Comparing the performance of distributed hash tables under churn

Jinyang Li, Jeremy Stribling, Thomer M. Gil, Robert Morris, M. Frans Kaashoek
MIT Computer Science and Artificial Intelligence Laboratory
{jinyang, strib, thomer, rtm, kaashoek}@csail.mit.edu

Abstract

A protocol for a distributed hash table (DHT) incurs communication costs to keep up with churn—changes in membership—in order to maintain its ability to route lookups efficiently. This paper formulates a unified framework for evaluating cost and performance. Communication costs are combined into a single cost measure (bytes), and performance benefits are reduced to a single latency measure. This approach correctly accounts for background maintenance traffic and timeouts during lookup due to stale routing data, and also correctly leaves open the possibility of different preferences in the tradeoff of lookup time versus communication cost. Using the unified framework, this paper analyzes the effects of DHT parameters on the performance of four protocols under churn.

1 Introduction

The design space of DHT protocols is large. While all designs are similar in that nodes forward lookups for keys through routing tables that point to other nodes, algorithms differ in the amount of state they keep: from $O(1)$ with respect to a network size of size n [7, 9] to $O(\log n)$ [10, 13, 14, 16] to $O(\sqrt{n})$ [6] to $O(n)$ [5]. They also differ in the techniques used to find low latency routes, in the way they find alternate paths after encountering dead intermediate nodes, in the expected number of hops per lookup, and in choice of parameters such as the frequency with which they check other nodes for liveness.

How is one to compare these protocols in a way that separates incidental details from more fundamental differences? Most evaluations and comparisons of DHTs have focused on lookup hopcount latency, or routing table size in unchanging networks [2, 12, 15]. Static analysis, however, may unfairly favor protocols that keep large amounts of state, since they pay no penalty to keep the state up to date, and more state usually results in lower lookup hopcounts and latencies.

This paper presents a framework for evaluating DHT algorithms in the face of joining and leaving nodes, in a way that makes it easy to compare tradeoffs between state maintenance costs and lookup performance. The paper compares the Tapestry [16], Chord [14], Kelips [6], and Kademlia [10] lookup algorithms within this framework. These four reflect a wide range of design choices for DHTs.

We have implemented a simple simulator that models inter-node latencies using the King method [3]. This model ignores effects due to congestion. We compare the performance of the DHTs using a single workload consisting of lookup operations and a particular model of churn. With these restrictions, we find that with the right parameter settings all four DHTs have similar overall performance. Furthermore, we isolate and analyze the effects of individual parameters on DHT performance, and conclude that common parameters such as base and stabilization interval can behave differently in DHTs that make different design decisions.

2 A Cost Versus Performance Framework

DHTs have multiple measures for both *cost* and *performance*. Cost has often been measured as the amount of per-node state. However, an analysis should also include the cost of keeping that state up to date (which avoids timeouts), and the cost of exploring the network to search for nearby neighbors (which allows low-latency lookup). A unified cost metric should indicate consumption of the most valuable system resource; in our framework it is the number of bytes of messages sent. This choice reflects a judgment that network capacity is a more limiting resource to DHTs than memory or CPU time.

Lookup performance has often been measured with hopcount, latency, success rate, and probability of timeouts. Our framework uses lookup latency as the unified performance metric relevant to applications. Lookup hopcount can be ignored except to the extent that it contributes to latency. The framework accounts for the cost of trying to contact a dead node during a lookup as a latency penalty equal to a small constant multiple of the round trip time to the dead node, an optimistic simulation of the cost of a timeout before the node pursues the lookup through an alternate route. DHTs retry alternate routes for lookups that return failed or incorrectly for up to four seconds, which effectively converts failed lookups into high latencies.

Protocol parameters tend to obscure differences in cost and performance among protocols, since the parameters may be tuned for different workloads. A key challenge is to understand differences solely due to parameter choices. We evaluate each protocol over a range of parameter values, outlining a performance envelope from which we can extrapolate an optimal cost-performance tradeoff curve.

This research was conducted as part of the IRIS project (http://project-iris.net/), supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660.

Parameter	Range
Base	2 – 128
Stabilization interval	36 sec – 19 min
Number of backup nodes	1 – 4
Number of nodes contacted during repair	1 – 20

Table 1: Tapestry parameters.

Parameter	Range
Number of successors	4 – 32
Finger base	2 – 128
Finger stabilization interval	40 sec – 19 min
Successor stabilization interval	4 sec – 19 min

Table 2: Chord parameters.

Parameter	Range
Gossip interval	18 sec – 19 min
Group ration	8, 16, 32
Contact ration	8, 16, 32
Contacts per group	2, 8, 16
Times a new item is gossiped	2, 8
Routing entry timeout	30 min

Table 3: Kelips parameters.

3 Protocol Overviews

This paper evaluates the performance of four existing DHT protocols (Tapestry [16], Chord [14], Kelips [6], and Kademlia [10]) using the above framework. This section provides brief overviews of each DHT, identifying the tunable parameters in each.

3.1 Tapestry

The ID space in Tapestry is structured as a tree. A Tapestry node ID can be viewed as a sequence of l base-b digits. A routing table has l levels, each with b entries. Nodes in the m^{th} level share a prefix of length $m-1$ digits, but differ in the m^{th} digit. Each entry may contain up to c nodes, sorted by latency. The closest of these nodes is the entry's *primary neighbor*; the others serve as *backup neighbors*.

Nodes forward a lookup message for a key by resolving successive digits in the key (*prefix-based routing*). When no more digits can be resolved, an algorithm known as *surrogate routing* determines exactly which node is responsible for the key [16]. Routing in Tapestry is recursive.

For lookups to be correct, at least one neighbor in each routing table entry must be alive. Tapestry periodically checks the liveness of each primary neighbor, and if the node is found to be dead, the next closest backup in that entry (if one exists) becomes the primary. When a node declares a primary neighbor dead, it contacts some number of other neighbors asking for a replacement; the number of neighbors used in this way is configurable. Table 1 lists Tapestry's parameters for the simulations.

3.2 Chord

Chord identifiers are structured in an identifier circle. A key k is assigned to k's successor (i.e., the first node whose ID is equal to k, or follows k in the ID space). In this paper's variant of Chord, a lookup for a key visits the key's *predecessor*, the node whose ID most closely precedes the key. The predecessor tells the query originator the identity of the key's successor node, but the lookup does not visit the successor. The base b of the ID space is a parameter: a node with ID x keeps $(b-1)\log_b(n)$ fingers whose IDs lie at exponentially increasing fractions of the ID space away from itself. Any node whose ID lies within the range $x+(\frac{b-1}{b})^{i+1} * 2^{64}$ and $x+(\frac{b-1}{b})^{i} * 2^{64}$, modulo 2^{64}, can be used as the i^{th} finger of x. Chord leverages this flexibility to obtain Proximity Neighbor Selection [2, 13]. Each node also keeps a *successor list* of s nodes. Chord can route either iteratively or recursively [14]; this paper presents results for the latter.

A Chord node x periodically pings all its fingers to check their liveness. If a finger i does not respond, x issues a lookup request for the key $x+(\frac{b-1}{b})^{i} * 2^{64}$, yielding node f. Node x retrieves f's successor list, and uses the successor with the lowest latency as the level i finger. A node separately stabilizes its successor list by periodically retrieving and merging its successor's successor list; successor stabilization is separate because it is critical for correctness but is much cheaper than finger stabilization. Table 2 lists the Chord parameters that are varied in the simulations.

3.3 Kelips

Kelips divides the identifier space into k groups, where k is a constant roughly equal to the square root of the number of nodes. A node's group is its ID mod k. Each node's routing table contains an entry for each other node in its group, and "contact" entries for a few nodes from each of the other groups. Thus a node's routing table size is a small constant times $\sqrt{n}$, in a network with n nodes.

The variant of Kelips in this paper defines lookups only for node IDs. The originating node executes a lookup for a key by asking a contact in the key's group for the IP address of the target key's node, and then (iteratively) contacting that node. If that fails, the originator tries routing the lookup through other contacts for that group, and then through randomly chosen routing table entries.

Nodes periodically gossip to discover new members of the network, and may also learn about other nodes due to lookup communication. Routing table entries that have not been refreshed for a certain period of time expire. Nodes learn RTTs and liveness information from each RPC, and preferentially route lookups through low RTT contacts.

Table 3 lists the parameters we use for Kelips. Rations are the number of nodes mentioned in the gossip messages. Con-

Parameter	Range
Nodes per entry (k)	4, 8, 16, 32
Parallel lookups (α)	1 – 10
Stabilization interval	20 min – 1 hour

Table 4: Kademlia parameters.

tacts per group is the maximum number of contact entries per group in a node's routing table; if it has value c, then the size of each node's routing table is $\sqrt{n} + c(\sqrt{n} - 1)$.

3.4 Kademlia

Kademlia structures its ID space as a tree. The distance between two keys in ID space is their exclusive or, interpreted as an integer. The k nodes whose IDs are closest to a key y store a replica of y. The routing table of a node x has 64 buckets b_i ($0 \leq i < 64$) that each store up to k node IDs with a distance to x between 2^i and 2^{i+1}.

Kademlia performs iterative lookups: a node x starts a lookup for key y by sending parallel lookup RPCs to the α nodes in x's routing table whose IDs are closest to y. A node replies to a lookup RPC by sending back a list of the k nodes it believes are closest to y in ID space. Each time node x receives a reply, it sends a new RPC to the next-closest node to y that it knows about, trying at all times to keep α outstanding RPCs. This continues until some node replies with key y, or until k nodes whose IDs are closest to y (according to x) did not return any new node ID closer to y. The simulated workloads look up node IDs, and the last step in a lookup is an RPC to the target node. Our Kademlia implementation favors proximate nodes. Like Kelips, Kademlia learns existence and liveness information from each lookup. Table 4 summarizes the parameters varied in the Kademlia simulations.

4 Evaluation

We implemented these four DHTs in a discrete-event packet-level simulator, p2psim.[1] The simulated network consists of $1,024$ nodes with inter-node latencies derived from measuring the pairwise latencies of $1,024$ DNS servers using the King method [3]. The average round-trip delay is 152 milliseconds, which serves as a lower bound for the average DHT lookup time for random keys. The simulator does not simulate link transmission rate or queuing delay. All experiments involve only key lookup, as opposed to data retrieval.

Nodes issue lookups for random keys at intervals exponentially distributed with a mean of ten minutes, and nodes crash and rejoin at exponentially distributed intervals with a mean of one hour. This choice of mean session time is consistent with past studies [4], while the lookup rate guarantees that nodes perform several lookups per session. Each experiment runs for six hours of simulated time, and nodes keep their IP address and ID for the duration of the experiment.

[1] http://pdos.lcs.mit.edu/p2psim

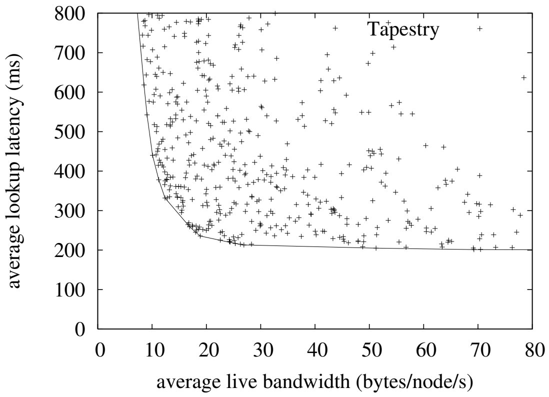

Figure 1: Cost versus performance under churn in Tapestry. Each point represents the average lookup latency and communication cost achieved for a unique set of parameter values. The convex hull represents the best achievable cost-performance combinations.

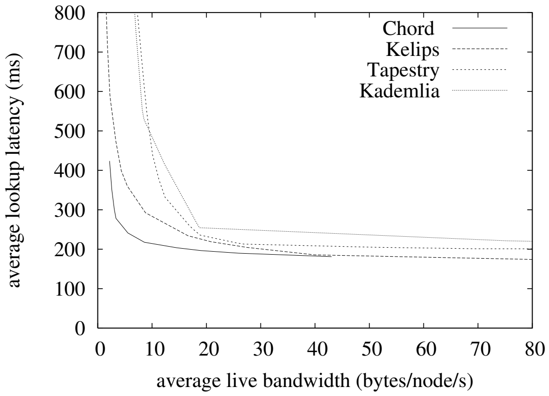

Figure 2: The convex hulls for all four protocols.

For each of the graphs below, the x-axis shows the communication cost: the average number of bytes sent per second sent by live nodes. The communication cost includes lookup, join, and routing table maintenance traffic. The size in bytes of each message is counted as 20 bytes for headers plus 4 bytes for each node mentioned in the message. The y-axis shows performance: the average lookup latency, including timeout penalties (three times the round trip time) and lookup retries (up to a maximum of four seconds).

4.1 Protocol Comparisons

Each protocol has a number of parameters that affect cost and performance. As an example, Figure 1 shows Tapestry's cost and performance for several hundred parameter combinations. There is no single best combination of parameter values. Instead, there is a set of best achievable cost-performance combinations: for each given cost, there is a least achievable latency, and for each latency, there is a least achievable cost. These best points are on the *convex hull* of the full set of points, a segment of which is shown by the line in Figure 1. Points not on the convex hull represent inefficient parameter settings which waste bandwidth.

Figure 2 compares the convex hulls of Tapestry, Chord, Kademlia and Kelips. Any of the protocols can be tuned to achieve a latency less than 250 ms if it is allowed enough bandwidth. The small difference in latency (20 ms) between Chord and Tapestry when bandwidth is plentiful is because Tapestry achieves a success rate of only 99.5%, compared to 100% in Chord, due to a slower join process that causes inconsistent views on which node is responsible for a given key. Kademlia uses iterative routing, which we observe to be slower than recursive routing when hopcounts are similar. When bandwidth is limited, the protocols differ significantly in performance. Chord in particular uses its bandwidth quite efficiently and can achieve low lookup latencies at little cost. This behavior appears to be due to Chord giving priority to stabilizing successors over fingers when bandwidth is limited, since correct successors are all that is needed to ensure correct lookups. By focusing its limited stabilization traffic on this small, constant amount of state (as opposed to its full $O(\log n)$ state), Chord is able to maintain correctness. The other protocols do not have a simple way to ensure correct lookups, and so their lookup times are increased by the need to retry lookups that return incorrect responses.

4.2 Parameter Exploration

Figure 1 shows that some parameter settings are much more efficient than others. This result raises the question of which parameter settings cause performance to be on the convex hull; more specifically,

- What is the relative importance of different parameters on the performance tradeoff for a single protocol?
- Do similar parameters have similar effects on the performance tradeoffs of different protocols?

These questions are not straightforward to answer, since different parameters can interact with one another, as we will see below. To isolate the effect of a single parameter, we calculate the convex hull segment for each fixed value of that parameter while varying all the other parameter values. The convex hull of these segments should trace the full convex hull as shown in Figure 2.

Tapestry: Figure 3 shows the effect of identifier base on the performance of Tapestry. Each line on the figure represents the convex hull segment for a specific value of base. With respect to bandwidth, these results are not surprising; as we decrease base, each node has fewer entries in its routing table,[2] and thus needs to contact fewer nodes during stabilization, using less bandwidth. For bases 2 and 4 nodes keep exactly the same amount of state, but base 4 lowers the hopcount leading to slightly improved latencies.

[2] If identifiers have base b in a network with n nodes, routing tables contain $b * \log_b n$ entries on average [16].

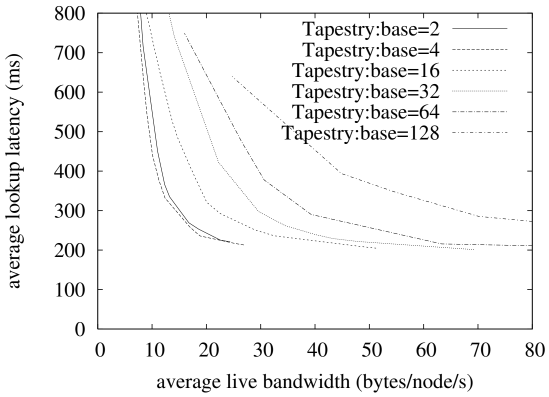

Figure 3: The effect of base in Tapestry.

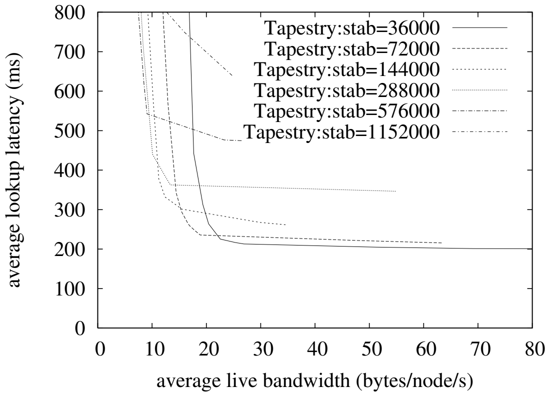

Figure 4: The effect of stabilization interval in Tapestry (values are in milliseconds).

The latency results, however, are a bit counter-intuitive: every value of base is able to achieve the same lookup performance, even though a smaller base results in more hops per lookup on average. This behavior is due to Tapestry's proximity routing. The first few hops in every lookup tend to be to nearby neighbors, and so the time for the lookup becomes dominated by the last hop, which is essentially to a random node in the network. Therefore, in a protocol with proximity routing, the base can be configured as a small value in order to save bandwidth costs due to stabilization.

Figure 4 illustrates the effect of stabilization interval on the performance of Tapestry. As nodes stabilize more often, they achieve lower latencies by avoiding more timeouts on the critical path of a lookup. Although this improvement comes at the cost of bandwidth, the results show that the cost in bandwidth is marginal when compared to the savings in lookup latency. Thus, not only can the base be set low, but stabilization also can happen frequently to keep routing state up to date under churn. For this workload, a reasonable value is 72 seconds.

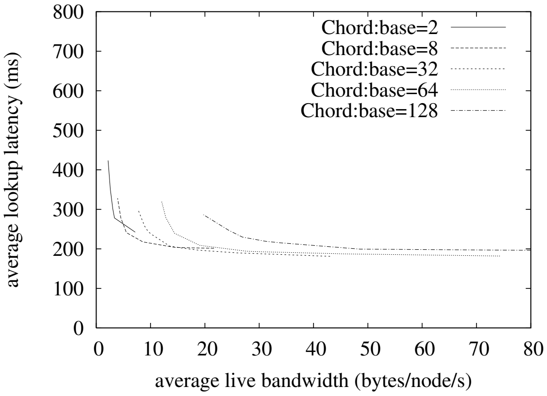

Figure 5: The effect of base in Chord.

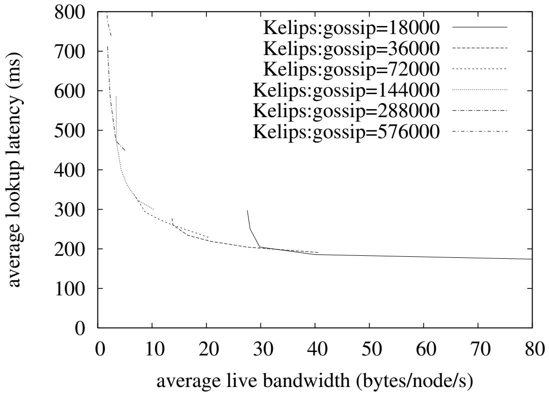

Figure 6: The effect of gossip interval in Kelips (values are in milliseconds).

Other experiments (not shown here) indicate that best performance is largely insensitive to number of backup nodes (as long as there are more than 1) and numbers of nodes contacted during repair.

Chord: Chord separately stabilizes finger and successors. While the relevant graphs are not shown for reasons of space, a 72 second successor stabilization interval is enough to ensure a high success rate (above 99%); faster rates result in wasted bandwidth while slower rates result in a greater number of timeouts during lookups. The finger stabilization interval affects performance without affecting success rate, so its value must be varied to achieve the best tradeoff. Faster finger stabilization results in lower lookup latency due to fewer timeouts, but at a higher communication cost.

Unlike Tapestry, there is no single best base value for Chord. Figure 5 shows the convex hulls for different base values. The final convex hull is essentially made up of two base values (2 and 8). Changing the base from 2 to 8 causes the best achieved lookup latency to drop from 240 milliseconds to 203 milliseconds due to decreased hopcount from 3.3 to 2.5. In comparison, small bases in Tapestry (see Figure 3) can achieve the same low latencies as higher bases; we believe this is due to a more involved join algorithm that samples a larger number of candidate neighbors during PNS.

Kelips: The most important parameter in Kelips in the gossip interval. Figure 6 shows that its value has a strong effect on the cost versus performance tradeoff. The other parameters improve performance without increasing cost, and thus are simple to set. For example, more contacts per group are always preferable, since that results in a more robust routing table and a higher probability that a lookup will complete in just one hop, at only slightly higher cost. With 16 contacts, for instance, the best lookup latency is 180 ms in 1.2 average hops as opposed to 280 ms in 1.9 hops for 2 contacts.

Kademlia: Figure 7 shows the effect on Kademlia of varying the number of parallel lookups (α). The final convex hull is made up of higher values of α, with bigger α resulting in lower latency at the cost of more lookup traffic. A bigger α decreases the time spent waiting for timeouts and increases the chances of routing lookups through proximate nodes.

Figure 8 shows that the Kademlia stabilization interval has little effect on latency, but does increase communication cost. Stabilization does decrease the number of routing table entries pointing to dead nodes, and thus decreases the number of timeouts during lookups. However, parallel lookups already ensure that these timeouts are not on the critical path for lookups, so their elimination does not decrease lookup latency.

4.3 Discussion

Base and stabilization interval have the most effect on DHT performance under churn, although they affect different protocols in different ways. These results are tied to our choice of workload: the effect of base depends on the size of the network, while the effect of stabilization depends on the average session time of churning nodes.

5 Related Work

This paper's contribution is a unified framework for comparing DHTs under churn and the effects of their parameters on performance under churn. Liben-Nowell et al. [8] focus only on the asymptotic communication cost due to Chord stabilization traffic. Rhea et al. [11] present Bamboo, a DHT protocol designed to handle networks with high churn efficiently and gracefully. In a similar vein, Castro et al. [1] describe how they optimize their Pastry implementation, MSPastry, to handle consistent routing under churn with low overhead. The implementation of the protocols described here include similar optimizations.

6 Conclusions and Future Work

This paper presents a unified framework for studying the cost versus lookup latency tradeoffs in different DHT pro-

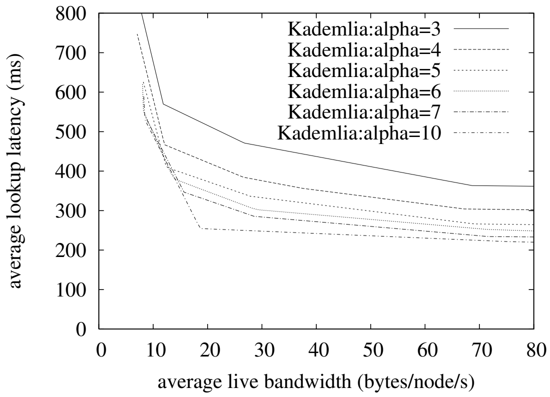

Figure 7: The effect of parallel lookups in Kademlia. Values of 1 and 2 are not shown, and perform considerably worse than $\alpha = 3$.

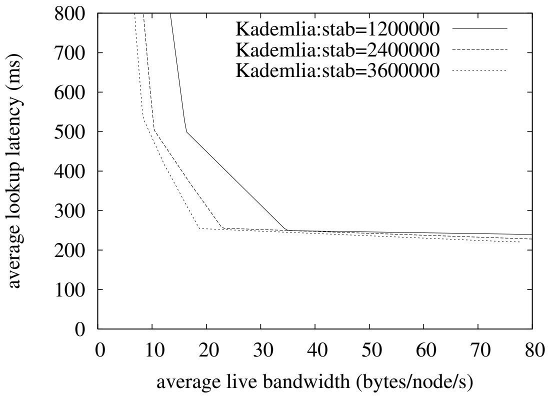

Figure 8: The effect of stabilization interval in Kademlia (values are in milliseconds).

tocols, and evaluates Tapestry, Chord, Kelips, and Kademlia in that framework. Given the workload described in Section 4, these protocols can achieve similar performance if parameters are sufficiently well-tuned. However, parameter tuning is a delicate business; not only can different parameters interact within a protocol to affect the cost versus performance tradeoff, but similar parameters in different protocols, such as base and stabilization interval, can behave differently. We also identify several parameters that are irrelevant under churn.

As future work, we plan to isolate and evaluate the design decisions that cause performance deviations between the protocols. We will also explore how varying the workload affects the cost versus performance tradeoff. We hope that understanding the tradeoffs inherent in different design decisions, as well as in parameter tuning, will lead to more robust and efficient DHT designs.

Acknowledgments

We thank Russ Cox for his help writing the simulator, Frank Dabek for numerous useful discussions, and the anonymous reviewers for their helpful comments.

References

[1] Miguel Castro, Manuel Costa, and Antony Rowstron. Performance and dependability of structured peer-to-peer overlays. Technical Report MSR-TR-2003-94, Microsoft Research, December 2003.

[2] Krishna P. Gummadi, Ramakrishna Gummadi, Steven Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the 2003 ACM SIGCOMM*, Karlsruhe, Germany, August 2003.

[3] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating latency between arbitrary Internet end hosts. In *Proceedings of the 2002 SIGCOMM Internet Measurement Workshop*, Marseille, France, November 2002.

[4] P. Krishna Gummadi, Stefan Saroiu, and Steven Gribble. A measurement study of Napster and Gnutella as examples of peer-to-peer file sharing systems. *Multimedia Systems Journal*, 9(2):170–184, August 2003.

[5] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. One hop lookups for peer-to-peer overlays. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems*, May 2003.

[6] Idnranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the Second IPTPS*, 2003.

[7] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal hash table. In *Proceedings of the Second IPTPS*, 2003.

[8] David Liben-Nowell, Hari Balakrishnan, and David R. Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the 2002 ACM Symposium on Principles of Distributed Computing*, August 2002.

[9] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable dynamic emulation of the butterfly. In *Proceedings of the 2002 ACM Symposium on Principles of Distributed Computing*, August 2002.

[10] Peter Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the First IPTPS*, March 2002.

[11] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. Technical Report UCB/CSD-3-1299, UC Berkeley, Computer Science Division, December 2003.

[12] Sean Rhea, Timothy Roscoe, and John Kubiatowicz. Structured peer-to-peer overlays need application-driven benchmarks. In *Proceedings of the Second IPTPS*, 2003.

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

[14] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, pages 149–160, 2002.

[15] Jun Xu. On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks. In *Proceedings of the IEEE Infocom*, March 2003.

[16] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.