure routing consistency.

Leafsets provide redundancy at the lowest level of the routing table, so that a limited number of node failures or departures can be tolerated without disrupting address space consistency. Because of this redundancy at the lowest level, many overlay networks consider knowledge of neighbors at other levels of the routing table as strictly an optimization – not something necessary for correctness. A consequence of this choice is that different strategies are often used to maintain the leafset than other levels of the routing table (for instance, different ping intervals).

Supporting CLB routing invalidates the assumptions behind these design choices. Consistency of CLB routing requires consistency at all levels of the R-Table – not just the lowest level. To see this, consider a CLB message that has arrived at its CLB domain. To continue routing the message using its CLB suffix, the message must be routed using routing table entries at increasingly higher level rings. (These correspond to nodes with an increasing number of digits in their numeric IDs matching the CLB suffix.) If at any point a node routing the message does not have a neighbor at the appropriate level ring, then the node will consider itself to be the destination.

The R-Table design outlined in Section 3 is clearly vulnerable to a single node failure above ring 0 disrupting consistency of the CLB address space. Perhaps more surprisingly, the C-Table algorithm outlined in Section 5.4.1 is not sufficient to address this vulnerability. This is because the selection of C-Table entries does not incorporate name ID constraints; a node might only have C-Table entries at some level that are outside a particular CLB domain (i.e., contiguous section of name ID space), even though other nodes within that CLB domain were also candidates for inclusion in the C-Table at this level. Therefore, even when routing a CLB message by its CLB suffix, it is not always possible to rely solely on the entries in the C-Table.

Fortunately, dense R-Tables (Section 5.1) effectively have a leafset at every level of the R-Table, and this provides consistency guarantees for CLB routing equivalent to what leafsets provide for name ID routing. When a CLB message is being routed by its CLB suffix at any particular R-Table level the dense R-Table will have $k-1$ pointers to neighbors matching this node in the appropriate number of numeric ID digits. Therefore, at least $k-1$ unrepaired node failures or departures in a single node's R-Table must have occurred for consistency of the CLB address space to be disrupted. An appropriate choice of k makes this no more likely than the chance that name ID routing is disrupted due to multiple simultaneous failures in the leafset.

5.8 Virtual Nodes

Economies of scale and the ability to multiplex hardware resources among distinct web sites have led to the emergence of hosting services in the World Wide Web. We anticipate a similar demand for hosting *virtual nodes* on a single hardware platform in peer-to-peer systems. In this section, we describe a scheme for scalably supporting virtual nodes within the SkipNet design. For ease of exposition, we describe only the changes to the R-Table; the corresponding changes to the P-Table and C-Table are obvious and hence omitted.

Nothing in the SkipNet design prevents multiple nodes from co-existing on a single machine; however, scalability becomes a concern as the number of virtual nodes increases. As shown in Section 8.2, a single SkipNet node's R-Table will probably contain roughly $\log N$ pointers. If a single physical machine hosts v virtual nodes, the total number of R-Table pointers for all virtual nodes is therefore roughly $v \log N$. As v increases, the periodic maintenance traffic required for each of those pointers poses a scalability concern. To alleviate this potential bottleneck, the present section describes a variation on the SkipNet design that reduces the expected number of pointers required for v virtual nodes to $O(v + \log n)$, while maintaining logarithmic expected path lengths for searches by name ID. In Section 8.6 we provide mathematical proofs for the performance of this virtual node scheme.

Although Skip Lists have comparable routing path lengths as SkipNet, Section 3 mentioned two fundamental drawbacks of Skip Lists as an overlay routing data structure:

- Nodes in a Skip List experience markedly disproportionate routing loads.
- Nodes in a Skip List have low average edge connectivity.

Our key insight is that neither of these two Skip List drawbacks apply to virtual nodes. In the context of virtual nodes, we desire that:

- A peer-to-peer system must avoid imposing a disproportionate amount of work on any given *physical machine*. It is less important that virtual nodes on a single physical machine do proportionate amounts of work.
- Similarly, each physical machine should have high edge connectivity. It is less important that virtual

nodes on a single physical machine have high edge connectivity.

In light of these revised objectives, we can relax the requirement that each virtual node has roughly $\log n$ pointers. Instead, we allow the number of pointers per virtual node to have a similar distribution to the number of pointers per data record in a Skip List. More precisely, all but one of the virtual nodes independently truncate their numeric IDs such that they have length $i \geq 0$ with probability $1/2^{i+1}$. The one remaining virtual node keeps its full-length numeric ID, in order to ensure that the physical machine has at least $\log n$ expected neighbors. As a result, in this scheme, the expected number of total pointers for a set of v virtual nodes is $2v + \log n + O(1)$.

When a virtual node routes a message, it can use any pointer in the R-Table of any co-located virtual node. Simply using the pointer that gets closest to the destination (without going past it) will maintain path locality and logarithmic expected routing performance.

The interaction between virtual nodes and DHT functionality is more complicated. DHT functionality involves searching for a given numeric ID. Search by numeric ID terminates when it reaches a ring from which it cannot go any higher; this is likely to occur in a relatively high-level ring. By construction, virtual nodes are likely only to be members of low-level rings, and thus they are likely not to shoulder an equal portion of the DHT storage burden. However, because at least one node per physical machine is not virtualized, the storage burden of the physical machine is no less than it would be without any virtual nodes.

5.9 NewNeighbour and NeighbourRemoved Functionality

Some kinds of applications built using overlay networks need to be informed when a node's routing state changes. For instance, if the overlay network is used to build a dynamic hash table (DHT) with content for a key hosted at the node that is the overlay routing destination for that key, then when a new node is added to the overlay adjacent to a key's former routing destination node, then the DHT needs to determine whether the new node is now the key's overlay routing destination, and if so, move the content there. Thus, the application must be informed when new neighbors of a node join the overlay.

Some overlay applications, such as the Overlook distributed name server [41], cache content along overlay routes to the primary copy of a piece of content. Thus, when the overlay path to a node changes, any cached content along routes that are no longer used should be invalidated. This is an example of the need for applications to be informed when nodes are removed from a node's routing table.

SkipNet provides two upcalls to its overlay clients to inform them of these conditions: *NewNeighbour()* and *NeighbourRemoved()*. Both pass up as an argument the NodeID of the node being added or removed, respectively, from the current node's routing tables.

5.10 GetNextHop Functionality

Another facility that SkipNet exposes to its applications is its routing mechanism. Applications can ask SkipNet what the next hop in the overlay path to a given overlay destination is using the *GetNextHop()* call. This is useful for at least two kinds of uses: for caching information about the overlay routes used for a particular piece of content, and for implementing iterative overlay routing. (Iterative overlay routing doesn't forward overlay messages from node to node until they reach their destination. Instead, the source node for a message first iteratively asks each node along the overlay route to its destination what the next overlay hop to the destination is, and only actually sends the message once the destination node has been determined. With iterative routing, all messages are actually sent directly from the source to the destination, using point-to-point communication.)

5.11 Simple Pings versus Smart Pings

SkipNet supports both the traditional ping mechanism for checking the liveness of routing table neighbors, and a novel *smart ping* mechanism to coalesce ping traffic. A node using simple pings just sends ping messages once per ping interval to all its neighbors, which reply to the sender. In contrast, a node using smart pings instead sends pings only to its immediate neighbors at each ring level in the R-Table. Each node a smart ping message reaches adds state to it indicating that it is alive, then forwards the message to the next neighbor along the ring. When the message has reached k nodes, it is sent back to the originating node, containing a list of all the nodes that it reached.

Smart pings reduce the overhead of pinging neighbors in a dense R-Table (Section 5.1) by roughly a factor of k without lengthening the interval between checks for the liveness of a given neighbor. Thus, although a dense R-Table may contain many more neighbors than a sparse R-Table, the overhead of maintaining the two tables using smart pings is roughly the same. Because P-Tables and C-Tables are updated significantly less frequently than the R-Table (recall that their accuracy is not required for routing consistency), significantly reducing the R-Table maintenance traffic substantially reduces the total amount of background traffic.

One reason we can coalesce ping traffic is that dense R-Tables are structured such that adjacent nodes at any level of the R-Table are interested in a mostly overlapping set of liveness queries. Smart pings exploit this redundancy.

Routing the message along the ring, rather than pinging each node individually, is sufficient to halve the ping traffic. The factor of k savings in message overhead results from the fact that each node along the ring that is contacted by this smart ping message will learn about the liveness of all the nodes that the message has already contacted. Although any given node will initiate smart ping messages only once per ping interval, the node will learn about the liveness of its neighbors much more often, $k-1$ times per ping interval. Because of this, a much longer ping interval can be used with smart pings while updating liveness information just as frequently as simple pings would on a short ping interval. The end result is that the total amount of message traffic used to check liveness is much less for the same frequency of liveness checking.

Although smart pings result in significant savings in maintenance traffic, they also raise new subtleties. Smart pings combine the function of pings with the function of fixing up the routing table. A smart ping may discover new neighbors while it is being routed because any node that notices a missing entry (based on its routing tables) can forward the smart ping message to this missing node. To ensure timely return of the smart ping message to the originating node, we insist that the smart ping return as soon as it has tried to reach $k-1$ nodes, regardless of whether it successfully reached every node in the originating node's routing table at this level.

If a node attempts to send a smart ping to a failed node the send failure will be noticed and the messages will be re-sent to a different (hopefully live) node. No annotation is added to the message saying that that the node could not be contacted. This means that other nodes the ping reaches may try to independently contact the failed node if their routing tables indicate that it should have been between the node originating the ping and the present recipient – communication that will also fail if the node is actually down. However, we decided that this was preferable to the alternative of marking the node as having failed in the message. Marking the node as dead in the message posed the danger that an intransitive failure between any two nodes would lead every other node to believe that one of the nodes was dead, even though it was both alive and currently reachable. An example of how this might occur would be if A and C could communicate, B and C could communicate, but A and B could not. Suppose A tried to ping B, timed out, and then marked B as having failed. C might then receive the smart ping message from A and remove B from its routing table, even though B was both alive and reachable. Our concern with this intransitive failure scenario led us to choose the alternative design where multiple failed sends might be caused by a single failed node.

5.12 Ping Piggybacking

SkipNet nodes periodically ping their routing table neighbors to ensure that they are still reachable. Given that applications may utilize knowledge of overlay neighbor relationships (which can be gained by observing message routes, via the *GetNextHop()* call, or via the *NewNeighbour()* and *NeighbourRemoved()* calls), it is also useful to allow them to use the ping messages sent to overlay neighbors to not just check neighbor liveness, but to also check application-level invariants between overlay neighbors. SkipNet provides a mechanism to allow applications to add information to the period SkipNet ping messages called *ping piggybacking* that lets applications accomplish this. Ping piggybacking eliminates the need for a second level of application-level ping messages to neighbors, reducing the total amount of overlay and overlay application maintenance traffic that would otherwise be needed by eliminating the redundant layer of ping messages.

SkipNet exports ping piggybacking to its applications by allowing them to register upcalls that are invoked when ping messages will be sent and when they are received. The send upcall is told which node the ping is addressed to and allows the application to add data to be sent in the ping message, along with a key identifying the kind of data being piggybacked. Upon ping receipt, SkipNet checks whether each SkipNet application has registered a receive upcalls under the keys received, and if so, invokes those upcalls in the applications. Thus, ping piggybacking can be used to piggyback arbitrary application-level messages onto the periodic SkipNet node liveness-checking pings. The FUSE system [12], for instance, uses ping piggybacking to perform delegated liveness checking, allowing the liveness status of arbitrary numbers of groups of nodes to be checked in a manner that requires only a constant amount of liveness-checking traffic, independent of the number of groups.

Ping piggybacking is straightforward for R-Table routes whether we are employing smart pings or simple pings. The neighbor relationship in the R-Table is symmetric, and hence if A pings B regularly, B will ping A regularly. In contrast, P-Table and C-Table construction does not necessarily lead to a symmetric neighbor relationship. For example, because the neighbor relationship is not symmetric, A contacting B as part of A's P-Table reconstruction does not imply that B will contact A as part of B's P-Table reconstruction. This is one reason we do not support ping piggybacking for P-Table or C-Table neighbors. We wanted to support ping piggybacking behavior where applications could count on two neighbors both being able to initiate pings.

A second reason for not supporting ping piggybacking for P-Table or C-Table neighbors is that the rest of the SkipNet design treats the P-Table and C-Table as routing optimizations – not routing information that needs to be maintained with the same reliability as the R-Table. By restricting ping piggybacking to the R-Table, we were able to utilize the R-Table's consistently shorter ping interval.

Lastly, P-Table and C-Table construction is done in a batch update fashion that aggressively looks for new neighbors that will have better latency characteristics than the current set of neighbors. Ping piggybacking seems less useful when the set of neighbors is allowed to fluctuate significantly even in the absence of failure events.

5.13 Transparency of Overlay State

Mogul et al. [27] recently argued that state maintained by network protocol implementations should also be made available to clients of those protocols. Analogously, in our experience building overlay applications, nearly all internal state maintained by overlay networks is also useful to the overlay applications themselves. Indeed, one the aspects of the SkipNet design that we believe that other overlays should adopt is to export as much of the overlay's state to its applications as possible.

In summary, some of the facilities that SkipNet provides to make its overlay state visible to and usable by its applications are the *CLBNeighbour()* call, the *GetNextHop()* call, the via the *NewNeighbour()* and *NeighbourRemoved()* calls), and *ping piggybacking*. Many of them are actually implemented as overlay-independent facilities (and all of them could be). For instance, a C# generic overlay base class from which the SkipNet overlay class is derived is where the *NewNeighbour()* and *NeighbourRemoved()* APIs are actually defined. Our Pastry and Chord implementations also provide implementations of them. This means that overlay-independent overlay applications could use the same code, for instance, to handle changes in either Pastry's or SkipNet's neighbor sets, depending upon which overlay they were being run on.

6 Recovery from Organizational Disconnects

In this section, we characterize the behavior of SkipNet with respect to a common failure mode: when organizations become disconnected from the Internet. We describe and evaluate the recovery algorithms used to repair the SkipNet overlay when such failures occur. One key benefit of SkipNet's locality properties is graceful degradation in response to disconnection – one of the more common forms of Internet failure, which can be caused by router misconfigurations and link and router faults [24, 28]. Because SkipNet orders nodes according to their names, and assuming that organizations assign node names with one or a few organizational prefixes, an organization's nodes are naturally arranged into a few contiguous overlay segments. Should an organization become disconnected, its segments remain internally well-connected and intra-segment traffic can be routed with the same $O(\log M)$ hop efficiency as before, where M is the maximum number of nodes in any segment.

By repairing only a few key routing pointers between the "edge" nodes of each segment, the entire organization can be connected into a separate SkipNet that can route traffic with similar efficiency: Intra-segment traffic is still routed in $O(\log M)$ hops, but inter-segment traffic may initially require $O(\log M)$ hops for *every* segment that it traverses. In total, $O(S \log M)$ hops may be required for inter-segment traffic, where S is the number of segments in the organization.

A background process repairs the additional routing pointers, thereby eliminating the cross-segment penalty. SkipNet's structure enables this repair process to be done in a manner that avoids unnecessary duplication of work. When the organization reconnects to the Internet, these same repair operations can be used to merge the organization's segments back into the global SkipNet.

In contrast, most previous scalable, peer-to-peer overlay designs [32, 34, 38, 44] place nodes in the overlay topology according to a unique random numeric ID only. Disconnection of an organization in most of these systems will result in its nodes fragmenting into many disjoint overlay pieces. During the time that these fragments are reforming into a single overlay, network routing efficiency may be poor or unbalanced, or may even fail.

6.1 Recovery Algorithms

When an organization is disconnected from the Internet, its nodes will be able to communicate with each other over IP but will not be able to communicate with nodes outside the organization. If the organization's nodes' names employ only a few organizational prefixes then the nodes are mostly contiguous in SkipNet, and hence the global SkipNet will partition itself into several disjoint, but internally well-connected, segments. This is illustrated in Figure 8.

Because of SkipNet's path locality property, message traffic within each segment will be unaffected by disconnection and will continue to be routed with $O(\log M)$ efficiency, where M is the number of nodes within the segment. Assuming that the disconnecting organization constitutes a small fraction of the global SkipNet, cross-segment traffic among the global portions of the SkipNet will also remain largely unaffected because most cross-segment pointers among global segments will re-

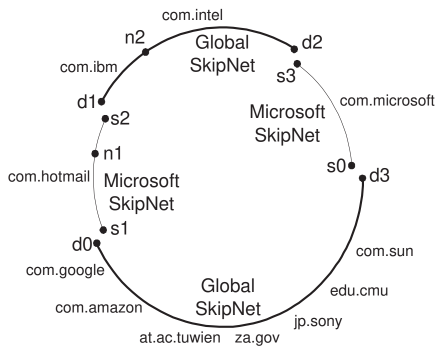

Figure 8. *Two partitioned SkipNets to be merged.*

main valid. This will not be true for the segments of the disconnected organization.

Gracefully handling a partition in the underlying IP network has two aspects: continuing to provide internal connectivity for the duration of the partition, and efficiently repairing the overlay when the underlying IP network partition heals. Maintaining internal connectivity of the overlay requires that communications be possible both within each overlay segment and across segments that still have IP connectivity to each other. Repairing the overlay when the partition heals involves reestablishing communications between overlay segments that were formerly unreachable by IP. Thus, the primary repair task after both disconnection and reconnection is the merging of overlay segments.

The algorithms employed in both the disconnection and reconnection cases are very similar: SkipNet segments must discover each other and then be merged together. For the disconnect case, the organization segments are merged into a separate SkipNet and the global segments are merged to reform the global SkipNet. For the reconnect case, all segments from the two separate SkipNets are merged into a single SkipNet.

6.2 Discovery Techniques

When an organization disconnects from the Internet there is no guarantee that the resulting non-contiguous segments will have pointers into each other. Therefore its segments may not be able to find each other using only SkipNet pointers. To solve this discovery problem we assume that organizations will divide their nodes into a relatively small number of name segments and that they designate some number of nodes in each segment as "well-known". For instance, Microsoft might maintain well-known members of segments with name prefixes *microsoft.com*, *hotmail.com*, *xbox.jp*, etc. Each node in an organization maintains a list of these well-known nodes and uses them as contact points between the various overlay segments.

When an organization reconnects to the Internet, the

```
ConnectRootLevel(n1, n2) {
  edgeNodes = GatherEdgeNodeInfo(n1, n2, null)
  Connect edge node pairs.
}

GatherEdgeNodeInfo(n1, n2, msg) {
  n2 routes msg to n1 in its SkipNet.
  Msg will arrive at d1.
  d1 appends d1 and next neighbor, d0, to msg contents.
  d1 sends msg directly to n1 over IP.
  n1 routes msg to d0 in its SkipNet.
  Msg will arrive at s1.
  if (memberOf(s0, msg contents))  // => all segments
    return msg contents            //    traversed
  else  // => Message needs to discover more edge nodes
    s1 appends s1 and next neighbor, s0, to msg contents.
    return GatherEdgeNodeInfo(s0, d0, msg)
}
```

Figure 9. *SkipNet root ring connection algorithm.*

organizational and global SkipNets discover each other through their segment edge nodes. Since each node maintains a leaf set, if a node discovers that one side of its leaf set, but not the other, is completely unreachable then it concludes that a disconnect event has occurred and that it is an edge node of a segment. These edge nodes keep track of their unreachable leaf set pointers and periodically ping them for reachability; should a pointer become reachable, the node initiates the merge process. Note that merging two previously independent SkipNets together—for example, when a new organization joins the system—is functionally equivalent to reconnecting a previously connected one, except that an alternate means of discovery is needed.

6.3 Connecting Root Ring Segments

The segment merge process is comprised of two steps: repair of the root ring pointers and repair of the pointers for all higher-level rings. The first step can be done quickly, as it only involves repair of the root ring pointers of the edge nodes of each segment. Once the first step has been done it will be possible to route messages correctly among nodes in different segments and to do so with $O(S \log M)$ efficiency, where S is the total number of segments and M is the maximum number of nodes within a segment. As a consequence, the second, more expensive step can be done as a background task, as described in Section 6.4.

The key idea for connecting SkipNet root ring segments is to discover the relevant edge nodes by having a node in one segment route a message towards the name ID of a node in the other segment. This message will be routed to the edge node in the first segment that is lexicographically nearest to the other node's name ID. By repeating this process one can enumerate all edge nodes and hence all segments.

The actual inter-segment pointer updates are then done as a single atomic operation among the segment edge nodes, using distributed two-phase commit. This avoids routing inconsistencies where a message destined for a specific node on one segment inadvertently ends up at a different node in another overlay segment because

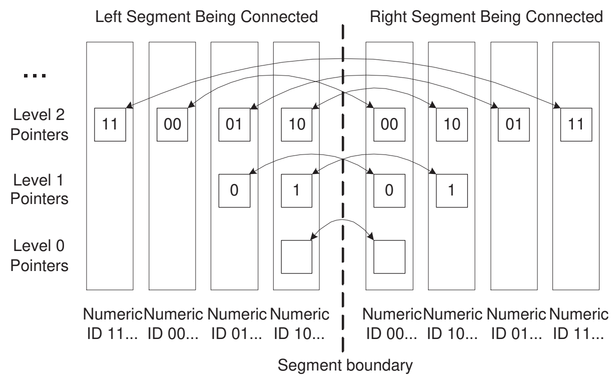

Figure 10. *Nodes whose pointers have been repaired at the boundary of two SkipNet segments.*

```
// Called initially with level h=0 at node
// to the left of the merge point
PostMergeRepair(h) {
  Find closest node to left whose numeric ID matches
      mine in the first h bits and whose ID differs from
      mine in the next bit, by following level h
      pointers to the left.
  On my node:
    cont = FixMyRightPointer(h+1)
    if (cont) PostMergeRepair(h+1)
  In parallel, on the other node:
    cont2 = FixMyRightPointer(h+1)
    if (cont) PostMergeRepair(h+1)
}

FixMyRightPointer(h) {
  Search right using level h-1 pointers until a node is
      found that matches my numeric id in h bits.
  Connect our level h pointers.
  if (pointers are already equal)
    return false
  else
    return true
}
```

Figure 11. *Level h ring repair algorithm for a single inter-segment boundary.*

the segments to be merged do not yet form a fully connected root ring.

To illustrate, Figure 8 shows two SkipNets to be merged, a Microsoft SkipNet and a global SkipNet, each containing two different name segments. Suppose that node $n1$ knows of node $n2$'s existence. Node $n1$ will send a message to node $n2$ (over IP) asking it to route a search message towards $n1$ in the global SkipNet. $n2$'s message will end up at node $d1$ and, furthermore, $d1$'s neighbor on the global SkipNet will be $d0$. $d1$ sends a reply to $n1$ (over IP) telling it about $d0$ and $d1$. $n1$ routes a search message towards $d0$ on the Microsoft SkipNet to discover $s1$ and $s0$ in the same manner. The procedure is iteratively invoked using $s0$ and $d0$ to gain information about $s2$, $s3$, $d2$, and $d3$. Figure 9 presents the algorithm in pseudo-code.

Immediately following root ring connection, messages sent to cross-segment destinations will be routed efficiently. Cross-segment messages will be routed to the edge of each segment they traverse and will then hop to the next segment using the root ring pointer connecting the segments. This leads to $O(S \log M)$ routing efficiency. When an organization reconnects its fully repaired SkipNet root ring to the global one, traffic destined for nodes external to the organization will be routed in $O(\log M)$ hops to an edge node of the organization's SkipNet. The root ring pointer connecting the two SkipNets will be traversed and then $O(\log N)$ hops will be needed to route traffic within the global SkipNet. Note that traffic that does not have to cross between the two SkipNets will not incur this routing penalty.

6.4 Repairing Routing Pointers following Root Ring Connection

Once the root ring connection phase has completed we can update all remaining pointers that need repair using a background task. We present here an algorithm for doing this that avoids unnecessary duplication of work through appropriate ordering of repair activities.

The key idea is that we recursively repair pointers at one level by using correct pointers at the level below to find the desired nodes in each segment. All pointers at one level must be repaired across a segment boundary before repair of a higher level can be initiated. To illustrate, consider Figure 10, which depicts a single boundary between two SkipNet segments after pointers have been repaired. Figure 11 presents an algorithm in pseudo-code for repairing pointers above the root ring across a single boundary. We begin by discussing the single boundary case, and later we extend our algorithm to handle the multiple boundary case.

Assume that the root ring pointers have already been correctly connected. There are two sets of two pointers to connect between the segments at level 1: the ones for the routing ring labeled 0 and the ones for the routing ring labeled 1 (see Figure 4). We can repair the level 1 ring labeled 0 by traversing the root (level 0) ring from one of the edge nodes until we find nodes in each segment belonging to the ring labeled 0. The same procedure is followed to correctly connect the level 1 ring labeled 1. After the level 1 rings, we use the same approach to repair the four level 2 rings.

Because rings at higher levels are nested within rings at lower levels, repair of a ring at level $h+1$ can be initiated by one of the nodes that had its pointer repaired for the enclosing ring at level h. A repair at level $h+1$ is unnecessary if the level h ring (a) contains only a single member or (b) does not have an inter-segment pointer that required repair. The latter termination condition implies that most rings—and hence most nodes—in the global SkipNet will not, in fact, need to be examined for potential repair.

The total work involved in this repair algorithm is $O(M \log(N/M))$, where M is the size of the disconnecting/reconnecting SkipNet segment and N is the size of the external SkipNet. Note that rings at level $h+1$ can be repaired in parallel once their enclosing rings at level h have been repaired across all segment boundaries. Thus, the repair process for a given segment boundary parallelizes to the extent supported by the underlying network infrastructure. We provide a theoretical analysis of the total work and total time to complete repair in Section 8.7.

To repair multiple segment boundaries, we simply call the algorithm described above once for each segment boundary. In the current implementation, we perform this process iteratively, waiting for the repair operation to complete on one boundary before initiating the repair at the next boundary. In future work, we plan to investigate initiating the segment repair operations in parallel — the open question is how to avoid repair operations from different boundaries interfering with each other.

6.5 Repairing P-Table and C-Table Entries

In normal operation, both a node's P-Table and the C-Table entries are updated periodically using information gathered from the node's R-table. Once the R-table repair algorithms above have run then these periodic update processes will likewise repair the node's P-Table an C-Table with no resulting increase in maintenance traffic.

7 Design Alternatives

SkipNet's locality properties can be obtained to a limited degree by suitable extensions to existing overlay network designs. We explore several such extensions in this section. However, none of these design alternatives provides all of SkipNet's locality advantages.

The space of alternative design choices can be divided into three cases: Rely on the inherent locality properties of the underlying IP network and DNS naming instead of using an overlay network; use a single overlay network—possibly augmented—that supports locality properties; or use multiple overlay networks that provide locality by spanning different sets of member nodes.

7.1 IP routing and DNS naming

A simple alternative to SkipNet's content placement scheme is to route directly using IP after a DNS lookup. This approach would also arguably provide path locality since most organizations structure their internal networks in a path-local manner. However, discarding the overlay network also discards all of its advantages, including:

- Implicit support for DHTs, and in the case of SkipNet, support for constrained load balancing.
- Seamless reassignment of traffic to well-defined alternative nodes in the presence of node failures.
- Better support for higher level abstractions, such as application-level multicast [6, 36, 33] and load-aware replication [41].
- The ability to reach named destinations independent of the availability of the DNS name lookup service.

7.2 Single Overlay Networks

Existing overlays are based on DHTs and depend on random assignment of node IDs in order to obtain a uniform distribution of nodes within their address spaces. To support explicit content placement onto a particular node requires changing either node or data naming. One could name a node with the hash of the data object's name, or some portion of its name. This scheme effectively virtualizes overlay nodes so that each node joins the overlay once per data object.

The drawback of this solution is that separate routing tables are required for each local data object. This will result in a prohibitive cost whenever a single node needs to store more than a few hundred data objects due to the network traffic overhead of building and maintaining large numbers of routing table entries.

Alternatively, one could change object names to use a two-part naming scheme, much like in SkipNet, where content names consist of unique node addresses concatenated to local, node-relative, names. Although this approach supports content placement, it does not support *guaranteed* path locality nor constrained load balancing (including continued content locality in the event of failover to a neighbor node).

One might imagine providing path locality by adding routing constraints to messages, so that messages are not allowed to be forwarded outside of a given organizational boundary. Unfortunately, such constraints would also prevent routing from being consistent. That is, messages sent to the same destination ID from two different source nodes would not be guaranteed to end up at the same destination node.

Overlay networks such as Pastry can partially mitigate the path locality problem using their support for network proximity [4]. However, Pastry's network proximity support depends on having a nearby node to use as a "seed" node when joining an overlay. If the nearby node is not within the same organization as the joining node, Pastry might not be able to provide good, let alone guaranteed, path locality. This problem is exacerbated for organizations that consist of multiple separate "islands" of nodes that are far apart in terms of network distance. In contrast, SkipNet is able to guarantee path locality, even across organizations that consist of separate clusters of nodes, as long as they are contiguous in name ID space.

An alternative to virtualizing node names would be to lengthen node IDs and partition them into separate, concatenated parts. For example, in a two-part scheme, node names would consist of two concatenated IDs and content names would also consist of two parts: a numeric ID value and a string name. The numeric ID would map to the first part of an overlay ID while the hash of the string name would map to the second part. The result is a static form of constrained load balancing: The numeric ID of a data object's name selects the DHT formed by all nodes sharing the same numeric ID and the string name determines which node to map to within the selected DHT. Furthermore, combining this approach with node virtualization provides explicit content placement.

This approach comes close to providing the same locality semantics as SkipNet: it provides explicit content placement, a static form of constrained load balancing, and path locality within each numeric ID domain. The major drawbacks of this approach are that the granularity of the hierarchy is frozen at the time of overlay creation by human decision; every layer of the hierarchy incurs an additional cost in the length of the numeric ID and in the size of the routing table that must be maintained; and the path locality guarantee is only with respect to boundaries in the static hierarchy.

7.3 Multiple Overlay Networks

Instead of using a single DHT-based overlay one might consider employing multiple overlays with different memberships. These multiple overlays can be arranged either as a static set of networks reflecting the desired locality requirements or as a dynamic set of overlays reflecting the participation of nodes in particular applications. In the static overlay case, a node could belong to just one of several alternative overlays, or belong to multiple overlays at different levels of a hierarchy.

In the case where each node belongs to only one of several overlays, one could imagine accessing other overlays by gateways. These gateways need not be a single point of failure if we give the backup gateway an appropriate neighboring numeric ID. One could either route directly to well-known gateways, or the gateways could organize an overlay network amongst themselves (imagine a overlay network of overlay networks). In either case, inter-domain routing requires serial traversal of the domain hierarchy, resulting in potentially large latencies when routing between domains.

If instead each node belonged to multiple overlays (for example, to a global overlay, an organization-wide overlay, and perhaps also a divisional or building-wide overlay), the associated overhead would correspondingly grow. Explicit content placement would still require extension of the overlay design. Furthermore, in this scheme, access to data that is constrained load balanced within a single overlay is not readily accessible to clients outside that overlay network, although it could be made so by introducing gateways in this design.

A final design alternative involving multiple overlays is to define an overlay network per application. This lets applications dynamically define the set of participating nodes, and thus ensure that application specific messages stay within this overlay. It does not provide any notion of locality within a subset of the overlay, and therefore fails to provide much of SkipNet's functionality, such as constrained load balancing.

In contrast, SkipNet provides explicit content placement, allows clients to dynamically define new DHTs over any name prefix scope, and guarantees path locality within any shared name prefix, all within a single shared infrastructure.

8 Analysis of SkipNet

In this section we analyze various properties of and costs of operations in SkipNet. Each subsection begins with a summary of the main results followed by a brief, intuitive explanation. The remainder of each subsection proves the results formally.

8.1 Searching by Name ID

Searches by name ID in a dense SkipNet take $O(\log_k N)$ hops in expectation, and $O(k \log_k N)$ hops in a sparse SkipNet. Furthermore, these bounds hold with high probability. (Refer to Section 5.1 for the definition of 'sparse', 'dense', and parameter k; the basic SkipNet design described in Section 3 is a sparse SkipNet with $k = 2$). We formally prove these results in Theorem 8.5 and Theorem 8.2. Intuitively, searches in SkipNet require this many hops for the same reason that Skip List searches do: every node's pointers are approximately exponentially distributed, and hence there will most likely be some pointer that halves the remaining distance to the destination. A dense SkipNet maintains roughly a factor of k more pointers and makes roughly a factor of k more progress on every hop.

For the formal analysis, we will consider a sparse R-Table first, and then extend our analysis to the dense R-Table. It will be helpful to have the following definitions: The node from which the search operation begins is called the *source* node and the node at which the search operation terminates is called the *destination* node. The search operation visits a sequence of nodes, until the destination node is found; this sequence is called the *search path*. Each step along the search path from one node to the next is called a *hop*. Throughout this subsection we will refer to nodes by their name IDs, and we will denote the name ID of the source by s, and the name ID of the destination by d.

The rings to which s belongs induce a Skip List structure on all nodes, with s at the head. To analyze the search path in SkipNet, we consider the path that the Skip List search algorithm would use on the induced Skip List; we then prove that the SkipNet search path is no bigger the Skip List search path. Let P be the SkipNet search path from s to d using a sparse R-Table. Let Q be the path that the Skip List search algorithm would use in the Skip List induced by node s. Note that both search paths begin with s and end with d, and all the nodes in the paths lie between s and d. To see that P and Q need not be identical, note that the levels of the pointers traversed in a Skip List search path are monotonically non-increasing; in a SkipNet search path this is not necessarily true.

To characterize the paths P and Q, it will be helpful to let $F(x, y)$ denote the longest common prefix in x and y's numeric IDs. The following useful identities follow immediately from the definition of F:

$$F(x, y) = F(y, x) \tag{1}$$

$$F(x, y) < F(y, z) \Rightarrow F(x, z) = F(x, y) \tag{2}$$

$$F(x, y) \leq F(y, z) \Rightarrow F(x, z) \geq F(x, y) \tag{3}$$

$$F(x, y) > f,\ F(x, z) > f \Rightarrow F(y, z) > f \tag{4}$$

The Skip List search path, Q, includes every node x between s and d such that no node closer to d has more digits in common with s. Formally, Q contains $x \in [s, d]$ if and only if $\nexists y \in [x, d]$ such that $F(s, y) > F(s, x)$.

The SkipNet search path P contains every node between s and d such that no node closer to d has more digits in common with *the previous node on the path*. This uniquely defines P by specifying the nodes in order; the node following s is uniquely defined, and this uniquely defines the subsequent node, etc. Formally, $x \in [s, d]$ immediately follows w in P if and only if it is the closest node following w such that $\nexists y \in [x, d]$ satisfying $F(w, y) > F(w, x)$.

Lemma 8.1. *Let P be the SkipNet search path from s to d using a sparse R-Table and let Q be the path that the Skip List search algorithm would use in the induced Skip List. Then P is a subsequence of Q. That is, every node encountered in the SkipNet search is also encountered in the Skip List search.*

Proof: Suppose for the purpose of showing a contradiction that some node x in P does not appear in Q. Let x be the first such node. Clearly $x \neq s$ because s must appear in both P and Q. Let w denote x's predecessor in P; since $x \neq s$, x is not the first node in P and so w is indeed well-defined. Node w must belong to Q because x was the first node in P that is not in Q.

We first consider the case that $F(s, x) > F(s, w)$, i.e., x shares more digits with s than w does. We show that this implies that w is not in Q, the Skip List search path (a contradiction). Referring back to the Skip List search path invariant, $x \in [w, d]$ plays the role of y, thereby showing that w is not in Q.

We next consider the case that $F(s, x) = F(s, w)$, i.e., x shares equally many digits with s as w does. We show that this implies that x is in Q, the Skip List search path (a contradiction). Referring back to the Skip List search path invariant, $\nexists y \in [w, d]$ such that $F(s, y) > F(s, w)$. Combining the assumption of this case, $F(s, w) = F(s, x)$, with $[x, d] \subset [w, d]$, we have that $\nexists y \in [x, d]$ such that $F(s, y) > F(s, x)$, and therefore x is in Q.

We consider the last case $F(s, x) < F(s, w)$, i.e., x shares fewer digits with s than w does. We show that this implies that x is not in P, the SkipNet search path (a contradiction). Applying Identity 2 yields that $F(s, x) = F(w, x)$, i.e., x shares the same number of digits with w as it does with s. By the assumption that x is not in Q, the Skip List search path, there exists $y \in [x, d]$ satisfying $F(s, y) > F(s, x)$. Combining $F(s, y) > F(s, x)$ with the case assumption, $F(s, w) > F(s, x)$ and applying Identity 4 yields $F(w, y) > F(s, x)$. Since $F(s, x) = F(w, x)$, this y also satisfies $F(w, y) > F(w, x)$. Combining this with $y \in [x, d]$ implies that y violates the SkipNet search path invariant for x; x is not in P. ■

A consequence of Lemma 8.1 is that the length of the Skip List search path bounds the length of the SkipNet search path. In the following theorem, we prove a bound on the length of the SkipNet search path as a function of D, the distance between the source s and the destination d, by analyzing the Skip List search path. Note that our high-probability result holds for arbitrary values of D; to the best of our knowledge, analyses of Skip Lists and of other overlay networks [39, 34] prove bounds that hold with high probability for large N. Because of the SkipNet design, we expect that $D \ll N$ will be a common case. There is no reason to expect this in Skip Lists or other overlay networks.

It will be convenient to define some standard probability distribution functions. Let $f_{n,1/k}(g)$ be the distribution function of the binomial distribution: if each experiment succeeds with probability $1/k$, then $f_{n,1/k}(g)$ is the probability that we see exactly g successes after n experiments. Let $F_{n,1/k}(g)$ be the cumulative distribution function of the binomial distribution: $F_{n,1/k}(g)$ is the probability that we see at most g successes after n experiments. Let $G_{g,1/k}(n)$ be the cumulative distribution function of the *negative* binomial distribution: $G_{g,1/k}(n)$ is the probability that we see g successes after at most n experiments.

We use the following two identities below:

$$G_{g,\frac{1}{k}}(n) = 1 - F_{n,\frac{1}{k}}(g-1) \tag{5}$$

$$F_{n,\frac{1}{k}}(\alpha n - 1) < \frac{1-\alpha}{1-\alpha k} f_{n,\frac{1}{k}}(\alpha n) \text{ for } \alpha < \frac{1}{k} \tag{6}$$

Identity 5 follows immediately from the definitions of our cumulative distribution functions, F and G. Identity 6 follows from [8, Theorem 6.4], where we substitute our αn for their k, our $1/k$ for their p, and our $1 - 1/k$ for their q.

Theorem 8.2. *Using a sparse R-Table, the expected number of search hops in SkipNet is*

$$O(k \log_k D)$$

to arrive at a node distance D away from the source. More precisely, there exist constants $z_0 = \sqrt{e}$ and $t_0 = 9$, such that for $t \geq t_0$, the search requires no more than $(tk \log_k D + t^2 k)$ hops with probability at least $1 - 3/z_0^t$.

Proof: By Lemma 8.1, it suffices to upper bound the number of hops in the Skip List search path; we focus on the Skip List search path for the remainder of the proof. Define g to be $t + \log_k D$. Let X be the random variable giving the maximum level traversed in the Skip List search path. We now show that $Pr[X \geq g]$ is small. Note that the probability that a given node matches s in g or more digits is $1/k^g$. By a simple union bound, the probability that any node between s and d matches s in g or more digits is at most D/k^g. Thus,

$$\begin{aligned} Pr[X \geq g] &\leq D/k^g \\ &= 1/k^{g - \log_k D} \\ &= 1/k^t \end{aligned}$$

Let Y be the random variable giving the number of hops traversed in a Skip List search path, and define m to be tkg, i.e., $m = (tk \log_k D + t^2 k)$. We will upper bound the probability that Y takes more than m hops via:

$$\begin{aligned} Pr[Y > m] = \; & Pr[Y > m \text{ and } X < g] \\ & + Pr[Y > m \text{ and } X \geq g] \\ \leq \; & Pr[Y > m \text{ and } X < g] \\ & + Pr[X \geq g] \end{aligned}$$

It remains to show that the probability the search takes more than m hops without traversing a level g pointer is small. The classical Skip List analysis [31] upper bounds this probability using the negative binomial distribution, showing that $Pr[Y > m \text{ and } X < g] \leq 1 - G_{g,1/k}(m)$. Using Identity 5, we have $1 - G_{g,1/k}(m) = F_{m,1/k}(g - 1)$. Setting $\alpha = 1/tk$ and applying Identity 6 gives the following upper bound:

$$F_{m,\frac{1}{k}}(g-1) = F_{m,\frac{1}{k}}(\alpha m - 1) < \frac{1-\alpha}{1-\alpha k} f_{m,\frac{1}{k}}(\alpha m)$$

Note that $\frac{1-\alpha}{1-\alpha k}$ is at most 2, since t and k are both at least 2. This yields that $F_{m,1/k}(g-1)$ is less than:

$$\begin{aligned} & 2\binom{m}{g}(1/k)^g(1-1/k)^{m-g} \\ &= 2\binom{tkg}{g}(1/k)^g(1-1/k)^{tkg}(1-1/k)^{-g} \\ &< 2\frac{(tkg)^g}{g!}(1/k)^g e^{-tg}(1-1/k)^{-g} \\ &< 2e^{g \log tkg}\left(\frac{1}{\sqrt{2\pi g}}\left(\frac{g}{e}\right)^{-g}\right)e^{-g \log k}e^{-tg}e^g \\ &< 2e^{g \log tkg}e^{-g \log g}e^g e^{-g \log k}e^{-tg}e^g \\ &\leq 2e^{g(\log t + \log k + \log g) - g \log g + g - g \log k + g - tg} \\ &= 2e^{g \log t + g + g - tg} \\ &= 2e^{(-t + \log t + 2)g} \end{aligned}$$

For $t \geq 9$, we have $-t + \log t + 2 < -t/2 < 0$ and so $e^{(-t+\log t+2)g} < e^{-t/2}$. Thus,

$$F_{m,1/k}(g) < 2e^{-t/2}$$

Combining our results and letting $z_0 = \sqrt{e}$ yields

$$\begin{aligned} Pr[Y > m] &\leq Pr[Y > m \text{ and } X < g] + Pr[X \geq g] \\ &\leq 2/e^{t/2} + 1/k^t \\ &< 3/z_0^t \end{aligned}$$

Setting $t_0 = 9$, for $t \geq t_0$, we have that $Pr[Y > m] < 3/z_0^t$. That is, $Pr[Y \leq m] \geq 1 - 3/z_0^t$. The expectation bound straightforwardly follows. ■

We now consider the case of searching by name ID in a SkipNet using a *dense* R-Table. Recall that a dense R-Table points to the $k - 1$ closest neighbours in each direction at each level. Note that it would be possible to use the same approach to create a 'dense Skip List', but such a structure would not be useful because in a Skip List, comparisons are typically more expensive than hops. Whenever we refer to a Skip List, we are always referring to a sparse Skip List. Define P to be the SkipNet search path with a dense R-Table and, as before, let Q be the path that the Skip List search algorithm would use in the induced Skip List.

To characterize the path P, it will be helpful to let $G(x, y, h)$ denote to be the number of hops between nodes x and y in the ring that contains them both at level h. If $h > F(x, y)$ (meaning nodes x and y are not in the same ring at level h), we define $G(x, y, h) = \infty$. Note that node x has a pointer to node y at level h if and only if $G(x, y, h) < k$. At each intermediate node on the SkipNet search path we hop using the pointer that takes us as close to the destination as possible without going beyond it. The formal characterization is: $x \in [s, d]$ immediately follows w in P if and only if $G(w, x, F(w, x)) < k$ and $\nexists y, h$ such that $x < y \leq d$ and $G(w, y, h) < k$.

Lemma 8.3. *Let P be the SkipNet search path with a dense R-Table and let Q be the path that the Skip List* *search algorithm would use in the induced Skip List. Then P is a subsequence of Q.*

Proof: The proof begins by defining the same quantities as in the proof of Lemma 8.1. Suppose for the purpose of showing a contradiction that some node x in P does not appear in Q. Let x be the first such node; clearly $x \neq s$ because s must appear in both P and Q. Let w denote x's predecessor in P; since $x \neq s$, x is not the first node in P and so w is indeed well-defined. Node w must belong to Q because x was the first node in P that is not in Q.

We consider the three cases that $F(s,x) > F(s,w), F(s,x) = F(s,w), F(s,x) < F(s,w)$ separately. The first two were shown to lead to a contradiction in the proof of Lemma 8.1 without reference to the SkipNet search path; thus it remains to consider only the case $F(s,x) < F(s,w)$.

Let $l = G(w,x,F(w,x))$ be the number of hops between w and x in the highest ring that contains them both. Since $x \in P$, we must have $l < k$ (from the characterization of the dense SkipNet search path). Since $x \notin Q$, there must exist $y \in [x,d]$ such that $F(s,y) > F(s,x)$ (from the characterization of the Skip List search path). Since $w \in Q$ and $y \in [w,d]$, it cannot be the case that $F(s,y) > F(s,w)$, otherwise that would contradict the fact that $w \in Q$ (using the Skip List search path characterization again). Therefore $F(s,y) \leq F(s,w)$, and Identity 3 yields that $F(w,y) \geq F(s,y)$. Applying Identity 2 to $F(s,x) < F(s,w)$ (the case assumption) implies $F(w,x) = F(s,x)$. Putting the inequalities together yields $F(w,y) \geq F(s,y) > F(s,x) = F(w,x)$. We apply the conclusion, $F(w,y) > F(w,x)$, in the rest of the proof to derive a contradiction.

Consider the ring containing w at level $F(w,y)$. Node y must be in this ring but node x cannot be because $F(w,y) > F(w,x)$. Starting at w, consider traversing this ring until we encounter z, the first node on this ring with $x < z$ (to the right of x). Such a node z must exist because y is in this ring and $x < y$. Note that $x < z \leq y \leq d$.

Since this ring at level $F(w,y)$ is a strict subset of the ring at at level $F(w,x)$ (in particular, x is not in it), it takes at most $l < k$ hops to traverse from w to z. We now have $x < z \leq d$ and $G(w,z,F(w,y)) < k$, which contradicts the fact that $x \in Q$. ■

Lemma 8.4. *Let P be the SkipNet search path from s to d using a dense R-Table. Let Q be the search path from s to d in the induced Skip List. Let m be the number of hops along path Q and let g be the maximum level of a pointer traversed on path Q. Then the number of hops taken on path P is at most $\frac{m}{k-1} + g + 1$.*

Proof: Let $Q = (s, q_1, \ldots, q_m)$ be the sequence of nodes on path Q, where $q_m = d$. By choice of g, $F(s,q_i) \leq g$ for all $i \geq 1$. Thus, the q_i nodes are partitioned into levels according to the value of $F(s,q_i)$. Recall that $F(s,q_i)$ is monotonically non-increasing with i since Q is a Skip List search path. Thus the nodes in each partition are contiguous on path Q.

Suppose P contains q_i. Using the dense R-Table, it is possible to advance in one hop to any node in the Skip List path that is at most $k-1$ hops away at level $F(s,q_i)$. Thus, if there are l_i nodes at level i in P, then Q contains at most $\lceil l_i/(k-1) \rceil$ of those nodes. Summing over all levels, Q contains at most $\frac{m}{k-1} + g + 1$ nodes. ■

Theorem 8.5. *Using a dense R-Table, the expected number of search hops is*

$$O(\log_k D)$$

to arrive at a node distance D away from the source. More precisely, for constants $z_0 = \sqrt{e}$ and $t_0 = 9$, and for $t \geq t_0$, the search completes in at most $(2t+1)\log_k D + 2t^2 + t + 1$ hops with probability at least $1 - 3/z_0^t$.

Proof: As in the proof of Theorem 8.2, with probability at least $1 - 3/z_0^t$ the number of levels in the Skip List search path is at most $g = t + \log_k D$, and the number of hops is at most $m = tkg = (tk \log_k D + t^2 k)$. Applying Lemma 8.4, the number of hops in the dense SkipNet search path is

$$\begin{aligned}
\frac{m}{k-1} + g + 1 &= \frac{tkg}{k-1} + g + 1 \\
&\leq 2tg + g + 1 = (2t+1)g + 1 \\
&= (2t+1)(t + \log_k D) + 1 \\
&= (2t+1)\log_k D + 2t^2 + t + 1
\end{aligned}$$

■

8.2 Correspondence between SkipNet and Tries

The pointers of a SkipNet effectively make every node the head of a Skip List ordered by the nodes' name IDs. Simultaneously, every node is also the root of a trie [13] on the nodes' numeric IDs. Thus the SkipNet simultaneously implements two distinct data structures in a single structure. One implication is that we can reuse the trie analysis to determine the expected number of non-null pointers in the sparse R-Table of a SkipNet node. This extends previous work relating Skip Lists and tries by Papadakis in [29, pp. 38]: The expected height of a Skip List with N nodes and parameter p corresponds exactly to the expected height of a $\frac{1}{p}$-ary trie with $N+1$ keys drawn from the uniform $[0,1]$ distribution.

Recall that ring membership in a SkipNet is determined as follows: For $i \geq 0$, two nodes belong to the same ring at level i if the first i digits of their numeric ID match exactly. All nodes belong to the one ring at level 0, which is called the *root ring*. Note that if two nodes belong to ring R at level $i > 0$ then they must also belong to the same ring at level $i - 1$, which we refer to as the *parent ring* of ring R. Moreover, every ring R at level $i \geq 0$ is partitioned into at most k disjoint rings at level $i + 1$, which we refer to as the *child rings* of ring R. Thus, the rings naturally form a *Ring Tree* which is rooted at the root ring.

Given a Ring Tree, one can construct a trie as follows. First, remove all rings whose parent ring contains a single node — this will collapse any subtree of the trie that contains only a single node. Every remaining ring that contains a single node is called a *leaf ring*; label the leaf ring with the numeric ID of its single node. The resulting structure on the rings *is* a trie containing all the numeric IDs of the nodes in the SkipNet.

Let Y_N be the random variable denoting the number of non-null right (equivalently, left) pointers at a particular node in a SkipNet containing N nodes. Papadakis defines D_N to be the random variable giving the depth of a node in a k-ary trie with keys drawn from the uniform $[0, 1]$ distribution. Note that Y_N is identical to the random variable giving the depth of a node's numeric ID in the trie constructed above, and thus we have $Y_N = D_N$.

We may use this correspondence and Papadakis' analysis to show that $E[Y_N] = 1 + V_{\frac{1}{k}}(N)$, where $V_{\frac{1}{k}}(N)$ is (as defined in [23]):

$$V_{\frac{1}{k}}(N) = \frac{1}{N} \sum_{g=2}^{N} \binom{N}{g} (-1)^g \frac{g \cdot (1/k)^{g-1}}{1 - (1/k)^{g-1}}$$

Knuth proves in [23, Ex. 6.3.19] that $V_{\frac{1}{k}}(N) = \log_k N + O(1)$, and thus the expected number of right (equivalently, left) non-null pointers is given by $E[Y_N] = \log_k N + O(1)$.

8.3 Searching by Numeric ID

SkipNet supports searches by numeric ID as well as searches by name ID. Searches by numeric ID in a dense SkipNet take $O(\log_k N)$ hops in expectation, and $O(k \log_k N)$ in a sparse SkipNet. We formally prove these results in Theorem 8.6. Intuitively, search by numeric ID corrects digits one at a time and needs to correct at most $O(\log_k N)$ digits. In the sparse SkipNet correcting a single digit requires about $O(k)$ hops, while in the dense case only $O(1)$ hops are required.

Theorem 8.6. *The expected number of hops in a search by numeric ID using a sparse R-Table is $O(k \log_k N)$. In a dense R-Table, the expected number of hops is $O(\log_k N)$. Additionally, these bounds hold with high probability (i.e., the number of hops is close to the expectation).*

Proof: We use the same upper bound as in the proof of Theorem 8.2,

$$\begin{aligned} &Pr[\text{search takes more than } m \text{ hops}] \\ &\quad \leq Pr[\text{more than } m \text{ hops and at most } g \text{ levels}] \\ &\qquad + Pr[\text{more than } g \text{ levels}] \end{aligned}$$

and bound the two terms separately. In Theorem 8.2 we showed that the maximum number of digits needed to uniquely identify a node is $g = O(\log_k N)$ with high probability, and thus no search by numeric ID will need to climb more than this many levels. This upper bounds the right-hand term. The number of hops necessary on any given level in the sparse R-Table before the next matching digit is found is upper bounded by a geometric random variable with parameter $1/k$. The sum of g of these random variables has expectation gk, and this random variable is close to its expectation with high probability (by standard arguments). Thus the expected number of hops in a search by numeric ID using a sparse R-Table is $O(k \log_k N)$, and additionally the bound holds with high probability.

For a search by numeric ID using a dense R-Table, we upper bound the number of hops necessary on any given level differently. Informally, instead of performing one experiment that succeeds with probability $1/k$ repeatedly, we perform $k - 1$ such experiments simultaneously. Formally, the probability of finding a matching digit in one hop is now $1 - (1 - 1/k)^{k-1} \geq 1/2$. Therefore the analysis in the case of a sparse R-Table need only be modified by replacing the parameter $1/k$ with $1/2$. Thus the expected number of hops in a search by numeric ID using a dense R-Table is $O(\log_k N)$, and additionally the bound holds with high probability. ■

8.4 Node Joins and Departure

We now analyze node join and departure operations using the analysis of both search by name ID and by numeric ID from the previous sections. As described in Section 3.5, a node join can be implemented using a search by numeric ID followed by a search by name ID, and will require $O(k \log_k N)$ hops in either a sparse or a dense SkipNet. Implementing node departure is even easier: As described in Section 3.5, a departing node need only notify its left and right neighbors at every level that it is leaving, and that the left and right neighbors of the departing node should point to each other. This yields a bound of $O(\log_k N)$ hops for the sparse SkipNet and $O(k \log_k N)$ for the dense SkipNet, where hops measure the total number of hops traversed by messages since these messages may be sent in parallel.

Theorem 8.7. *The number of hops required by a node join operation is $O(k \log_k N)$ in expectation and with high probability in either a sparse or a dense SkipNet.*

Proof: The join operation can be decomposed into a search by numeric ID, followed by a Skip List search by name ID. Because of this, the bound on the number of hops follows immediately from Theorem 8.2 and Theorem 8.6. It only remains to establish that the join operation finds all required neighbors of the joining node.

For a sparse SkipNet, the joining node needs a pointer at each level h to the node whose numeric ID matches in h digits that is closest to the right or closest to the left in the order on the name IDs. For a dense SkipNet, the joining node must find the same nodes as in the sparse SkipNet case, and then notify $k-2$ additional neighbors at each level.

The join operation begins with a search for a node with the most numeric ID digits in common with the joining node. The search by name ID operation for the joining node starts at this node, and it is implemented as a Skip List search by name ID; the pointers traversed are monotonically decreasing in height, in contrast to the normal SkipNet search by name ID. Whenever the Skip List search path drops a level, it is because the current node at level h points to a node beyond the joining node. Therefore this last node at level h on the Skip List search path is the closest node that matches the joining node in h digits. This gives the level h neighbor on one side, and the joining node's level h neighbor on the other side is that node's former neighbor. The message traversing the Skip List search path accumulates this information about all the required neighbors on its way to the joining node. This establishes the correctness of the join operation. ■

8.5 Node Stress

We now analyze the distribution of load when performing searches by name ID using R-Tables. To analyze the routing load, we must assume some distribution of routing traffic. We assume a uniform distribution on both the source and the destination of all routing traffic. This assumption may or may not seem plausible, but its plausibility is increased if SkipNet uses an obvious optimization. If the destination of a SkipNet routing query is cached at the search originator, then subsequent searches to the same destination could be routed directly over IP. Servicing repeated queries directly from the cache would increase the randomness of the queries that SkipNet must handle.

Under some routing algorithms (which happen not to preserve path locality), the distribution of routing load is obviously uniform. For example, if routing traffic were always routed to the right, the load would be uniform. If the source and destination name ID do not share a common prefix, then path locality is not an issue and the SkipNet routing algorithm randomly chooses a direction in which to route — such traffic is uniformly distributed.

If the SkipNet routing algorithm can preserve path locality, it does so by always routing in the direction of the destination (i.e., if the destination is to the right of the source, routing proceeds to the right). We show that in this case load is approximately balanced: very few nodes' loads are much smaller than the average load. We also shows that no node's load exceeds the average load by more than a constant factor with high probability; this result is relevant whether the routing algorithm preserves path locality or not. In the interest of simplicity, our proof assumes that $k = 2$; a similar result holds for arbitrary k. Also, we have previously given an upper bound of $O(\log d)$ on the number of hops between two nodes at distance d. In order to estimate the average load, we assume a tight bound of $\Theta(\log d)$ without proof.

Theorem 8.8. *Consider an interval on which we preserve path locality containing N nodes. Then the u^{th} node of the interval bears a $\Theta(\frac{\log \min\{u, N-u\}}{\log N})$ fraction of the average load in expectation.*

Proof: We first establish the expected load on node u due to routing traffic between a particular source l and destination r. The search path can only encounter u if, for some h, the numeric IDs of l and u have a common prefix of length h but no node between u and r has a longer common prefix with l. We observe that every node's random choice of numeric ID digits is independent, and apply a union bound over h to obtain the following upper bound on the probability that the search encounters u. Denote the distance from u to r by d.

$$\begin{aligned} &Pr[\text{search encounters } u] \\ &\quad \leq \sum_{h \geq 0} Pr[u \text{ and } l \text{ share } h \text{ digits}] \\ &\qquad\quad \cdot Pr[\text{no node between } u \text{ and } r \text{ shares more}] \\ &\quad = \sum_{h \geq 0} \frac{1}{2^h} \cdot \left(1 - \frac{1}{2^{h+1}}\right)^d \end{aligned}$$

Denote the term in the above summation by $H(h)$. Because $H(h)$ falls by at most a factor of 2 when h increases by 1, we can upper bound the summation using:

$$\sum_{h \geq 0} H(h) \leq 2 \cdot \int_{h \geq 0} H(h) dh$$

Making the change of variables $\alpha = 1 - \frac{1}{2^{h+1}}$, and hence

$d\alpha = \frac{\ln 2}{2^{h+1}} \cdot dh$, we obtain:

$$\int_{h\geq 0} H(h)dh = \int_{\alpha=1/2}^{1} \frac{2}{\ln 2} \cdot \alpha^d \cdot d\alpha$$
$$= \frac{2}{\ln 2} \cdot \frac{1^{d+1} - (\frac{1}{2})^{d+1}}{d+1} = O(1/d)$$

This completes the analysis of a single source/destination pair. A similar single pair analysis was also noted in [1]. We complete our theorem by considering all source/destination pairs.

Our bound on the average load of a node is given by the total number of source/destination pairs multiplied by the bound on search hops divided by the total number of nodes. Summing over all the routing traffic that passes through u and dividing by the average load yields the proportion of the average load that u carries. To within a constant factor, this is:

$$\frac{\sum_{l\in[1,u-1]} \sum_{r\in[u+1,r]} \left(\frac{1}{|u-l|} + \frac{1}{|u-r|}\right)}{(\binom{N}{2}\log N)/(N)}$$
$$= \frac{u\log(N-u) + (N-u)\log u}{((N-1)\log N)/2}$$
$$= \Theta\Big(\frac{\log \min\{u, N-u\}}{\log N}\Big)$$

■

Corollary 8.9. *The number of nodes with expected load less than $\Theta(\alpha \cdot$ average load$)$ is N^α.*

Proof: Apply Theorem 8.8 and note that $\frac{\log u}{\log N} < \alpha$ implies that $u < N^\alpha$. ■

This completes the analysis showing that few nodes expect to do much less work than the average node in the presence of path locality. Our next theorem shows that it is very unlikely any node will carry more than a constant factor times the average load; this analysis is relevant whether the routing policy maintains path locality or not.

Theorem 8.10. *With high probability, no node bears more than a constant factor times the average load.*

Proof: Consider any node u. There are at most N nodes to the left of u and at most N nodes to the right. As in the previous theorem, let l and r denote nodes to the left and right of u respectively. Then the Skip List path from l to r (of which the SkipNet path is a subsequence) encounters u only if there is some number h such that l and u share exactly h bits, but no node between u and r shares exactly h bits with u. Considering only routing traffic passing from left to right affects our bound by at most a factor of two.

Let L_h be a random variable denoting the number of l that share exactly h bits with u. Let R_h denote the number of r such that no node between u and r shares exactly h bits with u. (Note that if r shares exactly h bits with u, it must share *more than* h bits with l, and thus routing traffic from l to r does not pass through u.) The analysis in the previous paragraph implies that the load on u is exactly $\sum_h L_h R_h$. We desire to show that this quantity is $O(N \log N)$ with high probability.

The random variable L_h has the binomial distribution with parameter $1/2^{h+1}$. From this observation, standard arguments (that we have made explicit in earlier proofs in this section) show that L_h has expectation $N/2^{h+1}$, and for $h \in [0, \log N - \log\log N]$, $L_h = O(N/2^{h+1})$ with high probability. The number of l that share more than $\log N - \log\log N$ bits with u is $\log N$ in expectation, and is $O(\log N)$ with high probability; these l (whose number of common bits with u we do not bound) can contribute at most $O(N \log N)$ to the final total.

To analyze the random variables R_h, we introduce new random variables R'_h that stochastically dominate R_h. In particular, let R'_h be the distance from u to the first node *after* node R'_{h-1} that matches u in exactly h bits. Also, let $R'_0 = R_0$. We define additional random variables Y_h using the recurrence $R'_h = \sum_{i=0}^{h} Y_i$. The Y_h are completely independent of each other; Y_h only depends on the random bit choices of nodes after the nodes that determine Y_{h-1}.

The random variable Y_h is distributed as a geometric random variable with parameter $1/2^{h+1}$ (and upper bounded by N). We rewrite the quantity we desire to bound as

$$\sum_h L_h R_h =$$
$$O(N\log N) + \sum_{h=0}^{\log N - \log\log N} O\Big(\frac{N}{2^{h+1}}\Big) \cdot \sum_{i=0}^{h} Y_i$$

Using that the $N/2^{h+1}$ form a geometric series, we apply the upper bound

$$\sum_{h=0}^{\log N - \log\log N} \frac{N}{2^{h+1}} \cdot \sum_{i=0}^{h} Y_i \leq \sum_{h=0}^{\log N - \log\log N} \frac{2N}{2^{h+1}} \cdot Y_h$$

We have that $\sum_h L_h R_h$ equals $O(N \log N)$ plus the sum of (slightly fewer than) $\log N$ independent random variables, where the h^{th} random variable is distributed like a geometric random variable with parameter $1/2^h$ multiplied by $O(N/2^h)$, and thus has expectation $O(N)$. This yields the $O(N \log N)$ bound with high probability. ■

8.6 Virtual Node Analysis

We outlined in Section 5.8 a scheme by which a single physical node could host multiple virtual nodes. Using this scheme, the bounds on search hops are unaffected, and the number of pointers per physical node is only $O(k \log_k N + kv)$ in the dense case, where v is the number of virtual nodes. In the sparse case, the number of pointers is just $O(\log_k N + v)$.

Intuitively, we obtain this by relaxing the requirement that nodes after the first have height $O(\log_k N)$. We instead allow node heights to be randomly distributed as they are in a Skip List. Because Skip List nodes maintain a constant number of pointers in expectation, we add only $O(k)$ pointers per virtual node in the dense case, and $O(1)$ in the sparse case. Search are still efficient, just as they are in a Skip List.

Theorem 8.11. *Consider a single physical node supporting v virtual nodes using the scheme of Section 5.8. In the dense case, searches require $O(\log_k D)$ hops, and the number of pointers is $O(k \log_k N+kv)$. In the sparse case, searches require $O(k \log_k D)$ hops, and the number of pointers is $O(\log_k N + v)$. All these bounds hold in expectation and with high probability.*

Proof: The bound on the number of pointers is by construction. Consider the sparse case. The leading term in the bound, $O(\log_k N)$, is due to the one virtual node that is given all of its SkipNet pointers. The additional virtual nodes have heights given by geometric random variables with parameter 1/2, which is $O(1)$ in expectation. The claimed bound on the number of pointers immediately follows, and the dense case follows by an identical argument with an additional factor of k.

We now analyze the number of search hops, focusing first on the sparse case. Because we might begin the search at a virtual node that does not have full height, we will break the analysis into two phases. During the first phase, the search path uses pointers of increasing level. At some point, we encounter a node whose highest pointer goes beyond the destination. From this point on (the second phase), we consider the Skip List search path to the destination that begins at this node. As in Theorem 8.2, the rest of the actual search path will be a subsequence of this Skip List path.

As in Theorem 8.2, the maximum level of any pointer in this interval of D nodes is $O(\log_k D)$ with high probability. Suppose that some particular node t is the first node encountered whose highest pointer points beyond the destination. In this case, the first phase is exactly a search by numeric ID for t's numeric ID, and therefore the high probability bound of Theorem 8.6 on the number of hops applies. The second phase is a search from t for d, and the high probability bound of Theorem 8.2 on the number of hops applies. There is a subtlety to this second argument — although some or all of the intermediate nodes may be virtual, the actual search path is necessarily a subset of the search path in the Skip List induced by t (by the arguments of Lemma 8.1 and Lemma 8.3). We previously supposed that t was fixed; because there are at most D possibilities for t, considering all such possibilities increases the probability of requiring more than $O(k \log_k D)$ hops by at most a factor of D. Because the bound held with high probability initially, the probability of exceeding this bound remains negligible.

This yields the result in the sparse case. An identical argument holds in the dense case. ■

8.7 Ring Merge

We now analyze the performance of the proactive algorithm for merging disjoint SkipNet segments, as described in Section 6. Consider the merge of a single SkipNet segment containing M nodes with a larger SkipNet segment containing N nodes. In the interest of simplicity, our discussion assumes that $k = 2$; a similar analysis applies for arbitrary k. Recall that the expected maximum level of a ring in the merged SkipNet is $O(\log N)$ with high probability (Section 8.2). Intuitively, the expected time to repair a ring at a given level after having reached that level is $O(1)$ and ring repair occurs in parallel across all rings at a given level. This suggests that the expected time required to perform the merge operation is $O(\log N)$, and we will show this formally in Theorem 8.12 under the assumption that the underlying network accommodates unbounded parallelization of the repair traffic. In practice, the bandwidth of the network may impose a limit: doing many repairs in parallel may saturate the network and hence take more time.

The expected amount of work required by the merge is $O(M \log(N/M)) = O(N)$. We first give an intuitive justification for this. The merge operation repairs at most four pointers per SkipNet ring. Since the total number of rings in the merged SkipNet is $O(N)$ and the expected work required to repair a ring is $O(1)$, the expected total work performed by the merge operation is $O(N)$. Additionally, if M is much less than N, the bound $O(M \log(N/M))$ proved in Theorem 8.13 is much less than $O(N)$.

Now consider an organization consisting of S disjoint SkipNet segments, each of size at most M, merging into a global SkipNet of size N. In this case, the merge algorithm sequentially merges each segment of the organization one at a time into the global SkipNet. The total time required in this case is $O(S \log N)$ and the total work performed is $O(SM \log(N/M))$; these are straightforward corollaries of Theorem 8.12 and Theorem 8.13.

Theorem 8.12. *The time to merge a SkipNet segment of size M with a larger SkipNet segment of size N is $O(\log N)$ with high probability, assuming sufficient bandwidth in the underlying network.*

Proof: After repairing a ring, the merge operation branches to repair both child rings in parallel, until there are no more child rings. Using the analogy with tries from Section 8.2, consider any path along the branches from the root ring to a ring with no children. We show that this path uses $O(\log N)$ hops with high probability. Union bounding over all such paths will complete the theorem.

We can assume that the height of any pointer is at most $c_1 \log N$. The number of hops to traverse this path is then upper bounded by a sum of $c_1 \log N$ geometric random variables with parameter $1/2$. We now show that this sum is at most $c_2 \log N = O(\log N)$ with high probability. Applying the same reduction as in Section 8.1, using Identity 5 and Identity 6, we obtain the following upper bound on the probability of taking more than $c_2 \log N$ hops:

$$
\begin{aligned}
& F_{c_2 \log N, 1/2}(c_1 \log N) \\
& \leq \frac{1 - c_1/c_2}{1 - 2c_1/c_2} f_{c_2 \log N, 1/2}(c_1 \log N) \\
& = \left(\frac{1 - c_1/c_2}{1 - 2c_1/c_2} \right) \binom{c_2 \log N}{c_1 \log N} (1/2)^{c_2 \log N} \\
& \leq \left(\frac{1 - c_1/c_2}{1 - 2c_1/c_2} \right) \frac{(c_2 \log N)^{c_1 \log N}}{(c_1 \log N)!} (1/2)^{c_2 \log N} \\
& \leq \left(\frac{1 - c_1/c_2}{1 - 2c_1/c_2} \right) \frac{(c_2 \log N)^{c_1 \log N}}{\left(\frac{c_1 \log N}{e} \right)^{c_1 \log N}} 2^{-c_2 \log N} \\
& < \left(\frac{1 - c_1/c_2}{1 - 2c_1/c_2} \right) \left(\frac{c_2 \cdot e}{c_1} \right)^{c_1 \log N} 2^{-c_2 \log N}
\end{aligned}
$$

Choosing $c_2 = \max\{7c_1, 7\}$, this is at most $2N^{-2}$. Applying a union bound over the N possible paths completes the proof. ■

Theorem 8.13. *The expected total work to merge a SkipNet segment of size M with a larger SkipNet segment of size N is $O(M \log(N/M))$.*

Proof: Suppose all the pointers at level i have been repaired and consider any two level $i+1$ rings that are children of a single level i ring. To repair the pointers in these two child rings, the nodes adjacent to the segment boundary at level i must each find the first node in the direction away from the segment boundary who differs in the i^{th} bit. The number of hops necessary to find either node is upper bounded by a geometric random variable with parameter 1/2. Only $O(1)$ additional hops are necessary to finish the repair operation.

By considering a particular order on the random bit choices, we show that the number of additional hops incurred in every ring repair operation are independent random variables. Let all the level i bits be chosen before the level $i+1$ bits. Then the number of hops incurred in fixing any two level $i+1$ rings that are children of the same level i ring depends only on the level $i+1$ random bits of those two rings. Also, only rings that require repair initiate a repair operation on their children. Therefore we can assume that the level i rings from which we will continue the merge operation are fixed before we choose the level $i+1$ bits. Hence the number of hops incurred in repairing these two child rings is independent of the number of hops incurred in the repair of any other ring.

We now consider the levels of the pointers that require repair. For low levels, we use the bound that the number of pointers needing repair at level i is at most 2^i because there are at most 2^i rings at this level. For higher levels, we prove a high probability bound on the total number of pointers that need to be repaired, showing that the total number is $M(\log N + O(1))$ with high probability in M.

A node of height i cannot contribute more than i pointers to the total number needing repair. We upper bound the probability that a particular node's height exceeds h by:

$$
Pr[\text{height} > h] \leq \frac{N+M}{2^h} \leq \frac{2N}{2^h} = \frac{1}{2^{h - \log N - 1}}
$$

Thus each node's height is upper bounded by a geometric random variable starting at $(\log N + 1)$ with parameter $1/2$, and these random variables are independent. By standard arguments, their sum is at most $M(\log N + 3)$ with high probability in M.

The contribution of the first $\log M$ levels is at most $2M$ pointers, while the remaining levels contribute at most $M(\log N + 3 - \log M)$ with high probability. In total, the number of pointers is $O(M \log(N/M))$. The total number of hops is bounded by the sum of this many geometric random variables. This sum has expectation $O(M \log(N/M))$ and is close to this expectation with high probability, again by standard arguments. ■

8.8 Incorporating the P-Table and the C-Table

We first argue that our bounds on search by numeric ID, node join, and node departure continue to hold with the addition of C-Tables to SkipNet. Search by numeric ID corrects at least one digit on each hop, and there are never more than $O(\log_k N)$ digits to correct (Section 8.2). Construction of a C-Table during node join amounts to a search by numeric ID, using C-Tables, from an arbitrary SkipNet node to the joining node. This yields the same bound on node join as on search by numeric ID. During node departure, no work is performed to maintain the C-Table.

We only give an informal argument that search by name ID, node join, and departure continue to be efficient with the addition of P-Tables. Intuitively, search by name ID using P-Tables encounters nodes that interleave the R-Table nodes and since the R-Table nodes are exponentially distributed in expectation, we expect the P-Table nodes to be approximately exponentially distributed as well. Thus search should still approximately divide the distance to the destination by k on each hop.

P-Table construction during node join is more involved. Suppose that the intervals defined by the R-Table are perfectly exponentially distributed. Finding a node in the furthest interval is essentially a single search by name ID, and thus takes $O(\log_k N)$ time. Suppose the interval we are currently in contains g nodes. Finding a node in the next closest interval (containing at least g/k nodes) has at least constant probability of requiring only one hop. If we don't arrive in the next closest interval after the first hop, we expect to be much closer, and we expect the second hop to succeed in arriving in the next closest interval with good probability. Iterating over all intervals, the total number of hops is $O(k \log_k N)$ to fill in every P-Table entry.

This completes the informal argument for construction of P-Tables during node join. As with C-Tables, no work is performed to maintain the P-Table during node departure.

9 Experimental Evaluation

To understand and evaluate SkipNet's design and performance, we used a simple packet-level, discrete event simulator that counts the number of packets sent over a physical link and assigns either a unit hop count or a specified delay for each link, depending upon the topology used. It does not model either queuing delay or packet losses because modelling these would prevent simulation of large networks.

Our simulator implements three overlay network designs: Pastry, Chord, and SkipNet. The Pastry implementation is described in [34]. Our Chord implementation is based on the one available on the MIT Chord web site [20], adapted to operate within our simulator. The corresponding algorithms are described in [38]. For our simulations, we run the Chord stabilization algorithm until no finger pointers need updating after all nodes have joined. We use two different implementations of SkipNet: a "basic" implementation that uses only the R-Table with duplicate pointer elimination, and a "full" implementation that includes the P-Table and C-Table as well. The full SkipNet implementation uses a sparse R-Table, and a dense P-Table with density parameter $k = 8$. For full SkipNet, we run two rounds of stabilization for P-Table entries before each experiment.

All our experiments were run both on a Mercator topology [40] and a GT-ITM topology [43]. The Mercator topology has 102,639 nodes and 142,303 links. Each node is assigned to one of 2,662 Autonomous Systems (ASs). There are 4,851 links between ASs in the topology. The Mercator topology assigns a unit hop count for each link. All figures shown in this section are for the Mercator topology. The experiments based on the GT-ITM topology produced similar results.

Our GT-ITM topology has 5050 core routers generated using the Georgia Tech random graph generator according to a transit-stub model. Application nodes were assigned to core routers with uniform probability. Each end system was directly attached by a LAN link to its assigned router (as was done in [6]). We used the routing policy weights generated by the Georgia Tech random graph generator [43] to perform IP unicast routing. The delay of each LAN link was set to 1ms and the average delay of core links was 40.5ms.

9.1 Methodology

We measured the performance characteristics of lookups using the following evaluation criteria:

Relative Delay Penalty (RDP): The ratio of the latency of the overlay network path between two nodes to the latency of the IP-level path between them.

Physical network distance: The absolute length of the overlay path between two nodes, in terms of the underlying network distance. For the Mercator topology we measure latency in terms of physical network hops since the Mercator topology does not provide link latencies. For the GT-ITM topology we measure latency in terms of milliseconds. In contrast, RDP measures the penalty of using an overlay network relative to IP. However, since part of SkipNet's goal is to enable the placement of data near its clients, we also care about the absolute latency that a DHT lookup request incurs.

Number of failed lookups: The number of unsuccessful lookup requests in the presence of failures.

We also model the presence of organizations within the overlay network; each participating node belongs to a single organization. The number of organizations is a parameter to the experiment, as is the total number of nodes in the overlay. For each experiment, the total number of client lookups is ten times the number of nodes in the overlay.

The format of the names of participating nodes is *org-name/node-name*. The format of data object names is *org-*

name/node-name/random-obj-name. Therefore we assume that the "owner" of a particular data object will name it with the owner node's name followed by a node-local object name. In SkipNet, this results in a data object being placed on the owner's node; in Chord and Pastry, the object is placed on a node corresponding to the SHA-1 hash of the object's name. For constrained load balancing experiments we use data object names that include the '!' delimiter following the name of the organization.

We model organization sizes two ways: a uniform model and a Zipf-like model.

- In the uniform model the size of each organization is uniformly distributed between 1 and N – the total number of application nodes in the overlay network.
- In the Zipf-like model, the size of an organization is determined according to a distribution governed by $x^{-1.25} + 0.5$ and normalized to the total number of overlay nodes in the system. All other Zipf-like distributions mentioned in this section are defined in a similar manner.

We model three kinds of node locality: uniform, clustered, and Zipf-clustered.

- In the uniform model, nodes are uniformly spread throughout the overlay.
- In the clustered model, the nodes of an organization are uniformly spread throughout a single randomly chosen autonomous system in the Mercator topology and throughout a randomly chosen stub network in GT-ITM. In Mercator we ensure that the selected AS has at least 1/10-th as many core router nodes as overlay nodes. For GT-ITM, if an organization has 1000 or less member nodes, then we spread it across a single stub network, otherwise we spread it across a "stub cluster" – a set of stub networks that all connect to the same transit link.
- For Zipf-clustered, we place organizations within ASes or stub networks, as before. However, the nodes of an organization are spread throughout its AS or stub network as follows: A "root" physical node is randomly placed within the AS or stub network and all overlay nodes are placed relative to this root, at distances modeled by a Zipf-like distribution. In this configuration most of the overlay nodes of an organization will be closely clustered together within their AS or stub network. This configuration is especially relevant to the Mercator topology, in which some ASes span large portions of the entire topology.

Data object names, and therefore data placement, are modelled similarly. In a uniform model, data names are generated by randomly selecting an organization and then a random node within that organization. In a clustered model, data names are generated by selecting an organization according to a Zipf-like distribution and then a random member node within that organization. For Zipf-clustered, data names are generated by randomly selecting an organization according to a Zipf-like distribution and then selecting a member node according to a Zipf-like distribution of its distance from the "root" node of the organization. Note that for Chord and Pastry, but not SkipNet, hashing spreads data objects uniformly among all overlay nodes in all of these three models.

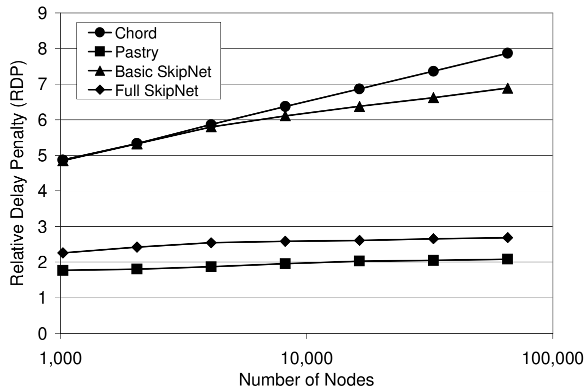

Figure 12. *RDP as a function of network size. Configuration: 1000 organizations with Zipf-like sizes, nodes and data names are Zipf-clustered.*

For SkipNet, the actual node names used in our simulations may impact performance, so we used realistic distributions for both host names and organization names. Our distribution of organization names was derived from a list of 5,608 unique organizations which had at least one peer participating in Gnutella in March 2001 [37]. The host name distribution was obtained from a list of 177,000 internal host names in use at Microsoft Corporation.

We model locality of data access by specifying what fraction of all data lookups will be forced to request data local to the requestor's organization. Finally, we model system behavior under Internet-like failures and study document availability within a disconnected organization. We simulate domain isolation by failing the links connecting the organization's AS to the rest of the network in Mercator and by failing the relevant transit links in GT-IM.

Each experiment is run ten times, with different random seeds, and the mean values are presented. SkipNet uses 128-bit numeric IDs and a leaf set of 16 nodes. Chord and Pastry use their default configurations [38, 34].

Our experiments measured the costs of sending overlay messages to overlay nodes using the different over-

Chord	Basic SkipNet	Full SkipNet	Pastry
16.3	41.7	102.2	63.2

Table 1. *Average number of unique routing entries per node in an overlay with* 2^{16} *nodes.*

lays under various distributions of nodes and content. Data gathered included:

Application Hops: The number of application-level hops required to route a message via the overlay to the destination

Relative Delay Penalty (RDP): The ratio between the average delay using overlay routing and the average delay using IP routing.

Experimental parameters varied included:

Overlay Type: *Chord*, *Pastry*, *Basic SkipNet*, or *Full SkipNet*.

Topology: *Mercator* (the default) or *GT-ITM*.

Message Type: Either *DHT Lookup* (the default), indicating that messages are DHT lookups, or *Send*, indicating that messages are being sent to randomly chosen overlay nodes.

Nodes (N): Number of overlay nodes. Most experiments vary N from 2^8 through 2^{16} increasing by powers of two. Some fix N at 2^{16}.

Lookups: Number of lookup requests routed per experiment. Usually $10 \times N$.

Trials: The number of times each experiment is run, each with different random seed values. Usually 10. Results reported are the average of all runs.

Organizations: Number of distinct organization names content is located within. Typical values include 1, 10, 100, and 1000 organizations. Nodes within an organization are located within the same region of the simulated network topology. For Mercator topologies they are located within the same Autonomous System (AS). In a GT-ITM topology for small organizations they are all nodes attached to the same stub network and for large organizations they are all nodes connected to the same stub cluster – a set of stub networks that all connect to the same transit link.

Organization Sizes: One of *Uniform* – indicating randomly chosen organization sizes between 1 and N in size or *Zipf* – indicating organization sizes chosen using a $\frac{1}{x^{1.25}}$ Zipf distribution with the largest organization size being $\frac{1}{2}N$.

Node Locality: One of *Uniform* or *Zipf*. Controls how node locations cluster within each organization. *Uniform* spreads nodes randomly among the nodes within an organization's topology. *Zipf* sorts candidate nodes by distance from a chosen root node within an organization's topology and clusters nodes using a Zipf distribution near that node.

Document Locality: One of *Uniform*, *By Org*, or *By Node*. *Uniform* spreads document names uniformly across all nodes. *By Org* applies a Zipf-like distribution causing larger organizations to have a larger share of documents, with documents uniformly distributed across nodes within each organization. *By Node* is used in conjunction with a Zipf-like distribution of nodes within an organization to distribute documents within the organization with the same distribution as the nodes themselves.

% Local: Fraction of lookups that are constrained to be local to documents within the client's organization. Non-local lookups are distributed among all documents in the experiment.

Overlay-specific parameter defaults were:

Chord: NodeID Bits = 40.

Pastry: NodeID Bits = 128, Bits per Digit (b) = 4, Leaf Set size = 16.

SkipNet: *Basic configuration*: Random ID Bits = 128, Leaf Set size = 16, ring branching factor (k) = 2. *Full configuration*: Same as basic, except k = 8 and adds use of P-Table for proximity awareness and C-Table for efficient numeric routing.

9.2 Basic Routing Costs

To understand SkipNet's routing performance we simulated overlay networks varying the number of nodes from 1,024 to 65,536. We ran experiments with 10, 100, and 1000 organizations and with all the permutations obtainable for organization size distribution, node placement, and data placement. The intent was to see how RDP behaves under various configurations. We were especially curious to see whether the non-uniform distribution of data object names would adversely affect the performance of SkipNet lookups, as compared to Chord and Pastry.

Figure 12 presents the RDPs measured for both implementations of SkipNet, as well as Chord and Pastry. Table 1 shows the average number of unique routing table entries per node in an overlay with 2^{16} nodes. All other configurations, including the completely uniform ones, exhibited similar results to those shown here.

Our conclusion is that basic SkipNet performs similarly to Chord and full SkipNet performs similarly to Pastry. This is not surprising since both basic SkipNet and Chord do not support network proximity-aware routing whereas full SkipNet and Pastry do. Since all our other configurations produced similar results, we conclude that SkipNet's performance is not adversely affected by non-uniform distributions of names.

9.3 Exploiting Locality of Placement

RDP only measures performance relative to IP-based routing. However, one of SkipNet's key benefits is that

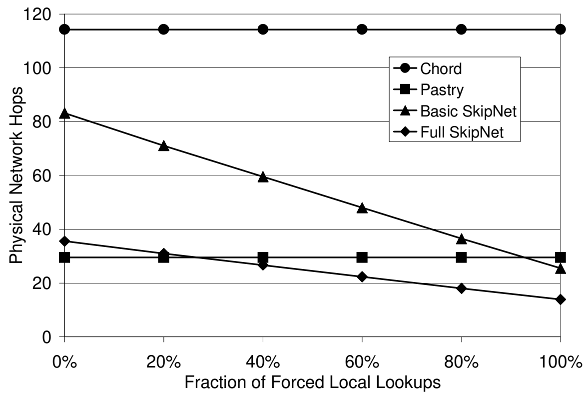

Figure 13. *Absolute latency (in network hops) for lookups as a function of data access locality (percentage of lookups forced to be within a single organization). Configuration:* 2^{16} *nodes, 100 organizations with Zipf-like sizes, nodes and data names are Zipf-clustered.*

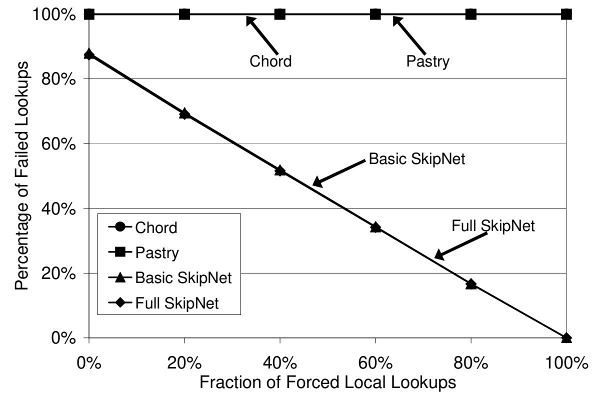

Figure 14. *Number of failed lookup requests as a function of data access locality (percentage of lookup requests forced to be within a single organization) for a disconnected organization. Configuration:* 2^{16} *nodes, 100 organizations with Zipf-like sizes, nodes and data names are Zipf-clustered.*

it enables localized placement of data. Figure 13 shows the average number of physical network hops for lookup requests. The x-axis indicates what fraction of lookups were forced to be to local data (i.e., the data object names that were looked up were from the same organization as the requesting client). The y-axis shows the number of physical network hops for lookup requests.

As expected, both Chord and Pastry are oblivious to the locality of data references since they diffuse data throughout their overlay network. On the other hand, both versions of SkipNet show significant performance improvements as the locality of data references increases. It should be noted that Figure 13 actually understates the benefits gained by SkipNet because, in our Mercator topology, inter-domain links have the same cost as intra-domain links. In an equivalent experiment run on the GT-ITM topology, SkipNet end-to-end lookup latencies were over a factor of seven less than Pastry's for 100% local lookups.

9.4 Fault Tolerance

Content locality also improves fault tolerance. Figure 14 shows the number of lookups that failed when an organization was disconnected from the rest of the network.

This (common) Internet-like failure had catastrophic consequences for Chord and Pastry. The size of the isolated organization in this experiment was roughly 15% of the total nodes in the system. Consequently, Chord and Pastry will both place roughly 85% of the organization's data on nodes outside the organization. Furthermore, they must also attempt to route lookup requests with 85% of the overlay network's nodes effectively failed (from the disconnected organization's point-of-view). At this level of failures, routing is effectively impossible. The net result is a failed lookups ratio of very close to 100%.

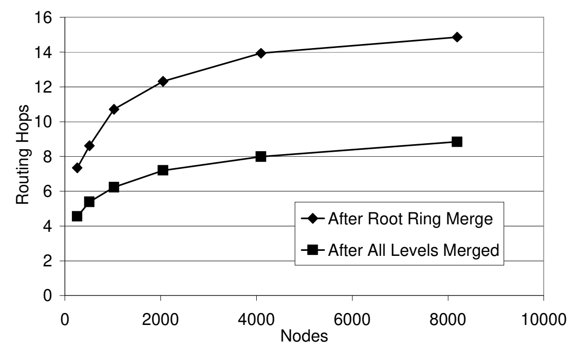

Figure 15. *Number of routing hops taken to route inter-organizational messages, as a function of network size, after an organization's internal SkipNet has been reconnected to the global SkipNet root ring and after the merge has been fully completed.*

In contrast, both versions of SkipNet do better the more locality of reference there is. When no lookups are forced to be local, SkipNet fails to access the 85% of data that is non-local to the organization. As the percentage of local lookups is increased to 100%, the percentage of failed lookups goes to 0.

To experimentally confirm the behavior of SkipNet's disconnection and merge algorithms described in Section 6, we extended the simulator to support disconnection of AS subnetworks. Figure 15 shows the routing performance we observed between a previously disconnected organization and the rest of the system once the organization's SkipNet root ring has been connected to the global SkipNet root ring. We also show the routing performance observed when all higher level pointers have been repaired.

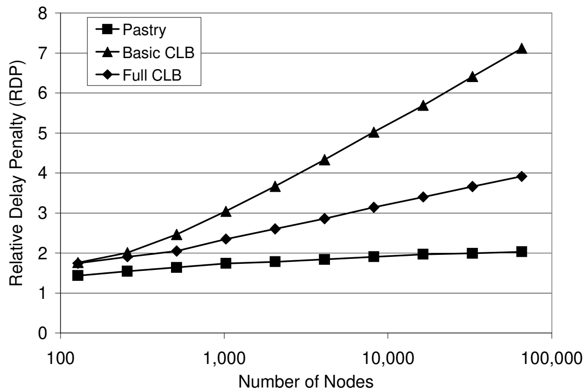

Figure 16. *RDP of lookups for data that is constrained load balanced (CLB) as a function of network size. Configuration: 100 organizations with Zipf-like sizes, nodes and data names are Zipf-clustered.*

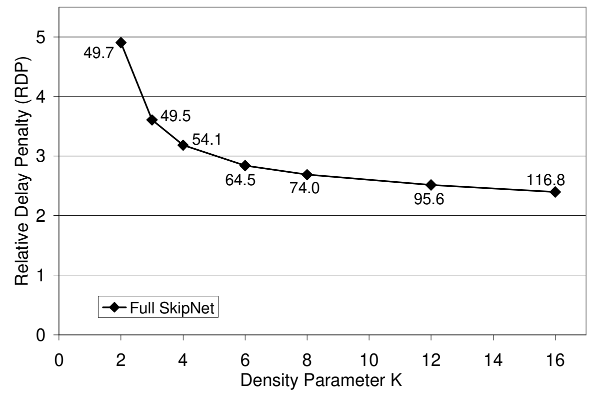

Figure 17. *RDP for Full SkipNet as a function of the density configuration parameter k. The labels next to each point represent the average number of unique pointers per node. Configuration: 2^{16} nodes, 1000 organizations with Zipf-like sizes, nodes and data names are Zipf-clustered.*

9.5 Constrained Load Balancing

Figure 16 explores the routing performance of two different CLB configurations, and compares their performance with Pastry. For each system, all lookup traffic is organization-local data. The organization sizes as well as node and data placement are clustered with a Zipf-like distribution. The Basic CLB configuration uses only the R-Table described in Section 3, whereas Full CLB makes use of the R-Table and the C-Table, as described in Section 5.4.

The Full CLB curve shows a significant performance improvement over Basic CLB, justifying the cost of maintaining the extra routing tables. However, even with the additional tables, the Full CLB performance trails Pastry's performance. We plan to investigate further techniques to reduce the latency of CLB. The key observation, however, is that in order to mimic the CLB functionality with a traditional peer-to-peer overlay network, multiple routing tables are required, one for each domain that you want to load-balance across.

9.6 Network Proximity

Figure 17 shows the performance of SkipNet routing using the P-Table. The x-axis varies the configuration parameter k which controls the density of P-Table pointers. The y-axis shows the routing performance in terms of RDP, and each data point is labelled with the average number of unique pointers per node. Note that the C-Table was not enabled so the pointers are from the R-table, P-Table and leaf set. Figure 17 shows that for small values of k, increasing k yields a large RDP improvement with a small increase in the number of pointers. As k grows, we see minimal improvement in RDP but significantly more pointers. This suggests that choosing $k = 8$ provides most of the RDP benefit with a reasonable number of pointers.

We also analyzed the sensitivity of P-Table performance to the choice of the initial seed node. We compared the performance when choosing a seed node at random with choosing the seed as the closest node in the system. Our results show virtually identical performance, which indicates that the P-Table join mechanism is effective at locating a nearby seed.

10 Conclusion

To become broadly acceptable application infrastructure, peer-to-peer systems need to support both content and path locality: the ability to control where data is stored and to guarantee that routing paths remain local within an administrative domain whenever possible. These properties provide a number of advantages, including improved availability, performance, manageability, and security. To our knowledge, SkipNet is the first peer-to-peer system design that achieves both content and routing path locality. SkipNet achieves this without sacrificing the performance goals of previous peer-to-peer systems: Nodes maintain a logarithmic amount of state and operations require a logarithmic number of message hops.

SkipNet provides content locality at any desired degree of granularity. Constrained load balancing encompasses placing data on a particular node, as well as traditional DHT functionality, and any intermediate level of granularity. This granularity is only limited by the hierarchy encoded in nodes' name IDs.

Clustering node names by organization allows SkipNet to perform gracefully in the face of a common type of Internet failure: When an organization loses connectivity to the rest of the network, SkipNet fragments into two segments that are still able to route efficiently internally. SkipNet also provides a mechanism to efficiently re-merge these segments with the global SkipNet when the network partition heals. With uncorrelated and independent node failures, SkipNet behaves comparably to other peer-to-peer systems.

Our evaluation has demonstrated that SkipNet's performance is similar to other peer-to-peer systems such as Chord and Pastry under uniform access patterns. Under access patterns where intra-organizational traffic predominates, SkipNet performs better. Our experiments show that SkipNet is significantly more resilient to organizational network partitions than other peer-to-peer systems.

In future work, we plan to deploy SkipNet across a testbed of 2000 machines emulating a WAN. This deployment should further our understanding of SkipNet's behavior in the face of dynamic node joins and departures, network congestion, and other real-world scenarios. We also plan to evaluate SkipNet as infrastructure for implementing a scalable event notification service [2].

Finally, we'd like to close by saying that the code described in this paper is available as part of the public SkipNet release [18]. Thus, you can see for yourselves the actual algorithms and tradeoffs employed in our SkipNet implementation and several peer-to-peer facilities and applications built using it.

Acknowledgements

We thank Antony Rowstron, Miguel Castro, and Anne-Marie Kermarrec for allowing us to use their Pastry implementation and network simulator. We thank Atul Adya, who independently observed that Chord's structure suggested the possibility of a Skip List-based distributed data structure, and provided helpful feedback on drafts of this paper. We also thank Scott Sheffield for his insights on the analysis of searching by name.

References

[1] J. Aspnes and G. Shah. Skip Graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, Jan. 2003.

[2] L. F. Cabrera, M. B. Jones, and M. Theimer. Herald: Achieving a Global Event Notification Service. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

[3] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach. Security for peer-to-peer routing overlays. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI)*. USENIX, December 2002.

[4] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Microsoft Research, 2002.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC) (Special issue on Network Support for Multicast Communications)*, 2002.

[6] Y.-H. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *ACM SIGMETRICS 2000*, pages 1–12, Santa Clara, CA, June 2000.

[7] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 311–320, July 2000. ICSI, Berkeley, CA, USA.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *18th ACM Symposium on Operating Systems Principles*, Oct. 2001.

[10] J. R. Douceur. The Sybil Attack. In *Proceedings of First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

[11] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22nd ICDCS*, July 2002.

[12] J. Dunagan, N. J. A. Harvey, M. B. Jones, D. Kostic, M. Theimer, and A. Wolman. FUSE: Lightweight Guaranteed Distributed Failure Notification. *Submitted for publication*, 2003.

[13] E. Fredkin. Trie Memory. *Communications of the ACM*, 3(9):490–499, Sept. 1960.

[14] Gnutella. http://www.gnutelliums.com/.

[15] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, October 2000.

[16] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Mar. 2003.

[17] N. J. A. Harvey, M. B. Jones, M. Theimer, and A. Wolman. Efficient Recovery From Organizational Disconnects in SkipNet. In *Proceedings of Second International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

[18] Herald Project. Skipnet public code release. http://research.microsoft.com/sn/Herald/, February 2004.

[19] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, July 2002.

[20] F. Kaashoek, R. Morris, F. Dabek, I. Stoica, E. Brunskill, D. Karger, R. Cox, and A. Muthitacharoen. The Chord Project, 2002. http://www.pdos.lcs.mit.edu/chord/.

[21] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigraphy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[22] P. Keleher, S. Bhattacharjee, and B. Silaghi. Are Virtualized Overlay Networks Too Much of a Good Thing? In *Proceedings of First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

[23] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.

[24] C. Labovitz and A. Ahuja. Experimental Study of Internet Stability and Wide-Area Backbone Failures. In *Fault-Tolerant Computing Symposium (FTCS)*, June 1999.

[25] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, July 2002.

[26] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *Proceed-*

ings of the First International Workshop on Peer-to-Peer Systems (IPTPS'02), MIT, March 2002.

[27] J. Mogul, L. Brakmo, D. E. Lowell, D. Subhraveti, and J. Moore. Unveiling the Transport. In *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, Nov. 2003.

[28] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Mar. 2003.

[29] T. Papadakis. *Skip Lists and Probabilistic Analysis of Algorithms*. PhD thesis, University of Waterloo, 1993. Also available as Technical Report CS93-28.

[30] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.

[31] W. Pugh. A Skip List Cookbook. Technical Report CS-TR-2286.1, University of Maryland, 1990.

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

[33] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level Multicast using Content-Addressable Networks. In *Proceedings of the Third International Workshop on Networked Group Communication*, Nov. 2001.

[34] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, Nov. 2001.

[35] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles*, Oct. 2001.

[36] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Third International Workshop on Networked Group Communications*, Nov 2001.

[37] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking*, San Jose, CA, USA, Jan. 2002.

[38] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, pages 149–160, San Diego, California, August 2001.

[39] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. Technical Report TR-819, MIT, March 2001.

[40] H. Tangmunarunkit, R. Govindan, S. Shenker, and D. Estrin. The Impact of Routing Policy on Internet Paths. In *INFOCOM*, pages 736–742, April 2001.

[41] M. Theimer and M. B. Jones. Overlook: Scalable Name Service on an Overlay Network. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, July 2002.

[42] A. Vahdat, J. Chase, R. Braynard, D. Kostic, and A. Rodriguez. Self-Organizing Subsets: From Each According to His Abilities, To Each According to His Needs. In *Proceedings of First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

[43] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proceedings of IEEE Infocom '96*, April 1996.

[44] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.