

# Symphony: Distributed Hashing In A Small World

Gurmeet Singh Manku  
Stanford University  
manku@cs.stanford.edu

Mayank Bawa  
Stanford University  
bawa@cs.stanford.edu

Prabhakar Raghavan  
Verity Inc.  
pragh@verity.com

## Abstract

*We present Symphony, a novel protocol for maintaining distributed hash tables in a wide area network. The key idea is to arrange all participants along a ring and equip them with long distance contacts drawn from a family of harmonic distributions. Through simulation, we demonstrate that our construction is scalable, flexible, stable in the presence of frequent updates and offers small average latency with only a handful of long distance links per node. The cost of updates when hosts join and leave is small.*

## 1 Introduction

Peer to peer file sharing applications have surged in popularity in recent years. Systems like Napster, Gnutella, Kazaa and Freenet [4] have been used by millions of users. The research community is working on a wide variety of peer to peer applications like persistent data storage (CFS [6], Farsite [3], Oceanstore [11], PAST [19]), event notification and application level multicast (Bayeux [23], Scribe [20] and CAN-based Multicast [17]), DNS [5], resource discovery [2] and cooperative web caching [9]. Several of these applications have no centralized components and use a scalable distributed hash table (DHT) [8, 13, 16, 18, 22] as a substrate.

A DHT is a self-organizing overlay network of hosts that provides a service to add, delete and look up hash keys. Consider a network of  $n$  hosts over a wide area network that wish to cooperatively maintain a DHT with entries that change frequently. Replicating the entire hash table  $n$  times is unfeasible if  $n$  is large. One solution is to split the hash table into  $n$  blocks and let each host manage one block. This now requires a mapping table that maps a block id to its manager's network id. If  $n$  is of the order of hundreds or thousands of hosts and if hosts have short lifetimes, replicating this mapping table  $n$  times might be challenging. One possibility is to store this mapping table at a single central server, which would be consulted for every hash

lookup. The load on the central server can be reduced by caching mapping table entries with DNS-style leases. However, a central server is a single point of failure and has to bear all traffic related to arrivals and departures.

Researchers have recently proposed distributed hashing protocols [8, 13, 16, 18, 22] that do not require any central servers. The mapping table is not stored explicitly anywhere. Instead, hosts configure themselves into a structured network such that mapping table lookups require a small number of hops. Designing a practical scheme along these lines is challenging because of the following desiderata:

*Scalability:* The protocol should work for a range of networks of arbitrary size.

*Stability:* The protocol should work for hosts with arbitrary arrival and departure times, typically with small lifetimes.

*Performance:* The protocol should provide low latency for hash lookups and low maintenance cost in the presence of frequent joins and leaves.

*Flexibility:* The protocol should impose few restrictions on the remainder of the system. It should allow for smooth trade-offs between performance and state management complexity.

*Simplicity:* The protocol should be easy to understand, code, debug and deploy.

We propose Symphony, a novel distributed hashing protocol that meets the criteria listed above. The core idea is to place all hosts along a ring and equip each node with a few *long distance links*. Symphony is inspired by Kleinberg's Small World construction [10]. We extend Kleinberg's result by showing that with  $k = O(1)$  links per node, it is possible to route hash lookups with an average latency of  $O(\frac{1}{k} \log^2 n)$  hops. Among the advantages Symphony offers over existing DHT protocols are the following:

*Low State Maintenance:* Symphony provides low average hash lookup latency with fewer TCP connections per node than other protocols. Low degree networks reduce the number of open connections and ambient traffic corresponding to pings, keep-alives and control information. Moreover, sets of nodes that participate in locking and coordination for dis-

tribute state update are smaller sized.

*Fault Tolerance:* Symphony requires  $f$  additional links per node to tolerate the failure of  $f$  nodes before a portion of the hash table is lost. Unlike other protocols, Symphony does not maintain backup links for each long distance contact.

*Degree vs. Latency Tradeoff:* Symphony provides a smooth tradeoff between the number of links per node and average lookup latency. It appears to be the only protocol that provides this tuning knob *even* at run-time. Symphony does not dictate that the number of links be identical for all nodes. Neither is the number stipulated to be a function of current network size nor is it fixed at the outset. We believe that these features of Symphony provides three benefits: (a) support for heterogeneous nodes, (b) incremental scalability, and (c) flexibility. We discuss these further in Section 5.3.

**Road map:** Section 2 explores related work. Section 3 describes Symphony. Section 4 contains experimental results. In Section 5, we compare Symphony with other protocols. Section 6 concludes the paper.

## 2 Related Work

### 2.1 Distributed Hash Tables

Plaxton, Rajaraman and Richa [15] devised a routing protocol based on hypercubes for a static collection of nodes. Routing is done by *digit-fixing*: e.g., when a node with id *zedstu* receives a query for *zedaaa*, it forwards it to a neighbor with prefix *zeda*. It turns out that for  $b$  bits per digit, each neighbor must maintain  $O((2^b \log n)/b)$  neighbors resulting in  $O((\log n)/b)$  worst case routing latency. Tapestry [8] adapted this scheme to a dynamic network for use in a global data storage system [11]. Pastry [18] is another scheme along the same lines where a node forwards a query to a neighbor with the longest matching prefix. In both Tapestry and Pastry, the number of bits per digit  $b$  is a configurable parameter that remains fixed at run-time.

CAN [16] embeds the key-space into a torus with  $d$  dimensions by splitting the key into  $d$  variable-length digits. A node forwards a query to the neighbor that takes it closer to the key. Nodes have  $O(d)$  neighbors and routing latency is  $O(dn^{1/d})$ . The number of dimensions  $d$  is fixed in CAN. If the final network size can be estimated, then  $d$  could be made  $O(\log n)$ , resulting in  $O(\log n)$  routing latency and  $O(\log n)$  neighbors.

Chord [22] places participating nodes on a circle with unit perimeter. An  $m$ -bit hash key  $K$  is treated

as a fraction  $K/2^m$  for routing. Each node maintains connections with its immediate neighbors along the circle and a *finger table* of connections with nodes at distances approximately  $\langle \frac{1}{2}, \frac{1}{4}, \frac{1}{8} \dots \rangle$  along the circle. Routing is done by forwarding to the node closest to, but not past, the key being looked up. Chord requires  $O(\log n)$  neighbors and provides  $O(\log n)$  routing latency. The protocols for joining and leaving the network introduce complexity and require  $O(\log^2 n)$  messages each. A stabilization protocol is required to maintain network integrity.

Viceroy [13] is the first proposal that provides  $O(\log n)$  routing latency with only a constant number of links. Like Chord, nodes are placed along a circle. A node additionally belongs to one out of approximately  $O(\log n)$  concentric rings lying one above the other. These rings correspond to layers in Butterfly networks. A node maintains connections with two neighbors each along the two rings it belongs to. It also maintains two connections to a pair of nodes in a lower ring and one connection with a node in the ring above. Routing requires  $O(\log n)$  hops on average.

DHT's over clusters have been extensively studied by the SDDS (Scalable Distributed Data Structures) community in the 90's. The term was coined by Litwin, Niemat and Shneider in their seminal paper [12]. Gribble et al. [7] implemented a highly scalable, fault tolerant and available SDDS on a cluster for Internet services. The requirements for a DHT over wide area networks are very different.

### 2.2 Small World Networks

Milgram conducted a celebrated experiment [14] that demonstrated the *Small World* phenomenon. He discovered that pairs of people in a society were connected by short chains of acquaintances. He also discovered that people were actually able to route letters to unknown persons in a few hops by forwarding them along a short path through acquaintanceships. To model the Small World phenomenon, Kleinberg [10] recently constructed a two-dimensional grid where every point maintains four links to each of its closest neighbors and just *one* long distance link to a node chosen from a suitable probability function. He showed that a message can be routed to any node by *greedy routing* in  $O(\log^2 n)$  hops. Barriere et al. [1] studied Kleinberg's construction and proved its optimality under certain conditions.

Our work is inspired by Kleinberg's construction. We extend his result by showing that with  $k = O(1)$  links, the routing latency diminishes to  $O(\frac{1}{k} \log^2 n)$

hops. We also show how this basic idea can be adapted and engineered into a practical protocol for maintaining DHTs in a peer to peer network.

### 3 Symphony: The Protocol

Let  $I$  denote the unit interval  $[0, 1)$  that wraps around. It is convenient to imagine  $I$  as a circle (ring) with unit perimeter. Whenever a node arrives, it chooses as its id a real number from  $I$  uniformly at random. A node manages that sub-range of  $I$  which corresponds to the segment on the circle between its own id and that of its immediate clockwise predecessor. A node maintains *two short links* with its immediate neighbors. Since all nodes choose their id's uniformly from  $I$ , we expect that they manage roughlyly equi-sized sub-ranges.

The nodes cooperatively maintain a distributed hash table. If a hash function maps an object to an  $m$ -bit hash key  $K$ , then the manager for this hash entry is the node whose sub-range contains the real number  $K/2^m$ . No restriction on  $m$  is imposed. Unlike CAN, Pastry and Tapestry, there is no relationship between  $m$  and the number or links.

#### 3.1 Long Distance Links

Every node maintains  $k \geq 1$  *long distance links*. For each such link, a node first draws a random number  $x \in I$  from a probability distribution function that we will shortly define. Then it contacts the manager of the point  $x$  away from itself in the clockwise direction by following a *Routing Protocol* which we describe in Section 3.2. Finally, it attempts to establish a link with the manager of  $x$ .

We ensure that the number of incoming links per node is bounded by placing an upper limit of  $2k$  incoming links per node. Once the limit is reached, all subsequent requests to establish a link with this node are rejected. The requesting node then makes another attempt by re-sampling from its probability distribution function. As a practical matter, an upper bound is placed on the number of such attempts, before a node gives up. We also ensure that a node does not establish multiple links with another node.

*Probability distribution function (pdf)*: We denote the pdf by  $p_n$ , where  $n$  denotes the current number of nodes. The function  $p_n(x)$  takes the value  $1/(x \ln n)$  when  $x$  lies in the range  $[1/n, 1]$ , and is 0 otherwise. Drawing from  $p_n$  corresponds to a simple C expression: `exp (log(n) * (drand48() - 1.0))`, where `drand48()` produces a random number between 0 and 1. It is the continuous version of the discrete pdf proposed by Kleinberg. The distribution  $p_n$  belongs to

a family of harmonic distributions. This observation inspired the name Symphony.

*Estimation of  $n$* : Drawing from pdf  $p_n$  poses a problem: a node needs to know  $n$  to begin with. However, it is difficult for all nodes to agree on the exact value of current number of participants  $n$ , especially in the face of nodes that arrive and depart frequently. In Section 3.4, we will describe an *Estimation Protocol* that helps each of the nodes track  $n$  at run time. We denote an estimate of  $n$  by  $\tilde{n}$ . Thus a node draws from  $p_{\tilde{n}}$  instead of  $p_n$ .

*Choice of  $k$* : The number of links established by each node is a design parameter that is not fixed by our protocol. We experimentally show that as few as four long distance links are sufficient for low latency routing in large networks.

*Long Distance Links in Practice*: In a network with hosts spanning continents, we would have to embed the set of participating hosts onto a circle, taking network proximity into account. We expect this to require a fair amount of engineering and we are currently working on this problem. Once the circle has been established, we expect Symphony to be able to route lookups such that the latency does not exceed IP latency between the source and the final destination by a large factor. For example, CAN [16] demonstrated that a factor of two could be achieved for their construction for roughlyly 130K nodes.

The phrase "a network with  $k$  links per node" denotes a network where each node establishes 2 short links and  $k$  long links with other nodes. In terms of TCP connections, each node actually maintains an average of  $t = 2k + 2$  connections.

#### 3.2 Unidirectional Routing Protocol

When a node wishes to lookup a hash key  $x \in I$ , it needs to contact the manager of  $x$ .

*A node forwards a lookup for  $x$  along that link (short or long) that minimizes the clockwise distance to  $x$ .*

Kleinberg [10] analyzed a static network in which each participant knows  $n$  precisely, has one long distance link corresponding to  $p_n$ , manages a sub-range of size  $1/n$  and always routes clockwise. He showed that for  $k = 1$  the expected path length for greedy routing is  $O(\log^2 n)$  hops. We can show that in general, if each node has  $k = O(1)$  links, expected path length is  $O(\frac{1}{k} \log^2 n)$ .

**Theorem 3.1** *The expected path length with unidirectional routing in an  $n$ -node network with  $k = O(1)$  links is  $O(\frac{1}{k} \log^2 n)$  hops.*

**Proof:**

We sketch the proof assuming every attempted long-distance link is successful. However as noted above in Section 3.1, some of these connections in fact are rejected because the intended target of the link is “saturated” with  $2k$  incoming links. We account for this implementation detail by noting that provided  $k = O(1)$ , the fraction of such rejected links is a constant and this only inflates the expected number of hops by a constant. We also assume that all nodes have accurate knowledge of  $n$ .

The pdf that we use for generating long distance neighbors is  $p_n(x) = 1/(x \log n)$  for  $x \in [1/n, 1]$ . The probability  $p_{\text{half}}$  of drawing a value from  $[z/2, z]$  for any  $z \in [2/n, 1]$  is given by  $\int_{z/2}^z p_n(x) dx = 1/\log_2 n$ , which is independent of  $z$ . The significance of  $p_{\text{half}}$ : regardless of the current distance to the destination, it is the probability that any single long-distance link will cut the distance by at least half. The number of links to consider before the current distance diminishes by at least half follows a geometric distribution with mean  $1/p_{\text{half}} = \log_2 n$ . With  $k$  links per node, the expected number of nodes to consider before the current distance is at least halved is  $\lceil (\log_2 n)/k \rceil$ , which is less than  $(2 \log_2 n)/k$  for  $k \leq \log_2 n$ .

Successive nodes along the route of a hash lookup diminish the distance to the destination. Consider that portion of the route where the distance is more than  $2/n$ . We showed that the expected number of nodes along the path before we encounter a node that more than halves the current distance is at most  $(2 \log_2 n)/k$ . The maximum number of times the original distance could possibly be halved before it is less than  $2/n$  is  $\log_2(n/2)$ . Thus, the expected number of nodes along the route before the distance is less than  $2/n$  is at most  $2(\log_2 n)(\log_2(n/2))/k$ . The remainder of the path consists of nodes that diminish the distance from  $2/n$  to 0. The average contribution of these small path lengths is  $O(1)$  because each node chose its id in the interval  $[0, 1)$  uniformly at random. Thus the average path length of the full route is  $O(\frac{1}{k} \log^2 n)$ . ◉

It is important that links be chosen following a harmonic distribution. Using the proof technique in [10], it can be shown that the average latency is  $\Omega(\sqrt{n/k})$  if  $k$  links are chosen uniformly at random from the interval  $[0, 1)$ . Section 4.9 presents a graphical illustration of this observation.

### 3.3 Bidirectional Routing Protocol

In Section 3.1, we described how nodes establish long links with other nodes. The average number of *incoming links* is  $k$  per node. In practice, a link between two nodes would see continuous routing traffic. This would be materialized as a bidirectional TCP connection to leverage TCP’s flow control, duplicate elimination and in-order delivery guarantees. If we were to use UDP, we would have to replicate much of this functionality.

One way to leverage incoming links is to treat them as additional long distance links. This helps reduce average latency only marginally. Much more benefit can be obtained by exploiting the following insight: The distribution of the source id of an incoming link corresponds roughly to  $p_n$  but in the anticlockwise direction. The observation that a node has exactly  $k$  clockwise and roughly  $k$  anticlockwise long distance links motivates the following protocol:

*A node routes a lookup for  $x$  along that link (incoming or outgoing) that minimizes the absolute distance to  $x$ .*

**Theorem 3.2** *The expected path length with bidirectional routing in an  $n$ -node network with  $k = O(1)$  links is  $O(\frac{1}{k} \log^2 n)$  hops.*

**Proof:** Along the same lines as Theorem 3.1. ◉

Bidirectional routing improves average latency by roughly 25% to 30% (See Section 4.2). Note that if we minimize absolute distance but restrict ourselves to clockwise movement, it is possible to get infinite loops.

With Bidirectional Routing, the average latency is still  $O(\frac{1}{k} \log^2 n)$ . However, the constant hidden behind the big-O notation is less than 1. We experimentally show that coupled with the idea of 1-Lookahead (see Section 3.7), networks as large as  $2^{15}$  nodes have average latency no more than 7.5 for  $k = 4$ .

### 3.4 Estimation Protocol

Our Estimation Protocol is based on the following insight: Let  $X_s$  denote the sum of segment lengths managed by any set of  $s$  distinct nodes. Then  $\frac{s}{X_s}$  is an unbiased estimator for  $n$ . The estimate improves as  $s$  increases. The idea of estimating  $n$  in this fashion is borrowed from Viceroy [13] where it was also shown that the estimate is asymptotically tight when  $s = O(\log n)$ . When the Estimation Protocol executes, all  $s$  nodes that contributed their segment lengths update their own estimates of  $n$  as well.

*Choice of  $s$ :* Our experiments show that  $s = 3$  is good enough in practice. A node estimates  $n$  by using the length of the segment it partitions and its two neighboring segments. These three segment lengths are readily available at no extra cost from the two nodes between which  $x$  inserts itself in the ring. In Section 4.1, we show that the impact of increasing  $s$  on average latency is insignificant.

We note that an implementation of Symphony might employ a different Estimation Protocol. For example, an accurate value for  $n$  might be available from some central server where all participants must register before participating in the network. In another scenario, estimates of  $n$  could be piggybacked along with a small fraction of normal lookup messages, thereby amortizing the cost of maintaining  $\tilde{n}$ . A node might maintain the harmonic mean of the last few estimates it comes across. In this paper, we do not delve into sophisticated protocols for estimating  $n$ . For our purposes, a simple Estimation Protocol with  $s = 3$  segments works fine.

### 3.5 Join and Leave Protocols

**Join:** To join the network, a new node must know at least one existing member. It then chooses its own id  $x$  from  $[0, 1)$  uniformly at random. Using the Routing Protocol, it identifies node  $y$ , the current manager of  $x$ . It then runs the Estimation Protocol using  $s = 3$ , updating the estimates of three other nodes as well. Let  $\tilde{n}_x$  denote the estimate of  $n$  thus produced. Node  $x$  then uses pdf  $p_{\tilde{n}_x}$  to establish its long distance links. Since each link establishment requires a lookup that costs  $O(\frac{1}{k} \log^2 n)$  messages, the total cost of  $k$  link establishments is  $(\log^2 n)$  messages. The constant hidden behind the big-O notation is actually less than 1. See Section 4.6 for actual costs determined experimentally.

**Leave:** The departure of a node  $x$  is handled as follows. All outgoing and incoming links to its long distance neighbors are snapped. Other nodes whose outgoing links to  $x$  were just broken, reinstate those links with other nodes. The immediate neighbors of  $x$  establish short links between themselves to maintain the ring. Also, the successor of  $x$  initiates the Estimation Protocol over  $s = 3$  neighbors, each of whom also updates its own estimate of  $n$ . The departure of a node requires an average of  $k$  incoming links to be re-established. The expected cost is  $O(\log^2 n)$  messages. Again, the constant hidden behind the big-O notation is less than 1. See Section 4.6 for actual costs determined experimentally.

### 3.6 Re-linking Protocol

Each node  $x$  in the network maintains two values:  $\tilde{n}_x$ , its current estimate of  $n$  and  $\tilde{n}_x^{link}$ , the estimate at which its long distance links were last established. Over its lifetime,  $\tilde{n}_x$  gets updated due to the Estimation Protocol being initiated by other nodes. Whenever  $\tilde{n}_x \neq \tilde{n}_x^{link}$ , it is true that the current long distance links of  $x$  correspond to a stale estimate of  $n$ . One solution is to establish all links afresh. However, if a node were to re-link on *every* update of  $\tilde{n}_x$ , traffic for re-linking would be excessive. This is because re-establishment of all  $k$  long distance links requires  $O(\log^2 n)$  messages.

*Re-linking Criterion:* A compromise re-linking criterion that works very well is to re-link only when the ratio  $\tilde{n}_x / \tilde{n}_x^{link} \notin [1/2, 2]$ . The advantage of this scheme is that as  $n$  steadily grows or shrinks, traffic for re-linking is smooth over the lifetime of the network. In particular, if nodes arrive sequentially and each node knows  $n$  precisely at all times, then the number of nodes re-linking at any time would be at most one. We experimentally show that even in the presence of imprecise knowledge of  $n$ , the re-linking cost is smooth over the lifetime of a network. However, we also show that the benefits of re-linking are marginal.

### 3.7 Lookahead

Two nodes connected by a long link could periodically exchange some information piggy-backed on keep-alives. In particular, they could inform each other about the positions of their respective long distance contacts on the circle. Thus a node can learn and maintain a list of all its neighbor's neighbors. We call this the lookahead list. The lookahead list helps to improve the choice of neighbor for routing queries. Let  $u$  denote the node in the list that takes a query closest to its final destination. Then the query is routed to that neighbor that contains  $u$  in its neighbor set. Note that we do not route directly to  $u$ . The choice of neighbor is not greedy anymore. If hash lookups are exported by an RPC-style interface, (e.g., iterative/non-recursive queries in DNS), we *could* forward a client to a neighbor's neighbor, cutting down average latency by half. Upon receiving a forwarded lookup request, a neighbor makes a fresh choice for its own best neighbor to route to.

We experimentally show that 1-Lookahead effectively reduces average latency by roughly 40%.

What is the cost of 1-Lookahead? The size of the lookahead list is  $O(k^2)$ . The number of long links remains unchanged because a node does not

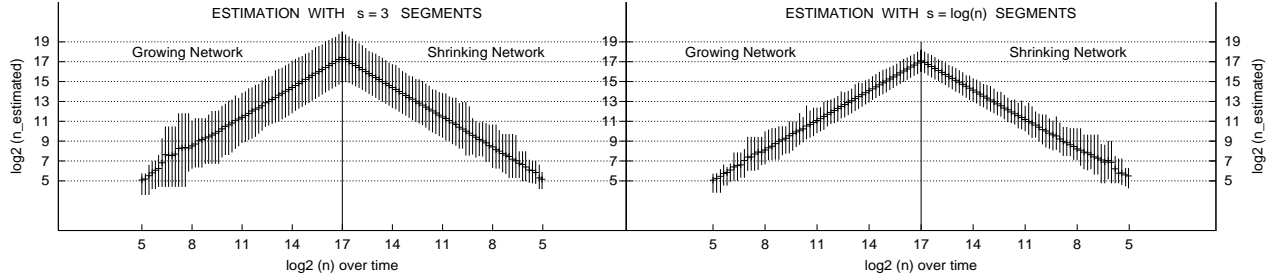


Figure 1: *Quality of estimated value of  $n$  as the network first expands and then shrinks. Each vertical line segment plots the average along with an interval that captures 99% of the distribution.*

directly link to its neighbors’ neighbors; it just remembers their id’s. However, arrival and departure of any node requires an average of  $k(2k+2)$  messages to update lookahead lists at  $k(2k+2)$  nodes in the immediate neighborhood. These messages need not be sent immediately upon node arrival/departure. They are sent lazily, piggy-backed on normal routing packets or keep-alives exchanged between pairs of nodes. Lazy update of lookahead lists might introduce temporary inconsistencies. This is acceptable because routing does not crucially depend on these lists. Lookaheads just provide a better hint. We are currently investigating further the role of approximate lookahead.

We could employ  $\ell$ -Lookahead in general, for  $\ell > 1$ . However, the cost of even 2-Lookahead becomes significant since each update to a link would now require  $O(k^3)$  additional messages for updating lookahead lists. If  $\ell$  were as large as  $O(\log^2 n)$ , each node could effectively compute the shortest path to any destination.

## 4 Experiments

In this section, we present results from our simulation of Symphony on networks ranging from  $2^5$  to  $2^{15}$  nodes in size. We systematically show the interplay of various variables ( $n$ ,  $k$  and  $s$ ), justifying our choices.

We study four kinds of networks: A STATIC network with  $n$  nodes is constructed by placing  $n$  nodes on a circle, splitting it evenly into  $n$  segments. Knowledge of  $n$  is global and accurate. An EXPANDING network is one that is constructed by adding nodes to the network sequentially. An estimate of  $n$  is used to establish long distance links. An EXPANDING-RELINK network is simply an EXPANDING network in which nodes re-establish links using the re-linking criterion mentioned in Section 3.6. Finally, a DYNAMIC network is one in which nodes not

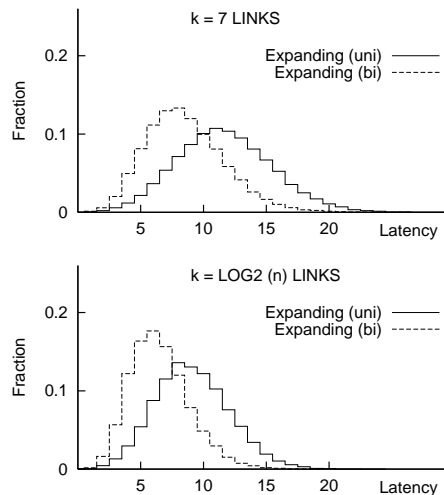


Figure 2: *Latency distributions for a network with  $2^{14}$  nodes.*

only arrive but also depart. We describe the exact arrival and departure distributions in Section 4.4.

### 4.1 Estimation Protocol

Figure 1 shows performance of the Estimation Protocol when a network grew from zero to  $2^{17}$  nodes and then shrank. Each vertical segment in the figure captures 99% of the nodes. The Estimation Protocol tracks  $n$  fairly well. The estimate is significantly improved if we use  $s = \log \tilde{n}$  neighbors, where  $\tilde{n}$  itself is obtained from any existing node. However, the impact on average latency is not significant, as we show in Section 4.3. All our experiments described hereafter were conducted with  $s = 3$ .

### 4.2 Routing Protocol

Figure 3 plots the average latency for three networks: STATIC, EXPANDING and EXPANDING-RELINK. The

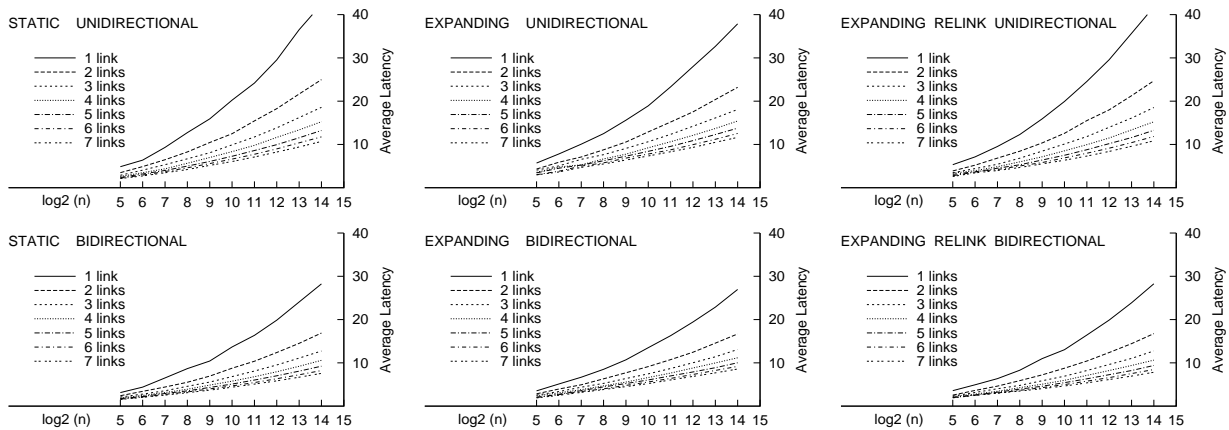


Figure 3: Average latency for various numbers of long distance links and  $n$  ranging from  $2^5$  and  $2^{14}$ .

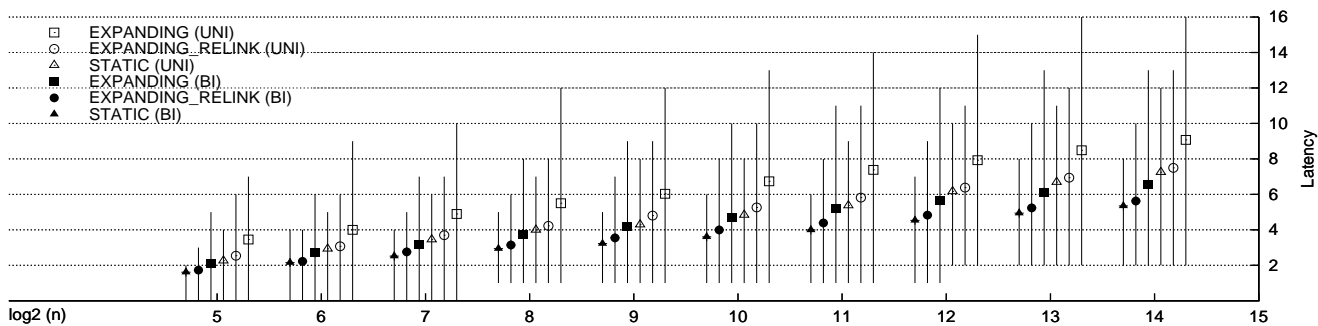


Figure 4: Latency for various networks with  $\log_2 \tilde{n}$  links per node. Each vertical segment plots the average along with an interval that captures 99% of the distribution.

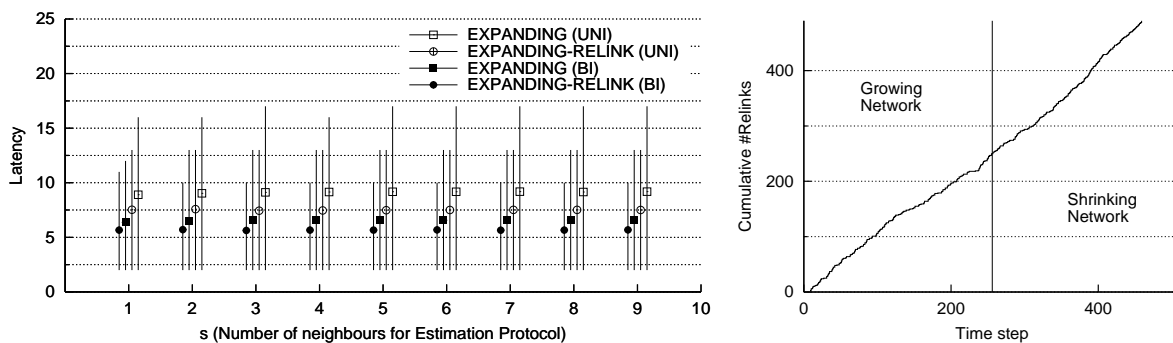


Figure 5: **Left:** Latency for EXPANDING networks using Estimation Protocol with various values of  $s$ , the number of neighbors contacted for estimating  $n$ . **Right:** Cumulative number of re links for a network that first expands from 0 to 256 nodes and then shrinks back to 0. At every time step, exactly one node joins or leaves.

number of links per node is varied from 1 to 7. Increasing the number of links from 1 to 2 reduces latency significantly. However, *successive additions have diminishing returns*. This is one reason that re-linking has marginal benefits. However, Bidirectional routing is a good idea as it improves latency by roughly 25% to 30%.

Figure 2 shows the latency distribution for various networks with either 7 or  $\log_2 \tilde{n}$  links each. The variance of latency distribution is not high. Having  $\log_2 \tilde{n}$  links per node not only diminishes average latency but also the variance significantly.

Figure 4 plots latency for a network in which each node maintains  $\log_2 \tilde{n}$  links. The vertical segments capture 99% of node-pairs. For a given type of network, average latency grows linearly with  $\log n$ , as expected.

### 4.3 Re-linking Protocol

In Figure 5, we plot average latency as  $s$ , the number of neighbors in the estimation protocol, varies. We observe that average latency is relatively insensitive to the value of  $s$  used. This justifies our choice of  $s = 3$ . Figure 5 also shows the cost of re-linking over the lifetime of a network that first expands to 256 nodes and then shrinks back to zero. Exactly one node arrives or leaves at any time step. We chose a network with small  $n$  for Figure 5 to highlight the kinks in the curve. For large  $n$ , the graph looks like a straight line. The cost of re-linking is fairly smooth.

### 4.4 Dynamic Network

A DYNAMIC network is one in which nodes arrive and depart. We studied a network with  $n = 100K$  nodes having  $\log \tilde{n}$  neighbors each. Each node alternates between two states: alive and asleep. Lifetime and sleep-time are drawn from two different exponential distributions with means 0.5 hours and 23.5 hours respectively. We grow the node pool linearly over a period of one day so that all 100K nodes are members of the pool at the end of the first day. During the second day, the pool remains constant. The third day sees the demise of random nodes at regular intervals so that the pool shrinks to zero by the end of 72 hours. The average number of nodes that participate in the ring at any time is  $\frac{0.5}{0.5+23.5} \times 100K \approx 2K$ . From Figure 6, we see that the Estimation Protocol is able to track  $n$  sufficiently accurately and that the average latency was always less than 5 hops.

We wish to point out that the network we simulated is very dynamic. The set of nodes at any point

of time is quite different from the set of nodes one hour earlier. This is because the average lifetime of any node is only 0.5 hour. A real-life network will have some nodes with longer lifetimes and variable distributions [3, 21]. Our current model is simple but sufficient to highlight the stability of Symphony in the presence of high activity.

### 4.5 Lookahead

Figure 7 shows the efficacy of employing 1-Lookahead when  $k$  is small. Average latency diminishes by around 40% with 1-Lookahead. Moreover, the spread of latency distribution (captured by vertical line segments in the graph) shrinks. For a network with  $2^{15}$  nodes, average latency is 7.6 with  $k = 4$ . We also simulated 1-Lookahead with  $k = \log \tilde{n}$  links per node and saw average latency drop to 4.4.

Note that 1-Lookahead does not entail an increase in the number of long links per node. Only neighbor-lists are exchanged between pairs of nodes periodically. This does not incur extra cost because increments to neighbor-lists are piggy-backed on normal routing traffic or keep-alives.

### 4.6 Cost of Joining and Leaving

Figure 8 plots the cost of joining and leaving the network. Whenever a node joins/leaves,  $k$  long distance links have to be created apart from updates to short links. Join/leave cost is proportional to the number of links to be established. The cost diminishes as average lookup latency drops. When using 1-Lookahead, there are an additional  $k(2k + 2)$  messages to update lookahead lists. However, these are exchanged lazily between pairs of nodes, piggy-backed on keep-alives. The cost of join/leave is  $O(\log^2 n)$ . Figure 8 clearly shows that the constant in the big-O notation is less than 1. For example, in a network of size  $2^{14}$ , we need only 20 messages to establish  $k = 4$  long links.

### 4.7 Load Balance

Figure 9 plots the number of messages processed per node in a network of size  $2^{15}$  corresponding to  $2^{15}$  lookups. Each lookup starts at a node chosen uniformly at random. The hash key being looked up is also drawn uniformly from  $[0, 1]$ . The routing load on various nodes is relatively well balanced. Both the average and the variance drop when we employ 1-Lookahead. Curiously, the distribution is bimodal

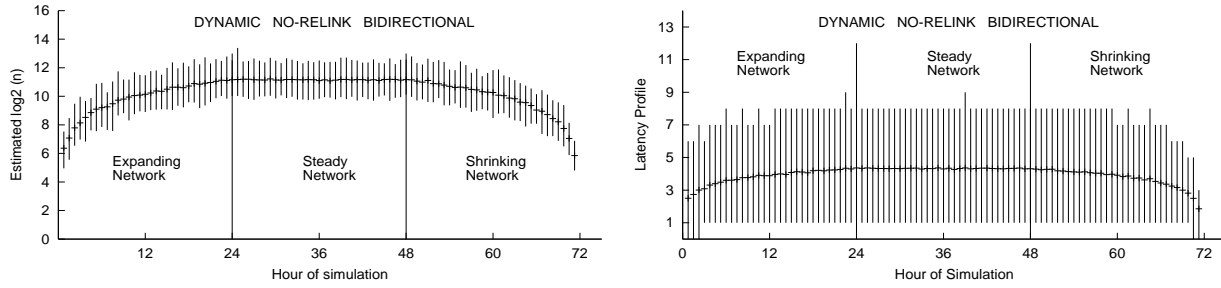


Figure 6: Performance of a DYNAMIC network of 100K nodes with  $\log \tilde{n}$ -links using the Estimation Protocol but no re-linking. Each node is alive and asleep on average for 0.5 hours and 23.5 hours respectively. The node pool linearly increases to 100K over the first day. The pool is steady on the second day. A random node departs at regular intervals on the third day until the network shrinks to zero. Each vertical segment plots the average along with the range of values that covers 99% of the distribution.

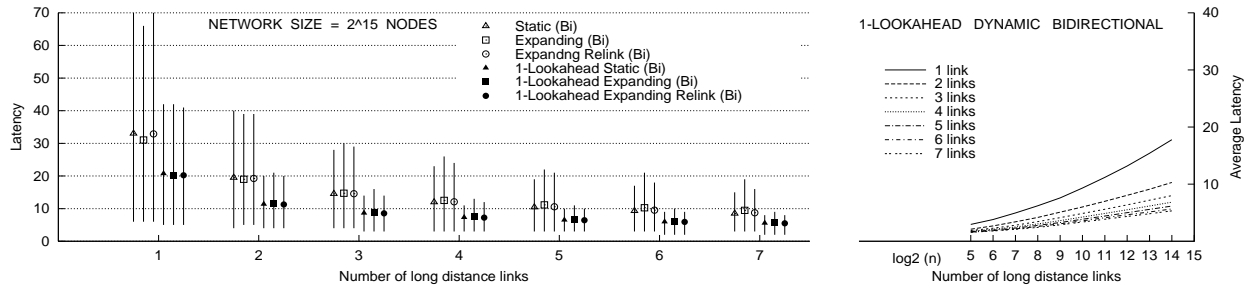


Figure 7: Impact of using 1-Lookahead in routing in a typical network with  $2^{15}$  nodes.

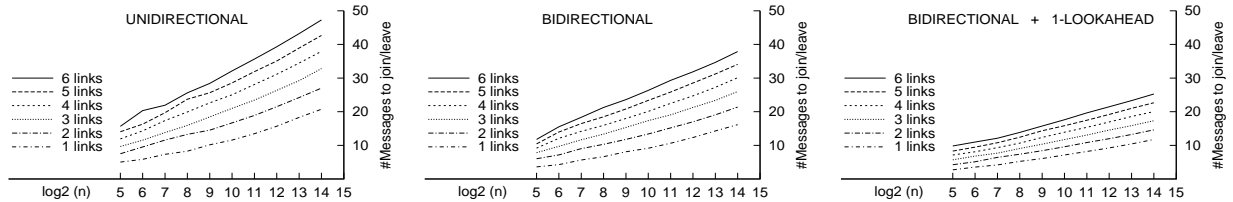


Figure 8: Cost of joining and leaving in a network with  $n = 2^{15}$  nodes.

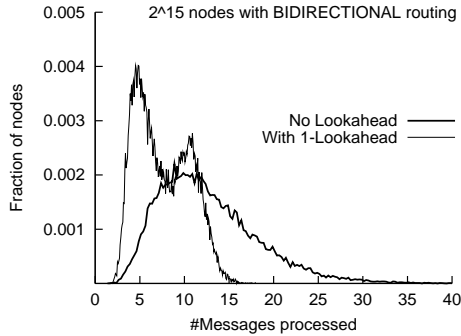


Figure 9: *Bandwidth profile in a network with  $n = 2^{15}$  nodes with  $k = 4$  links per node. Each node looks up one random hash key.*

when 1-Lookahead is employed. We are investigating the reason for this behavior.

#### 4.8 Resilience to Link Failures

Figure 10 explores the fault tolerance of our network. The top graph plots the fraction of queries answerable when a random subset of links (short as well as long) is deleted from the network. The bottom graph studies the impact of removing just long links. The slow increase in average latency is explained by Figure 3 which demonstrated diminishing returns of additional links. Figure 10 clearly shows that deletion of short links is much more detrimental to performance than deletion of long links. This is because removal of short links makes some nodes isolated. Removal of long links only makes some routes longer.

Figure 10 suggests that for fault tolerance, we need to fortify only the short links that constitute the ring structure. In Section 5.2, we leverage this insight to augment Symphony for fault tolerance.

#### 4.9 Comparison with $k$ Random Links

Figure 11 compares average latency for Symphony with a network where nodes form outgoing links with other nodes uniformly at random. The figure clearly shows that the obvious idea of choosing  $k$  uniformly random long distance neighbors does not scale since the path length grows as  $O(\sqrt{n/k})$ .

### 5 Comparison and Analysis

Symphony is a simple protocol that scales well and offers low lookup latency with only a handful of TCP

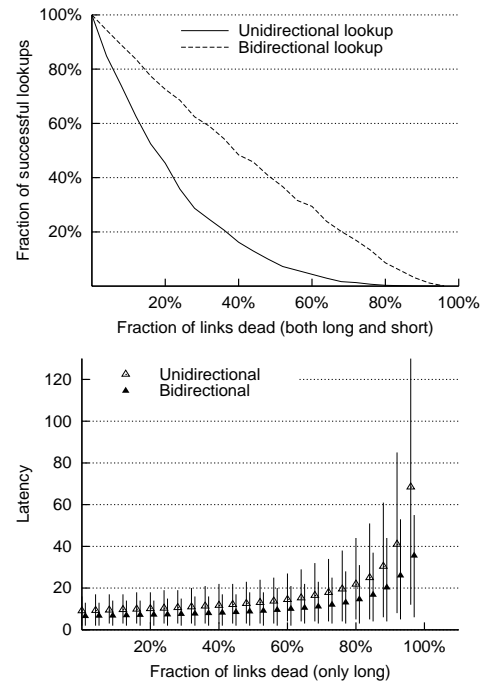


Figure 10: *Studying fault tolerance in a network of 16K nodes with  $\log \tilde{n}$  long distance links each. The top graph shows percentage of successful lookups when a fraction of links (short and long) are randomly deleted. The bottom graph shows increase in latency when only long links are randomly deleted.*

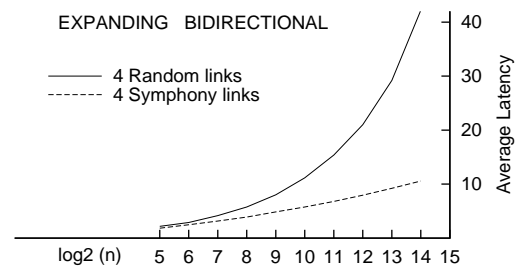


Figure 11: *Comparison with a network where each node links to  $k$  other nodes chosen uniformly at random. Network size  $n = 2^{15}$  nodes.*

connections per node. The cost of joining and leaving the network is small. Symphony is stable in the presence of frequent node arrivals and departures. We now highlight features unique to Symphony.

### 5.1 Low State Maintenance

Table 1 lists lookup latency vs. degree for various DHT protocols. Low degree networks are desirable for several reasons. First, fewer links in the network reduce the average number of open connections at servers and reduce ambient traffic corresponding to pings/keep-alives and control information. Second, arrivals and departures engender changes in DHT topology. Such changes are concomitant with the state update of a set of nodes whose size is typically proportional to the average degree of the network. Fewer links per node translates to smaller sets of nodes that hold locks and participate in some coordination protocol for distributed state update. Third, small out-degree translates to smaller bootstrapping time when a node joins and smaller recovery time when a node leaves without notice. Finally, it should be easier to isolate faults in low degree networks, making debugging faster. Symphony actually provides a smooth tradeoff between average latency and the amount of state per node.

### 5.2 Fault Tolerance

Symphony allows replication of content for fault tolerance by making  $f$  copies of a node's content at each of the  $f$  nodes succeeding it in the clockwise direction. A node maintains direct connections with all its  $f$  successors. This arrangement can tolerate  $f$  simultaneous failures before a portion of the hash table is lost. When a new key-value pair is inserted, a node propagates the changes to its  $f$  successors for consistency. Furthermore, all lookups succeed until some  $f$  successive nodes fail together. This is because the network remains connected (the base circle is intact) as long as at least one out of  $f$  successive nodes is alive for each node. Overall, the average number of TCP connections per node is  $2k + 2 + f$ . In practice, a small value of  $f$  less than ten should suffice, assuming independent failures and short recovery times.

Symphony's design for fault tolerance was motivated by Figure 10 (Section 4.8) where we showed that it is the short links that are crucial for maintaining connectivity. CFS [6] and PAST [19] use a variant of our fault tolerance scheme.

Symphony does not create any redundant long distance links for fault tolerance. There are no *backup*

links. It is only the short links that are fortified by maintaining connections with  $f$  successors per node. The long links contribute to the *efficiency* of the network; they are not critical for correctness (see Section 4.8). Protocols like Pastry, Tapestry and CAN maintain two to four backup links for *every* link a node has. A glance at Table 1 reveals that the overhead of redundant links for fault tolerance is significantly less for Symphony than other protocols. Having fewer links per node has several other benefits that we described in the preceding section.

### 5.3 Degree vs. Latency Tradeoff

Symphony provides a smooth tradeoff between the number of links per node and average lookup latency. It appears to be the only protocol that provides this tuning knob *even* at run-time. Symphony does not dictate that the number of links be identical for all nodes. Neither is the number stipulated to be a function of current network size nor is it fixed at the outset. We believe that these features of Symphony provides three benefits:

*Support for Heterogeneous Nodes:* Each node is merely required to have a bare minimum of two short-distance links. The number of long-distance links can be chosen for each individual node according to its available bandwidth, average lifetime, or processing capability. All the other DHT protocols specify the exact number and identity of neighbors for each node in the network. It is not clear how they would accommodate nodes with variable degrees. Symphony's randomized construction makes it adapt naturally to heterogeneous nodes ranging from home computers with dial-in connections to LAN-based office computers.

*Incremental Scalability:* Symphony scales gracefully with network size. The Estimation Protocol provides each participant with a reasonably accurate estimate of network size. It is possible for nodes to adapt the number of long distance links in response to changes in network size to guarantee small average lookup latency. This obviates the need to estimate in advance the maximum size of the network over its lifetime.

*Flexibility:* An application designer who uses a distributed hash table (DHT) would want to make its implementation more efficient by leveraging knowledge unique to the problem scenario. For example, the specifics of the network topology at hand, or the behavior of participating hosts, or a priori knowledge about the load on the DHT might be known. If the DHT itself has a rigid structure, the application designer is severely constrained. Sym-

# TCP Connections	Lookup Latency	Protocol	# TCP Connections	Lookup Latency	Notes
$2d$	$(d/2)n^{\frac{1}{d}}$	CAN	20	14.14	Fixed #dimensions
$2\log_2 n$	$(\log_2 n)/2$	Chord	30	7.50	Fixed #links
10	$\log_2 n$	Viceroy	10	15.00	Fixed #links
$(2^b - 1)(\log_2 n)/b$	$(\log_2 n)/b$	Tapestry	56	3.75	with b=4 digits
$(2^b - 1)(\log_2 n)/b$	$(\log_2 n)/b$	Pastry	22	7.50	with b=2 digits
			56	3.75	with b=4 digits
$2k + 2$	$c(\log^2 n)/k$	Symphony	10	7.56	k=4, bidirectional with 1-lookahead
			56	3.75	k=27, bidirectional with 1-lookahead

Table 1: Comparison of various protocols for a network of size  $2^{15}$ . Latencies are measured in terms of hops along the respective network topologies.

phony allows the number of links to be variable. All outgoing links are *identical* in the sense that they are drawn from the same probability distribution function. We believe that the randomized nature of Symphony poses few constraints as compared with other protocols.

## 5.4 Comparison with Other Protocols

We compare Symphony with other DHT protocols for a network of size  $n = 2^{15}$  nodes. We also discuss how other protocols could possibly use 1-lookahead and deploy additional links, if available.

(a) *CAN* can route among  $n$  nodes with an average latency of  $(d/2)n^{\frac{1}{d}}$ . The optimal value of  $d$  for  $n = 2^{15}$  nodes is 10 resulting in an average latency of 14.14. The average number of TCP connections is  $2d = 20$ . Dimensionality in CAN is fixed at the outset. It is not clear how dimensionality can be dynamically changed as the network expands or shrinks. Thus, CAN nodes would have 20 TCP connections each even if the network size is small. Unlike other protocols, CAN runs a *zone rebuilding protocol* in the background to adjust hash table partitions. CAN has low cost of joining. Heuristics for constructing a CAN topology that is aware of real network proximity have been shown to yield low IP latency on synthetic networks [16].

(b) *Chord* stipulates that every node in a network with  $2^{15}$  must have  $\log_2 n = 15$  outgoing links each, with the result that average latency is 7.5. In terms of TCP connections, nodes have  $2\log_2 n = 30$  connections each. Among existing DHT protocols, Symphony is closest in spirit to Chord. Chord could borrow ideas from Symphony for better performance. For example, Chord currently uses clockwise routing using unidirectional links. It can be modified to employ Symphony-style greedy routing over bidirectional links that minimizes absolute distance to the

target at each hop. Chord uses a rather expensive re-linking and stabilization protocol upon every insertion and deletion. When a node joins, up to  $O(\log n)$  other nodes who were pointing to this node's successor might have to re-establish their link with the new node. Experience with Symphony shows that re-linking is not worthwhile and that greedy routing continues to work satisfactorily even when nodes do not re-link.

(c) *Pastry*, with a digit size of 2 bits, would need an average of 22 TCP connections per node for average latency 7.5. Pastry can improve the latency to 3.75, but only with as many as 56 TCP connections per node. The digit size is a parameter that is fixed at the outset. For fault tolerance, Pastry maintains backup links for every link in its routing table. Moreover, content is replicated among  $L$  adjacent nodes. Pastry exploits network locality while choosing id's of nodes. The average latency over synthetic networks has been shown to be small [18].

(d) *Tapestry* uses 4-bit digits resulting in average lookup latency of 3.75 with 56 links per node. Tapestry is very similar to Pastry. The digit size is a parameter that is fixed at the outset. For fault tolerance, Pastry maintains backup links for every link in its routing table.

(e) *Viceroy* maintains seven links per node, irrespective of  $n$ , the size of the network. Each node has two neighbors along two rings, one up-link and two down-links. Four of these links are bidirectional, three are unidirectional. Thus, a Viceroy node would actually have an average of  $t = 10$  TCP connections per node. For  $n = 2^{15}$ , the average latency in Viceroy would be at least  $\log(n) = 15$ . This corresponds to an average 7.5 levels to reach *up* to the highest ring and another 7.5 levels to come *down* to the ring at the right level. Viceroy appears to be more complex and it is not clear how it would exploit network proximity while maintaining multiple concentric rings.

(f) *Symphony* offers a wide variety of choices for the number of TCP connections for a fixed value of  $n = 2^{15}$  nodes. Figure 7 shows that the average latency with  $k = 4$  long links with 1-Lookahead and bidirectional routing is 7.6. Such a topology results in 10 TCP connections per node on average. As  $k$  increases, *Symphony*'s average latency reduces. *Symphony* does not use backup links for long distance links. Instead each node replicates content on  $f$  successor nodes and maintains direct connections with them. This ensures content availability and successful lookups as long as no string of  $f$  successive nodes fails.

Lookahead seems to be of little value to CAN, Pastry and Tapestry. This is because the route of a lookup (and therefore, its length) is fixed. The protocol would not choose a different route if 1-Lookahead lists were available. The reason why lookahead is useful in *Symphony* is that a message could follow several paths from a source to a destination. We suspect that Chord might benefit by employing 1-Lookahead.

Let us see how additional links could possibly be used by various DHT protocols to improve performance. Chord could use a larger finger table. To change average degree at run-time in response to changes in network sizes, Chord could deploy *Symphony*'s Estimation protocol for estimating  $n$ . A CAN node maintains connections with its immediate neighbors in a  $d$ -dimensional torus. A natural way for CAN to use more links would be to increase its dimensionality. However, this is a fixed parameter and it is not clear how dimensionality could be changed at run time cleanly. CAN could presumably use additional links to connect to farther nodes along each dimension, giving it a flavor of Chord/*Symphony* per dimension. Viceroy uses seven links no matter how large  $n$  is. It remains unclear how Viceroy would employ more links if available; presumably it would entail a construction built on a  $d$ -way butterfly for  $d > 2$ . Pastry and Tapestry could use more links by increasing the digit-size which is a parameter fixed at the outset. Since the expected number of links is given by the formula  $(2^b \log n)/b$ , the possible values of average out-degree are limited and far apart. *Symphony* and a modified version of Chord appear to be the only protocols that offer a smooth trade-off between average latency and number of links per node.

## 6 Conclusions

We have presented *Symphony*, a simple protocol for managing a distributed hash table in a dynamic network of hosts with relatively short lifetimes. Through a series of systematic experiments, we have shown that *Symphony* scales well, has low lookup latency and maintenance cost with only a few neighbors per node. In particular,  $s = 3$  neighbors suffice for the Estimation Protocol and  $k = 4$  long distance links with Bidirectional routing and 1-Lookahead are sufficient for low latencies in networks as big as  $2^{15}$  nodes. We believe that *Symphony* is a viable alternative to existing proposals for distributed hashing.

We plan to adapt *Symphony* to an environment with heterogeneous nodes and gain experience with the implementation we are currently working on. An important next step in implementation is to take network proximity between nodes and heterogeneity into account.

## 7 Acknowledgments

We thank Geoff Voelker for his insightful comments that went a long way in improving the paper. We also thank the anonymous referees, Shankar Ponnekanti and Ramesh Chandra for their feedback. This work was partially supported by a grant from SNRC.

## References

- [1] L. Barriere, P. Fraigniaud, E. Kranakis, and D. Krizanc. Efficient routing in networks with long range contacts. In *Proc. 15th Intl. Symp. on Distributed Computing (DISC 01)*, pages 270–284, 2001.
- [2] M. Bawa, G. S. Manku, and P. Raghavan. SETS: Search Enhanced by Topic Segmentation. *Submitted for publication*, 2003.
- [3] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *ACM SIGMETRICS 2000*, pages 34–43, 2000.
- [4] I. Clarke, T. Hong, S. Miller, O. Sandberg, and B. Wiley. Protecting Free Expression Online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

- [5] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving dns using a peer-to-peer lookup service. In *Proc. 1st Intl. Workshop on Peer-to-Peer Systems (IPTPS 2002)*, pages 155–165, 2002.
- [6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, pages 202–215, 2001.
- [7] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proc. 4th Symposium on Operating System Design and Implementation (OSDI 2000)*, pages 319–332, 2000.
- [8] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proc. 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA 2002)*, pages 41–52, 2002.
- [9] S. Iyer, A. I. T. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC 2002)*, pages 213–222, 2002.
- [10] J. Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proc. 32nd ACM Symposium on Theory of Computing (STOC 2000)*, pages 163–170, 2000.
- [11] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. 9th Intl. conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 190–201, 2000.
- [12] W. Litwin, M. Neimat, and D. A. Schneider. Lh\*-a scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.
- [13] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proc 21st ACM Symposium on Principles of Distributed Computing (PODC 2002)*, pages 183–192, 2002.
- [14] S. Milgram. The small world problem. *Psychology Today*, 67(1), 1967.
- [15] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA 1997)*, pages 311–320, 1997.
- [16] S. Ratnasamy, P. Francis, M. Handley, and R. M. Karp. A Scalable Content-Addressable Network. In *Proc. ACM SIGCOMM 2001*, pages 161–172, 2001.
- [17] S. Ratnasamy, S. Handley, R. M. Karp, and S. Shenker. Application-level multicast using content addressable networks. In *Proc. 3rd Intl. Networked Group Communication Workshop (NGC 2001)*, pages 14–29, 2001.
- [18] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, pages 329–350, 2001.
- [19] A. I. T. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, pages 188–201, 2001.
- [20] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proc. 3rd Intl. Networked Group Communication Workshop (NGC 2001)*, pages 30–43, 2001.
- [21] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the Multimedia Computing and Networking (MMCN'02)*, 2002.
- [22] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM 2001*, pages 149–160, 2001.
- [23] S. Zhang, B. Zhao, B. Joseph, R. H. Katz, and J. Kubiatowicz. Bayeux: An architecture for wide-area, fault-tolerant data dissemination. In *Proc. 11th Intl. Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, pages 11–20, 2001.