SkipNet: A Scalable Overlay Network with Practical Locality Properties

Nicholas J.A. Harvey[*,†], 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.*

1 Introduction

Scalable overlay networks, such as Chord [27], CAN [21], Pastry [23], and Tapestry [32], 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, a distributed generalization of Skip Lists [20], 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.

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

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

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 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.

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. In the case of uncorrelated, independent failures, SkipNet has similar resiliency to previous overlay networks [23, 27].

The basic SkipNet design, not including its enhancements to support constrained load balancing or network proximity-aware routing, 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 discusses design alternatives to SkipNet, Section 7 presents an experimental evaluation, and Section 8 concludes the paper.

2 Related Work

A large number of peer-to-peer overlay network designs have been proposed recently, such as CAN [21], Chord [27], Freenet [6], Gnutella [10], Pastry [23], Salad [9], Tapestry [32], and Viceroy [18]. 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 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) [11], 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 [15]).

Distributed hash tables have been used, for instance, in constructing the PAST [24] and CFS [7] distributed filesystems, the Overlook [29] scalable name service, the Squirrel [14] cooperative web cache, and scalable application-level multicast [5, 25, 22]. 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. [16] make two main points: locality is a good thing, and DHTs destroy locality. Vahdat et al. [30] 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, and constrained load balancing is supported through a combination of searches in both the string name and numeric address spaces that SkipNet defines.

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.

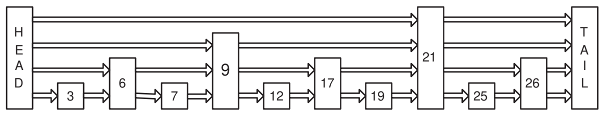

Figure 1. *A perfect Skip List.*

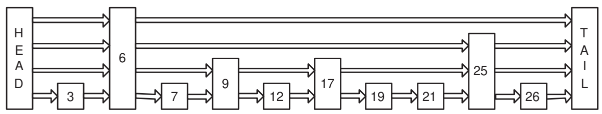

Figure 2. *A probabilistic Skip List.*

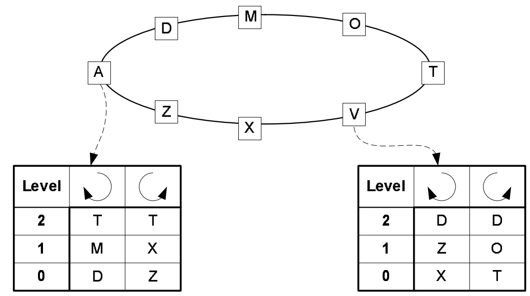

Figure 3. *SkipNet nodes ordered by name ID. Routing tables of nodes A and V are shown.*

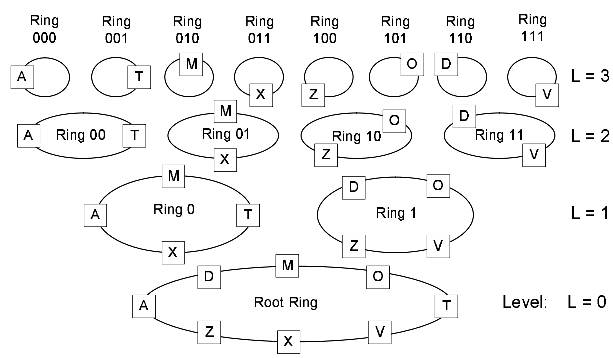

Figure 4. *The full SkipNet routing infrastructure for an 8 node system, including the ring labels.*

3.1 Analogy to Skip Lists

A Skip List, first described in Pugh [20], 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.

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 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 implicity 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 [27], 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. 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.

```
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.*

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 of this, see [12].

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 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 desti-

```
// 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*

nation 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 of this, see [12].

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.

```
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.*

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. 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 [12]).

The basic observation in handling node departures is that SkipNet can route correctly as long as the bottom level ring is maintained. All pointers but the level 0 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. In our current implementation we use a leaf set size of 16, just as Pastry does.

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 [17, 19] 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 level 0 ring is enough to ensure correct functioning of the overlay. Since each node maintains a leaf set of 16 neighbors at level 0, the level 0 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.

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 [21, 27, 23, 32], 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 7.4 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 a related paper [13].

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 [8] 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 random binary digits. However, we can also use non-binary random digits, 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. 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 7 we examine the relationship between routing performance and the amount of routing table state maintained.

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.

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. In [12], we provide an informal analysis of the performance of the P-Table routing and P-Table construction algorithms.

5.3.1 P-Table Construction

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. After filling in any possible intervals, the node checks whether any of the intervals are still empty. If so, the node forwards the join message, using its own P-Table entries, towards an unfilled interval. [12] explains why forwarding the join message to the the farthest unfilled interval from the joining node yields the smallest expected number of insertion forwarding hops. If all the intervals have at least one candidate, the node sends the completed join message back to the original joining node.

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 7.6 summarizes a performance evaluation of these two approaches.

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 [23]. Further details of the C-Table data structure and construction process can be found in [12].

6 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.

6.1 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 [5, 25, 22] and load-aware replication [29].
- The ability to reach named destinations independent of the availability of the DNS name lookup service.

6.2 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.

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.

6.3 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, interdomain 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.

7 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 [23]. Our Chord implementation is based on the algorithm in [27], adapted to operate within our simulator. 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. In addition to the information provided below, we provide a complete specification of all the configuration parameters used in our simulation runs in [12].

All our experiments were run both on a Mercator topology [28] and a GT-ITM topology [31]. 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.

7.1 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 hops: The absolute length of the overlay path between two nodes, measured in IP-level hops.

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 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. We ensure that the selected AS has at least 1/10-th as many core router nodes as overlay nodes. For Zipf-clustered, we place organizations within ASes, as before. However, the nodes of an organization are spread throughout its AS as follows: A "root" physical node is randomly placed within the AS and all overlay nodes are placed relative to this root, at distances modelled by a Zipf-like distribution. In this configuration most of the overlay nodes of an organization will be closely clustered together within their AS. 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.

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 [26]. 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.

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 [27, 23].

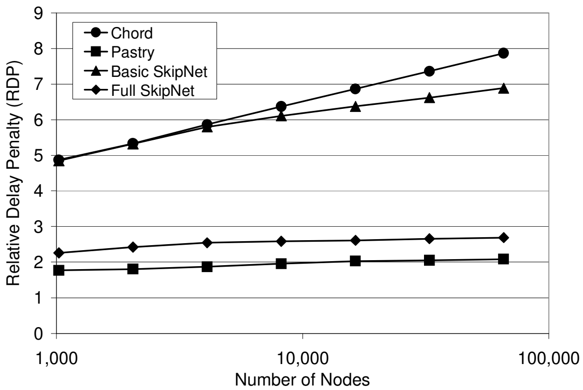

Figure 8. *RDP as a function of network size. Configuration: 1000 organizations with Zipf-like sizes, nodes and data names are Zipf-clustered.*

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.*

7.2 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 8 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.

7.3 Exploiting Locality of Placement

RDP only measures performance relative to IP-based routing. However, one of SkipNet's key benefits is that it enables localized placement of data. Figure 9 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

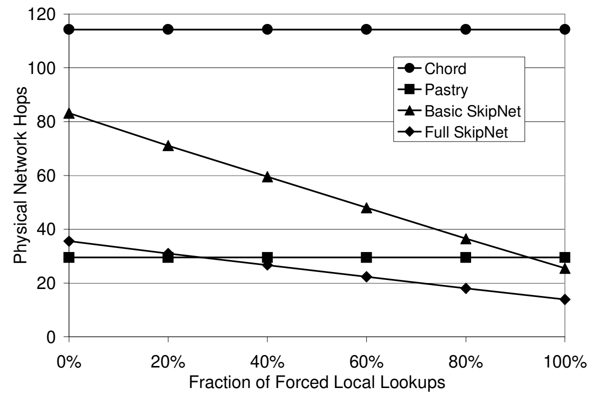

Figure 9. *Absolute latency (in network hops) for lookups as a function of data access locality (percentage of lookups forced to be within a single organization). Configuration: 2^{16} nodes, 100 organizations with Zipf-like sizes, nodes and data names are Zipf-clustered.*

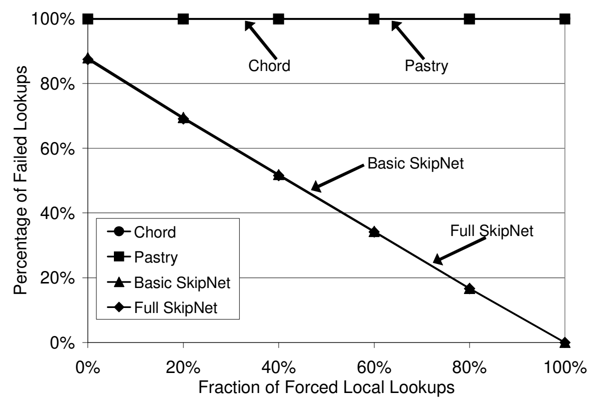

Figure 10. *Number of failed lookup requests as a function of data access locality (percentage of lookup requests forced to be within a single organization) for a disconnected organization. Configuration: 2^{16} nodes, 100 organizations with Zipf-like sizes, nodes and data names are Zipf-clustered.*

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 9 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.

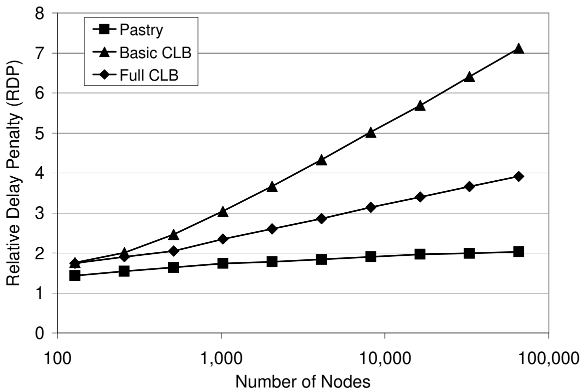

Figure 11. *RDP of lookups for data that is constrained load balanced (CLB) as a function of network size. Configuration: 100 organizations with Zipf-like sizes, nodes and data names are Zipf-clustered.*

7.4 Fault Tolerance

Content locality also improves fault tolerance. Figure 10 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%.

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.

7.5 Constrained Load Balancing

Figure 11 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.

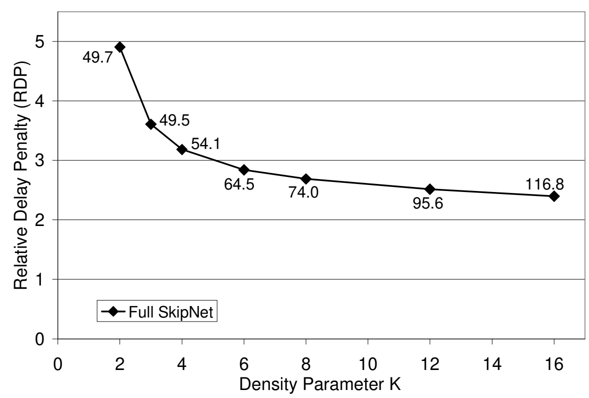

Figure 12. *RDP for Full SkipNet as a function of the density configuration parameter k. The labels next to each point represent the average number of unique pointers per node. Configuration: 2^{16} nodes, 1000 organizations with Zipf-like sizes, nodes and data names are Zipf-clustered.*

7.6 Network Proximity

Figure 12 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 12 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.

8 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. 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].

Acknowledgements

We thank John Dunagan for his substantial help with both the analysis of SkipNet and with crafting the text of this paper. 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. Finally, we thank the anonymous conference referees for their insightful comments.

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

[6] 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.

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

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

[9] 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.

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

[11] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proceedings of the Fourth OSDI*, Oct. 2000.

[12] N. J. A. Harvey, J. Dunagan, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. Technical Report MSR-TR-2002-92, Microsoft Research, 2002.

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

[14] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *Proceedings of the 21st Annual PODC*, July 2002.

[15] 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 STOC*, May 1997.

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

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

[18] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proceedings of the 21st Annual PODC*, July 2002.

[19] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Proceedings of 4th USITS*, Mar. 2003.

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

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

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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 ACM SIGCOMM*, Aug. 2001.

[28] H. Tangmunarunkit, R. Govindan, S. Shenker, and D. Estrin. The Impact of Routing Policy on Internet Paths. In *Proceedings of IEEE INFOCOM 2001*, April 2001.

[29] M. Theimer and M. B. Jones. Overlook: Scalable Name Service on an Overlay Network. In *Proceedings of the 22nd ICDCS*, July 2002.

[30] 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 *First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

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

[32] 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, UC Berkeley, April 2001.