Improving Lookup Performance over a Widely-Deployed DHT

Daniel Stutzbach, Reza Rejaie
Department of Computer & Information Science
University of Oregon
{agthorr,reza}@cs.uoregon.edu

Abstract— **During recent years, Distributed Hash Tables (DHTs) have been extensively studied through simulation and analysis. However, due to their limited deployment, it has not been possible to observe the behavior of a widely-deployed DHT in practice. Recently, the popular eMule file-sharing software incorporated a Kademlia-based DHT, called Kad, which currently has around one million simultaneous users.**

In this paper, we empirically study the performance of the key DHT operation, lookup, over Kad. First, we analytically derive the benefits of different ways to increase the richness of routing tables in Kademlia-based DHTs. Second, we empirically characterize two aspects of the accuracy of routing tables in Kad, namely completeness and freshness, and characterize their impact on Kad's lookup performance. Finally, we investigate how the efficiency and consistency of lookup in Kad can be improved by performing parallel lookup and maintaining multiple replicas, respectively. Our results pinpoint the best operating point for the degree of lookup parallelism and the degree of replication for Kad.

I. INTRODUCTION

Distributed Hash Tables (DHTs) present an elegant distributed solution for deterministically mapping items to locations. They provide a *structured* approach to Peer-to-Peer (P2P) applications since their item-to-location mapping can be used to *(i)* publish an item on a specific peer and *(ii)* efficiently lookup an item by locating its corresponding peer. During the past few years, the potential of DHTs has motivated a wealth of research including the design of new DHTs [1]–[5], performance evaluation and improvements [6], [7], and the development of a wide range of DHT-based distributed applications [8], [9]. Despite a great deal of attention from the research community, DHTs have not become widely-deployed until recently. In the absence of any large scale deployment, all previous studies on DHTs rely only on simulation, theoretical analysis, and limited-scale experiments. Therefore, the behavior of DHTs in practice has not been examined and thus is not well understood.

In practice, the dynamics of peer participation, or *churn*, can affect the accuracy of routing tables at each peer, and thus the performance of lookup operations in a DHT. More specifically, some entries in the routing table of individual peers might be missing or stale. Therefore, each peer does not have the expected connectivity to other peers. The inaccuracy of routing tables in turn affects the efficiency and consistency of lookup operations conducted by individual clients. For example,

a lookup may take more than the ideal number of hops or map to the wrong peer.

There are two classes of solutions to cope with the effect of churn on DHTs: *(i) DHT-based*: DHTs can incorporate various techniques to actively improve their resiliency to churn by increasing the degree of redundancy or the frequency of updates for the routing table at each peer. *(ii) Client-based*: Alternatively, a client operating over an inaccurate DHT can improve its lookup efficiency by conducting lookup in parallel and cope with lookup inconsistencies by active replication of content.

Previous studies have examined both DHT-based [10], [11] and client-based [4], [5] solutions as well as the interactions and trade-offs between them [7]. All of the previous studies have used either simulation, analysis, or small-scale experiments to study these issues. However the dynamics of user participation and their impact on routing table accuracy are not well understood. Given the limited understanding of churn characteristics, it is unclear how well simulation-based analysis of DHTs represents real-world behavior. Section VII discusses the related work in more detail.

This paper presents a measurement-based characterization of routing table inaccuracy and its impact on lookup performance in a widely-deployed DHT, namely *Kad*. Kad is an open, Kademlia-based [4] DHT with more than 1 million concurrent users that has been recently deployed by the popular eMule[1] file-sharing application to improve efficiency of search in the face of a growing user population. Section II presents an overview of Kademlia and Kad.

To study the inaccuracy of routing tables in a DHT, in Section III, we first establish an analytical framework to quantify the effect of routing table richness on the performance of lookup. To our knowledge, this is the first analysis to show that Kademlia's k-buckets improve lookup performance. In Section IV, we turn our attention to the accuracy of Kad's routing tables. Towards this end, we characterize both the freshness and completeness of routing tables in Kad through detailed and representative measurements using a tool we developed called *kFetch*. To explain the observed behavior, we carefully examined eMule's source code and present the

---

[1]eMule began as an open-source alternative for the eDonkey unstructured network.

underlying policies for routing table updates and redundancy management. Next, we turn our attention to different client-based techniques to improve lookup efficiency and consistency over Kad despite the inaccuracy of routing tables. Since we are dealing with a deployed DHT system, we are unable to explore DHT-based solutions.

In Section V, we examine two classes of parallel lookup techniques to improve lookup efficiency over Kad. Toward this end we developed a new tool called *kLookup*, which emulates a lookup from *any* source ID to *any* destination ID without requiring local access to the designated peers for these IDs. Furthermore, leveraging the iterative lookup scheme in Kad, kLookup enables us to empirically examine different parallel lookup techniques and identify major design trade-offs. Finally, in Section VI we characterize the frequency of inconsistent lookup results in Kad. We then explore how the degree of replication improves lookup consistency.

Our main contributions can be summarized as follows:

- **Analytical Framework**: We develop an analytical framework for computing the average performance of lookups for prefix-matching DHTs. This lead to the surprising result that redundancy in routing tables, such as Kademlia's k-buckets, directly improves mean lookup performance by reducing hop count
- **New Tools**: *(i)* kFetch, a tool for extracting the routing table from Kad peers, *(ii)* kLookup, a parameterized tool for performing lookups over Kad using a variety of lookup algorithms
- **Empirical Findings**: *(i)* Validating the predictions of our analytical framework, *(ii)* Locating the sweet spot for the degree of lookup parallelism to improve lookup efficiency, *(iii)* Locating the sweet spot for the degree of replication to overcome routing table inconsistencies

While this study is centered around Kad, our analysis, methodologies, tools and findings are mostly applicable to other DHTs with proper adjustment. To address the wider applicability of our work, we briefly discuss how some issues can be pursued in the context of other DHTs. Our extensive examination of eMule's source code also revealed several bugs, some of which were fixed in the next revision.

II. BACKGROUND

We first present some background on Kademlia, since it forms the basis for the Kad network that we use for our empirical study. Like most DHTs, peers in Kademlia each have an identifier that is assigned either uniformly at random or via a cryptographic hash. To determine the distance between two peers, Kademlia uses a unique "XOR metric", the bitwise XOR of their identifiers. For example, the distance between 0100 and 0111 is 0011 (or 3).

Kademlia belongs to the general class of prefix-matching DHTs, such as Pastry [3] and Tapestry [12]. At the high-level, these DHTs work in the same way. A *lookup* consists of a sequence of *lookup steps* (or hops). The first step consults the client's routing table for the target ID, which is guaranteed

to have a route where the high-order b bits match. The route points to another peer, which is consulted in the next step and is guaranteed to have a route where the first $2b$ bits match. The process continues until no next route can be found, indicating that the closest peer to the ID has been reached. We can view the distance between two identifiers as the number of bits that must be matched to reach from one to the other. For a network of n peers, most peers will be around $\log_2 n$ bits apart, and the expected number of steps to perform a lookup is $\frac{\log_2 n}{b}$. We call b the *symbol size*, and in basic Kademlia $b = 1$. Section III examines the impact of different choices for b on lookup latency (in hops) and route table size.

As IP is also a prefix-matching protocol, we borrow some terminology from IP to describe Kademlia routing tables. Each route in a Kademlia routing table is labeled with a *subnet address* and *mask*. When performing a lookup for a key, the most-specific routing table entry with a matching subnet is used, just as in IP routing. In this paper, the familiar "slash-notation" specifies the number of bits in the mask (*i.e.,* "/3" means an ID must match the highest-order 3 bits of the subnet address). In Kademlia, the routing table is structured to contain one route per address bit, with increasingly specific masks. The subnet addresses are the same as the ID of the peer hosting the routing table. The routing table structure can be viewed as a binary tree, as shown in Figure 1(a). For example, consider a Kademlia network using 4-bit identifiers[2] and a particular peer with the address 0000. There are route table entries for the following address–mask pairs: 0000/0, 0000/1, 0000/2, 0000/3, 0000/4. Because more-specific routes are preferred, the routing table entries are effectively for the following address–mask pairs: 1000/1, 0100/2, 0010/3, 0001/4, 0000/4. In other words, the 0000/0 line will only contain 1000/1 addresses since any 0000/1 address would map to one of the more specific entries.

The routing tables in all the Kademlia peers collectively form one large binary tree, with each peer containing a fraction $(O\left(\frac{\log n}{n}\right))$ of it. During a lookup, each routing step pivots to a different peer which is one bit closer to the target, guaranteeing that the lookup requires at most $O(\log n)$ steps.

For redundancy purposes, each routing table entry (or node in the binary tree) contains a list, called a k-bucket, of k matching contacts. Each contact includes the Kademlia ID, IP address, and port of the remote peer. Thus, each lookup step has a choice of k different contacts for the next step. Section III examines some of the consequences for choosing different values of k. We note that k-buckets could be adapted for use in other types of DHT as well.

Kademlia makes use of parallel routing to speed up lookups, as do EpiChord [13] and Accordion [5]. Issuing α lookup requests at a time avoids long waits while departed peers time out and also increase the probability of finding low-latency peers. Section V examines using different values of α in Kad.

Kademlia uses iterative routing, where the client is responsible for the entire lookup process. At each step, the client

---

[2]In practice no DHT would use such a small identifier space, but it's more tractable for illustrative purposes.

sends a *lookup request* to the next-hop peer and waits for a *lookup reply*. The reply lets the client know what the next hop is. Iterative routing contrasts with recursive routing, where the lookup request is forwarded automatically from one peer to another. While it has been shown that recursive routing typically has lower latency [14], iterative routing has several useful practical properties:

Fate-Sharing: Lookup messages cannot be lost due to the departure of an intermediate peer holding the lookup request [15].

Debugging: Iterative routing is easier to debug since information at each step is reported back to the client performing the lookup.

Compartmentalization: Iterative routing decouples route table maintenance and lookup technique, allowing them to be studied and improved independently in a deployed network. Our tool, kLookup, uses this division to evaluate a variety of lookup techniques directly over the existing Kad network, as shown in Sections V and VI.

Route Table Extraction: Iterative routing allows us to download the entire routing table of any peer. We make use of this feature in our tool, kFetch, described in Section IV-B.

In summary, the key properties of Kademlia (and thus Kad) are as follows: *(i)* routing by prefix-matching, *(ii)* redundancy in routing tables (k-buckets), *(iii)* parallel routing, and *(iv)* iterative routing. Redundancy, parallel routing, and iterative routing could be incorporated into most varieties of DHT. For example, EpiChord is a variant of Chord with parallel routing. Prefix-matching is an intrinsic property of Kademlia's design, which it shares with a number of other DHTs such as Pastry and Tapestry.

Kad is a Kademlia-based DHT network for file-sharing, composed of eMule clients. While Kad is based on Kademlia, Kad uses a slightly different routing table structure, described in detail in Section III. Kad has approximately 1 million simultaneous users, plus many more firewalled peers who utilize the Kad DHT for lookups but do not participate in the DHT structure. For each file an eMule client shares, the client computes the hash of each word in the filename, and publishes information about itself and the file to the peers responsible for the hashes. When an eMule user enters a keyword search, eMule computes the hash of the first keyword and initiates a lookup for the hash. The lookup returns a set of endpoints to which the client submits the full keyword list. Those peers process the query and return a set of matching results.

III. ANALYSIS OF KADEMLIA'S k-BUCKETS

In this section, we first establish an analytical framework to examine the effect on lookup performance of adding extra contacts to routing tables. We derive a formula for computing the typical number of hops needed to perform a lookup as a function of the way the extra contacts are structured, and use the formula to explore trade-offs between different methods for increasing the richness of routing tables.

Every DHT has some structure that determines a peer's potential neighbors based on identifiers. For example, in basic Kademlia a peer must have a neighbor with a different high-order ID bit, a neighbor with a matching first bit and a different second bit, a neighbor with the first two bits matching and a different third bit, etc. We call each address–mask pair a *bucket* (following the Kademlia terminology) where each bucket contains address information, called *contacts*, for several neighbors. A bucket with k contacts is called a k-bucket. In the base case, a DHT only contains enough information to perform the lookup in $\log_2 n$ steps. In prefix-matching DHTs such Kademlia, this implies a symbol size of $b = 1$ and one contact per bucket. In general, the expected number of steps required to perform a lookup is given as follows:

$$\text{steps per lookup} = \frac{\log_2 n}{\text{bits improved per step}} \quad (1)$$

A DHT can enrich the routing table structure beyond this base case by either 1) *adding more buckets* or 2) *adding more contacts per bucket*. By adding more buckets, a DHT can guarantee that a larger number of bits will be improved at each step, thereby decreasing the number of hops for a lookup. For example, Pastry [3] uses a default symbol size of $b = 4$ which guarantees 4 bits will be improved at each step. Tables in Chord can also be enriched in this way [7].

Adding more contacts per bucket is used to guard against churn, an approach employed by DHTs such as Kademlia [4] and Tapestry [12]. By having other contacts handy, a peer can more quickly repair its routing table when a failure is detected. Furthermore, as observed in [4], with heavy-tailed session times, storing backups and only evicting unresponsive peers implicitly leads to a set of peers with good uptime characteristics. Finally, multiple contacts per route allow for the use of parallel routing.

To examine the benefits and costs of the above two approaches for enriching routing tables, we analyze their impact in the context of Kademlia. Our analysis also directly applies to other prefix-matching systems such as Pastry and Tapestry, where we can quantify the improvement at each step in terms of the number of matching bits. For other DHTs that use a different basic geometry, our analysis could be adapted by modifying the formulas to reflect the appropriate distance metric.

There are two different approaches for adding more buckets to a routing table, both of which improve the number of lookup hops from $\log_2 n$ to $\log_{2^b} n$:

- **Discrete Symbols**: With this approach, illustrated in Figure 1(b), each interior node points to $2^b - 1$ buckets and an additional interior node. When searching a routing table, a peer begins by checking the first b bits. If all of them match the peer's ID, then it proceeds to the next b bits (*i.e.,* the next interior node). Otherwise, it proceeds immediately to the appropriate bucket. Using Discrete Symbols increases the routing table size from $\log_2 n$ rows of one k-bucket each to $\log_{2^b} n$ rows of $2^b - 1$ k-buckets each. This is the approach used in Kademlia and Pastry.
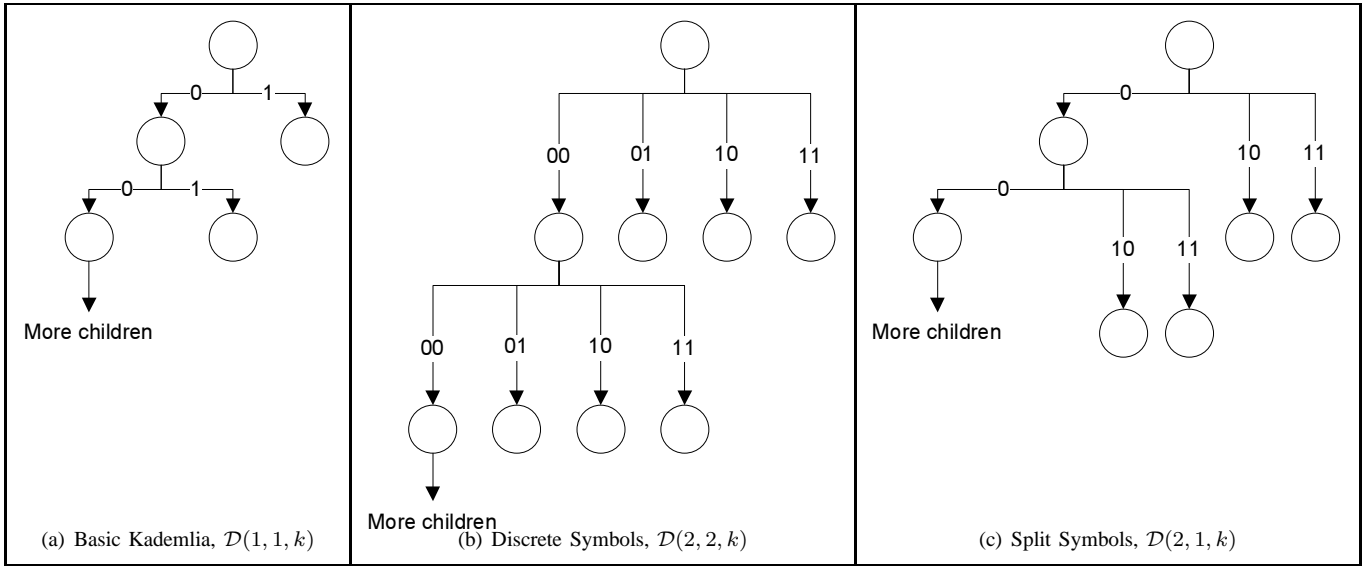
Fig. 1.   Routing Table Structures

(a) Basic Kademlia, $\mathcal{D}(1,1,k)$
(b) Discrete Symbols, $\mathcal{D}(2,2,k)$
(c) Split Symbols, $\mathcal{D}(2,1,k)$

- **Split Symbols**: With this approach, illustrated in Figure 1(c), each interior node points to 2^{b-1} buckets and an additional interior node. When searching a routing table, a peer begins by checking the first single bit. If it matches the peer's ID, then it proceeds to the next bit (*i.e.,* the next interior node). Otherwise, it examines the next b bits and proceeds to the appropriate bucket. Using Split Symbols increases the routing table size for $\log_2 n$ rows of one k-bucket each to $\log_2 n$ rows of 2^{b-1} k-buckets each. This is the approach used in Kad.

To compare and contrast these approaches for organizing routing table contacts, we create a general framework for analyzing their performance. We define $\mathcal{D}(b,r,k)$ as a system which uses b-bit symbols with r-bit resolution and k-buckets. $\mathcal{D}(1,1,k)$ is the basic Kademlia approach, $\mathcal{D}(b,b,k)$ is the Discrete Symbol approach, and $\mathcal{D}(b,1,k)$ is the Split Symbol approach used in Kad. Each routing table has $\log_{2^r} n$ rows of $2^b - 2^{b-r}$ k-buckets, for a total size of $k(2^b - 2^{b-r}) \log_{2^r} n$ contacts. Normalizing by a factor of $\log_2 n$ yields a normalized size of $k\frac{2^b-2^{b-r}}{r}$.

Most prior work on most DHTs[3] is concerned exclusively with the worst-case scenario where the selected contact will not match any *additional* bits of the target identifier. For example, consider searching for the key 111 in the routing table of peer 000 with the base $b = 1$ system. The peer looks in the bucket with the prefix 1, and returns a contact which we know matches the first bit of the key. However, that contact could be any of the peers 100, 101, 110, or 111. In other words, there's a $\frac{1}{2}$ chance of improving at least 1 extra bit, a $\frac{1}{4}$ chance of improving at least 2 extra bits, and so on. More precisely, the probability of improving at least δ bits is:

$$Pr[X \geq \delta] = \frac{1}{2^\delta} \qquad (2)$$

---

[3]To our knowledge, the only work on DHTs which has considered the average-case performance is Chord [1].

Therefore, the average-case is better than the worst-case given in prior work. In particular, the key insight is that large buckets ($k > 1$) improve the probability of *randomly* finding a contact with more matching bits since there are more options to choose from. As we will show, the average number of improved bits increases logarithmically with k, making the performance boost of increasing k comparable to the performance boost of increasing b. Generally, for a k-bucket the probability of improving by at least δ extra symbols is:

$$F(\delta,r,k) = Pr[X \geq \delta] = 1 - \left(1 - \frac{1}{2^{r\delta}}\right)^k \qquad (3)$$

and the probability of improving *exactly* δ symbols is:

$$f(\delta,r,k) = Pr[X = \delta] = F(\delta,r,k) - F(\delta+1,r,k) \qquad (4)$$

The key question is: *how many additional bits improve on average due to randomness?* Since we know the probability of improving exactly δ additional symbols ($f(\delta,r,k)$), we can compute average number of extra bits improved by finding the average value of δ and multiplying by the number of bits per symbol (r) as follows:

$$\text{extra bits improved per step: } m(r,k) = r \sum_{\delta=0}^{\infty} \delta \cdot f(\delta,r,k) \quad (5)$$

$$\text{total bits improved per step: } t(b,r,k) = b + m(r,k) \qquad (6)$$

Note that $m(r,k)$ is actually decreasing in r due to the r term in Formula 3. While we were unable to find a simple closed form for $m(r,k)$, it can be computed numerically without difficulty. For $r = 1$, $m(1,k)$ asymptotically approaches $\log_2 k + 0.3327$, somewhat exceeding this value for lower k. Significantly, for the base case $\mathcal{D}(1,1,1)$ of no additional routing table entries, $m(1,1) = 1$ indicating one extra bit improves per step. In other words, a basic $\mathcal{D}(1,1,1)$ system
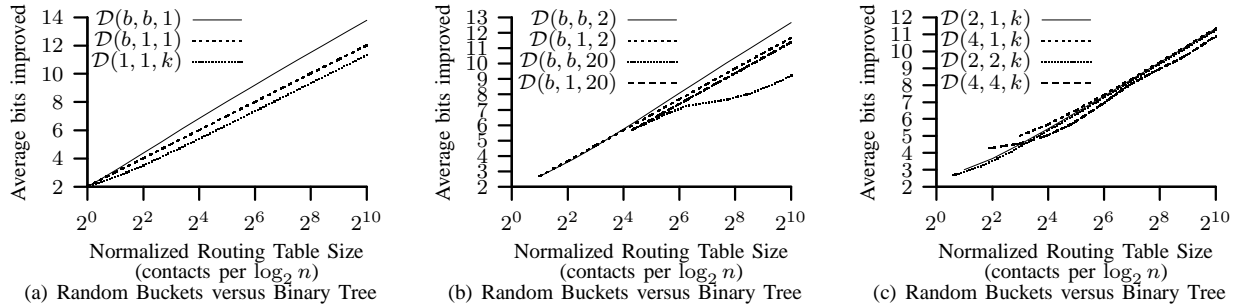
Fig. 2. Relative performance of different routing table structures

on average performs a lookup in *half as many hops* as reported by previous work.

For a Discrete Symbol configuration, $\mathcal{D}(b, b, k)$, the number of bits improved on average is $b + m(b, k)$. For a Split Symbol configuration, $\mathcal{D}(b, 1, k)$, the number of bits improved on average is $b + m(1, k)$. *While the Split Symbol approach does use more routing table space, it has the advantage that it can leverage a random improvement of a single extra bit.* The Discrete Symbol approach must randomly improve by b extra bits at a time to make use of random improvements.

To compare the different approaches, first consider the three extreme cases: $\mathcal{D}(1, 1, k)$ (pure Redundancy), $\mathcal{D}(b, b, 1)$ (pure Discrete Symbols), and $\mathcal{D}(b, 1, 1)$ (pure Split Symbols). Figure 2(a) presents the performance of each approach as a function of the normalized routing table size. Split Symbols and Redundancy have nearly identical performance, while Discrete Symbols performs slightly better. For the case of Split Symbols ($\mathcal{D}(b, 1, 1)$), the b-bit symbols guarantee an improvement of b bits in the worst case, plus an additional $m(1, 1) = 1$ bits on average, for a total of exactly $b + 1$ bits, dividing by the normalized size yields $\frac{b+1}{2^b-1}$. This is the slope of the Split Symbols ($\mathcal{D}(b, 1, 1)$) line in Figure 2(a).

For the case of Discrete Symbols ($\mathcal{D}(b, b, 1)$), the b-bit symbols again guarantee an improvement of b bits in the worst case, plus an additional $m(r, 1)$ bits on average. However, $m(r, 1)$ asymptotically approaches 0 for large r. As a point of reference, for Pastry's typical value of $b = r = 4$, the average improvement is 4.27 bits per step, roughly a 4% reduction in the mean number of lookup hops[4] compared to that reported by the Pastry authors [3]. The average improvement divided by the normalized size is $b\frac{b+m(r,1)}{2^b-1}$.

For the case of large buckets and 1-bit symbols ($\mathcal{D}(1, 1, k)$), the 1-bit symbols guarantee an improvement of 1 bit in the worst case, plus an additional $m(1, k)$ bits on average, for a total of $1 + m(1, k)$. Dividing by the normalized size results in $\frac{1+m(1,k)}{k}$. As a point of reference, for the value of $k = 20$ suggested in the Kademlia paper [4], the average improvement is 5.7 bits per step rather than 1 bit per step, resulting in a 60% reduction in the mean number of hops!

An important question is: *Can performance be improved by using a mixture of large buckets and large symbols?* The short answer is "No". Figures 2(b) and 2(c) plot several other permutations of $\mathcal{D}(b, r, k)$. Figure 2(b) holds k constant and varies b, while Figure 2(c) holds b constant and varies k. For small values of k (*e.g.,* 2) with varying b, both Discrete Symbols and Split Symbols have performance in between their regular performance and $\mathcal{D}(1, 1, k)$. For moderate values (*e.g.,* 20) of k, the performance of Split Symbols is virtually identical to $\mathcal{D}(1, 1, k)$, while the performance of Discrete Symbols plummets (as seen in Figure 2(b)). Because Discrete Symbols cannot make good use of randomness, the k-redundancy imposes a cost with little benefit on lookup performance.

In summary, increasing the symbol size offers a constant-factor improvement to worst-case performance, while using k-buckets offers comparable average-case improvement. Moreover, k-buckets offer other advantages as follows:

- Reduced implementation complexity
- Lower maintenance bandwidth; fewer restrictions on acceptable contacts allows for more contacts to be acquired passively
- Better resistance to churn by accumulating high-quality contacts

While our framework is motivated by our study of Kad, it applies to any prefix-matching DHT and could be extended to other DHTs that can accommodate different symbol or bucket sizes, such as Chord. In the following section, we use the formulas we have developed to compute a lower bound on the average lookup hops in Kad and empirically examine how close our predicted model is to the actual performance.

IV. ACCURACY OF ROUTING TABLES IN KAD

In this section, we empirically characterize the degree of routing table accuracy in Kad and identify the underlying reasons for inaccuracies. These characteristics help us explain the observed lookup performance in Section V. Our goal is to explore the structure and redundancy (*i.e.,* b and k) of routing tables in Kad by examining the eMule source code[5], and then empirically studying the impact of churn on routing table accuracy.

---

[4]The expected number of hops is equal to $\log_{\mathcal{B}} n$ where $\mathcal{B}$ is the average number of bits of improvement.

[5]There is no written specification that describes the Kad protocol so our explanations are based on our reading of the source code.

A. Predicting Kad Performance

Close examination of the eMule 0.46a source code reveals that Kad is based on Kademlia with a bucket size of 10 contacts ($k = 10$) and 3.25-bit Split Symbols, meaning Kad is a $\mathcal{D}(3.25, 1, 10)$ system. The $\frac{1}{4}$ bit is due to the fact that Kad uses unbalanced subtrees. Each interior node has branches with labels 0, 1000, 1001, 101, 110, and 111. The 0 branch leads to the next interior node; the other branches lead to k-buckets. The average improvement per step is $3.25 + m(1, k)$ bits. We also validated our understanding of the source code with empirical observations of its operation. Therefore, according to Formula 6 the mean number of improved bits per step is 6.98 in Kad. As a special case, Kad's root node has a full 16 branches, so it improves at least the 4 most significant bits and 7.73 bits on average on the first step. To account for this, we revise Formula 1 as follows:

$$\text{steps per lookup in Kad: } 1 + \frac{\log_2 n - t(4, 1, 10)}{t(3.25, 1, 10)} \quad (7)$$

Thus, the expected number of hops in Kad is $1 + \frac{\log_2(n) - 7.73}{6.98}$.
Estimating Network Size: Toward this end, we need an estimate of the size of the Kad network (n). An obvious approach would be to crawl the Kad network to capture the entire population of peers. However, crawling the *entire* Kad network takes too long due to the large size of routing tables at each peer and the large number of peers in the network. Because churn occurs while the crawler runs, a very long crawl would result in an inflated population count as it would record a large number of short-lived peers that are not simultaneously present. However, crawling a subnet is much faster than crawling the whole network. Since Kad identifiers are selected uniformly at random, any subset of the ID space (such as a subnet) is a representative sample of the total population. Multiplying the measured size of a subnet by the number of such subnets yields an estimate of the population size. By taking the mean over many such samples, we can get a good estimate for n.

In our previous work [16], we developed a parallel peer-to-peer overlay crawler, called *Cruiser*. Given a Kad overlay subnet as an input (*e.g.,* 0x5cd/12), Cruiser walks the DHT structure to capture a snapshot of all the active peers with IDs in the specified subnet. For example, it can capture a /10 subnet with roughly 1000 peers in around 3–4 minutes and a /12 subnet with roughly 250 peers in around one minute. During June of 2005, we captured the population size for several hundred randomly selected subnets with Cruiser. Our measurements reveal that the Kad network has a mean population size of approximately 980,000 concurrent peers. Given this estimated group size for the Kad network, a lookup over Kad requires $\log_2 980,000 \approx 19.9$ bits of improvement, and a lookup in Kad should take $\frac{19.9 - 7.73}{6.98} + 1 = 2.7$ hops (according to Formula 7), assuming perfect routing tables. This is significantly better than predicted by the formula of prior work [4] of $\frac{\log_2(n) - 4}{3} + 1 = 6.30$ hops. *Correctly incorporating the effect of randomness alters the predicted performance by more than a factor of two.*

B. kFetch

To study the accuracy of routing tables, we developed a new tool called *kFetch*. kFetch chooses a Kad peer at random, downloads its complete routing table, and identifies stale entries in the routing table by actively probing (*i.e.,* sending a lookup request) to each contact in the routing table. To locate a peer at random, kFetch generates a random Kad identifier, then performs a Kad lookup to locate the peer closest to that Kad identifier. The routing table of the target peer must be downloaded quickly in order to minimize any error due to ongoing churn (*i.e.,* a contact that was actually present in the network might depart before kFetch probes it). There are two challenges to download a routing table efficiently: *(i)* the rate of requests (which are UDP messages) must be properly paced to rapidly download the table without causing excessive network congestion, and *(ii)* lookup messages must request the right IDs to extract a peer's routing table with the minimum number of messages. kFetch implements congestion control using a variant of the SACK TCP algorithms to determine the proper rate for issuing requests. kFetch computes the routing table structure of the target peer according to Kad's rules for populating them and generates a query for each k-bucket the peer may have. This strategy could be used to extract the routing table in any DHT that uses iterative routing. In addition, it examines the returned data to determine when a branch of the tree is empty, and will not issue queries for the empty subtree. Additionally, for each discovered contact, kFetch queries the contact to verify whether it is still present in the network, concurrently with continuing to download the routing table.

C. Characteristics of Kad Tables

Using kFetch, we retrieved the routing tables of approximately 80,000 distinct Kad peers in June 2005 and examined two properties of their k-buckets: *(i) completeness* is the whether the bucket contains the appropriate number of entries, given the size of the Kad network; and *(ii) freshness* is the number of contacts in the routing table that are still active (*i.e.,* do not point to departed peers). Figure 3(a) shows the mean number of contacts ("Known") in each routing table bucket as a function of the bucket's subnet mask. It also shows what fraction of these contacts are fresh (*i.e.,* the contact responded to our ping). The "Ideal" line indicates the average number of contacts we would expect to be in each bucket if the routing tables were perfectly up to date, *i.e.,* $\min\left(10, \frac{n}{2^x}\right)$ where x is the number of bits in the address mask and n is the population size. All three curves (Ideal, Known, and Fresh) decrease off steeply as the mask length exceeds 16 bits, due to the limited number of matching contacts in the system. For shorter masks, on average each bucket has one or two empty slots and contains one stale contact. The mean number of empty slots is slightly higher as the mask length increases.

In Figures 3(b) and 3(c), we examine the number of fresh contacts in each bucket normalized by the total number in each bucket and by the expected (ideal) number, respectively. Figure 3(b) shows the mean number of fresh contacts as a
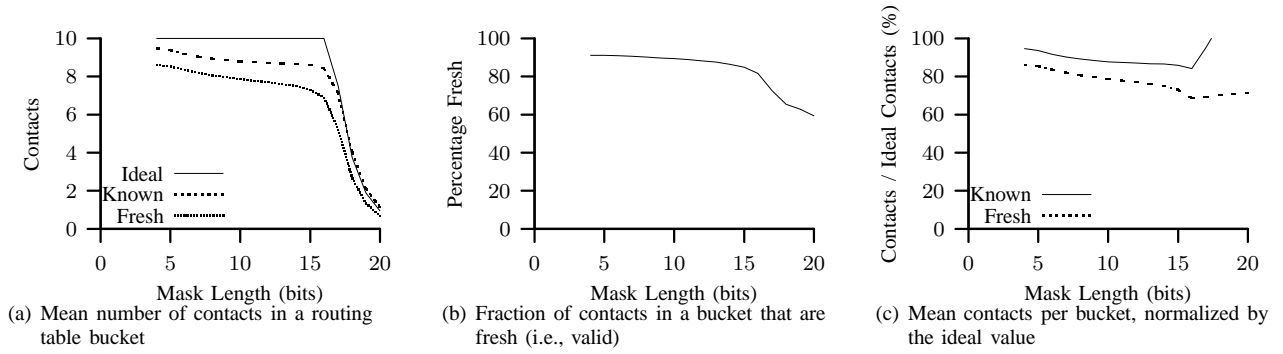
Fig. 3. Properties of routing table buckets in Kad, as a function of how specific the bucket's mask is
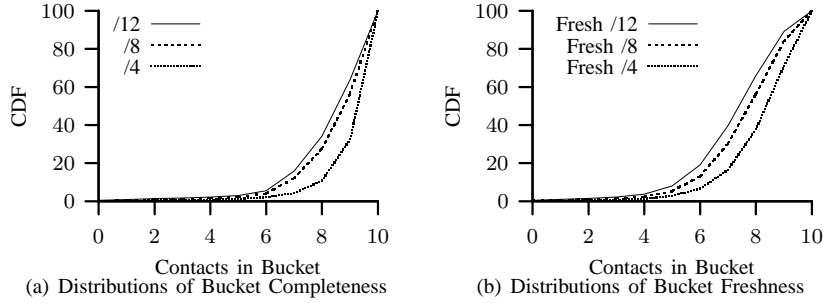

Fig. 4. Distributions of Completeness and Freshness in Kad's Buckets

fraction of the number of contacts actually present. This shows that around 90% of entries are fresh for masks up to length around 16, then the fraction of fresh entries decreases, *i.e.,* the number of stale entries increases. This is because the current implementation of eMule doesn't ping peers in buckets which are not at least 70% full. In fact, in Figure 3(c), where we examine the number of contacts relative to the ideal number, above /17 there are actually more stale contacts than active peers anywhere in that subnet, causing the normalized value to exceed 100%! Peers gradually accumulate stale contacts in these buckets which are expunged too slowly. As a consequence, virtually every lookup in Kad necessarily ends with timeouts to stale peers even though the closest peer has already been contacted! This is a direct result of eMule's policy of not expiring contacts in mostly-empty buckets. As this routing table maintenance problem can trivially be corrected in the eMule code, in the remainder of this paper we emulate the correct behavior as follows. After a lookup completes, we compute the latency as the time from the start of the lookup until kLookup receives a packet from the closest *responsive* peer.

From Figure 3(a), we see that on average there are 1.5 empty slots plus 1 stale contact per bucket. We could plug $k = 10 - 1.5 - 1 = 7.5$ into our formula, but first we must validate that most buckets are close to the average state. If the variance is very high (*e.g.,* if 85% of buckets had 10 entries and the other 15% were completely empty), then using the average would introduce considerable error. Towards this end, Figures 4(a) and 4(b) present the CDF of the number of contacts and fresh contacts across all observed buckets for masks /4, /8, and /12.

They show that for both completeness and freshness, nearly all buckets are close to the average value. Therefore, we may use the average value for the purposes of our computations without introducing considerable error.

Using an average of 1.5 empty slots plus 1 stale contact per bucket, we have an effective bucket size of $k = 10 - 1.5 - 1 = 7.5$. This increases the expected hop count slightly from 2.7 to $\frac{\log_2(n) - 7.33}{6.58} + 1 = 2.91$ hops, according to Formula 7. This is still significantly better than the previously predicted value of 6.30 hops.

Note that we are unable to change the routing tables in the entire Kad network. Therefore, we explore client-based alternatives to improve lookup performance in Kad and evaluate different techniques to improve the efficiency and consistency of lookup in the following two sections.

V. IMPROVING LOOKUP EFFICIENCY

We turn our attention to client-based approaches to improve the performance of iterative lookup over a DHT that has inaccurate routing tables. While incomplete buckets will degrade performance as described in the previous section, stale contacts can dramatically increase latency by causing timeouts to occur. Since the timeout interval is typically set to at least a few round-trip times, it can easily exceed the desired time for the entire lookup.

A. Parallel Lookup

To improve performance despite inaccurate routing tables, clients (*i.e.,* end-points) can perform parallel lookup. While

parallel lookup has traditionally been used exclusively with iterative DHTs, Jinyang Li *et al.* [5] present a technique for performing parallel lookup on a recursive DHT.

In a parallel lookup, a client simultaneously manages multiple lookup requests to different peers and performs the lookup process based on the information obtained from all requests, reducing the problem of hitting stale contacts, and improving lookup performance at the cost of greater network overhead (*i.e.,* a larger number of requests per lookup). Parallel lookup has two other significant advantages. First, lookup requests facilitate populating or passively updating the routing tables, which in turn reduces the bandwidth requirement for explicit updates, as shown in [7]. Second, during each step of the lookup process, parallelism increases the number of contacts searched, increasing the probability of finding a contact closer to the target (*i.e.,* with more matching bits) and thus decreasing the number of hops needed to reach the target. We examine the following two classes of parallel lookup techniques: *(i)* Strict Parallel lookup and *(ii)* Loose Parallel lookup.

1) **Strict Parallel Lookup**: In this approach, a client begins a lookup by sending lookup requests to the α best known contacts. Similar to the window-based congestion control in TCP, a client restricts the number of requests in-flight to α. A new request is issued only when a pending request times out or a response is received. The resulting overhead is limited to a factor of α. The downside of the strict approach is that when a client sends a packet to a departed contact, it must wait for a timeout to occur before giving up. In the meantime, the degree of parallelism is effectively reduced by one. However, a timeout is typically set to at least a few round-trip times which is on the order of the desired time for the entire lookup. Thus, in the strict approach, α roughly determines the number of timeout events a client can experience without incurring a significant latency penalty. Kademlia uses this approach.

2) **Loose Parallel Lookup**: Parallel lookup can be performed in a looser fashion by allowing more than α requests in flight. In this approach, a client can issue a lookup request to a contact that is among the top α contacts as soon as such a contact is identified, even if this lookup request increases the number of pending requests beyond α. For example, if $\alpha = 3$, the lookup begins by sending 3 lookup requests. If the first response contains 3 better contacts (which is likely), 3 more requests are sent immediately. While this approach appears to be significantly more expensive than strict parallel lookup, it incurs only modest additional overhead since later responses from the same step are less likely to contain better contacts (*i.e.,* each time a packet is sent, the bar has been raised). The advantage of this looser approach is the ability to quickly abandon lookups that are likely to time out. This approach is used by eMule.

kLookup: To examine different lookup strategies, we developed a new tool, called *kLookup*, which performs a lookup from *any* source ID to *any* destination ID without requiring local access to those peers. To emulate a lookup from a particular source ID, kLookup takes the following steps. First, it uses a local Kad routing table to locate the peer closest to the source ID (*i.e.,* the source peer), then it extracts the routing table of the source peer using kFetch. Finally, it performs a lookup to the destination ID using the routing table of the source peer. kLookup implements an adjustable degree of parallelism (α) with both strict and loose parallel lookup.

B. Evaluating Parallel Lookup

We evaluated the performance of both types of parallel lookup techniques under varying degrees of parallelism. Using kLookup, we captured several hundred lookups for different values of α for both strict and loose parallelism. Each lookup used a unique, randomly-selected source and a unique, randomly-selected destination. In our evaluation, we examine three metrics:

- **Hops**: The number of hops from the source to the destination
- **Latency**: The duration from the start of the lookup to when a response is received by the final destination, which is a function of the number of hops and the time spent waiting for responses and timeouts
- **Messages Sent**: The overhead used to perform the lookup

As we mentioned earlier, increasing α can reduce the number of lookup hops by providing more opportunities to randomly improve extra bits. Figure 5 shows that the mean number of hops decreases slightly as α increases[6], providing empirical support. Furthermore, the hop count for $\alpha = 1$ is around 3.2, which is close to our predicted lower-bound of 2.9.

Since the number of hops is as expected, the next question is: *how much latency is introduced to lookup by timing out due to stale contacts?* Figure 6(a) compares the latency of the two approaches for several values of α. The first observation is that the latency for $\alpha = 1$ is very high—close to 10 seconds. Using a value of $\alpha = 3$ dramatically reduces the latency, with diminishing returns for larger α. Second, Figure 6(a) reveals that the loose approach is just barely quicker than the strict approach for constant α. The greatest advantage of loose parallelism is that

[6]This figure is noisy due to the narrow y-axis range. The general downward trend is nevertheless visible.
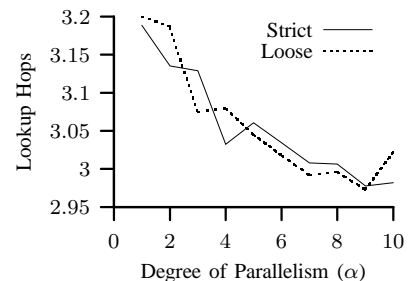

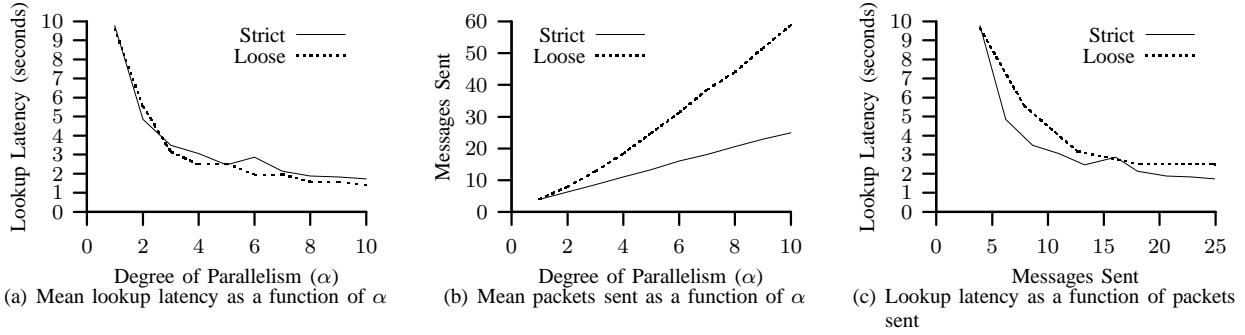Fig. 5.   Effect of parallel lookup on hop count

(a) Mean lookup latency as a function of α (b) Mean packets sent as a function of α (c) Lookup latency as a function of packets sent

Fig. 6.   Strict versus Loose Parallel Lookup
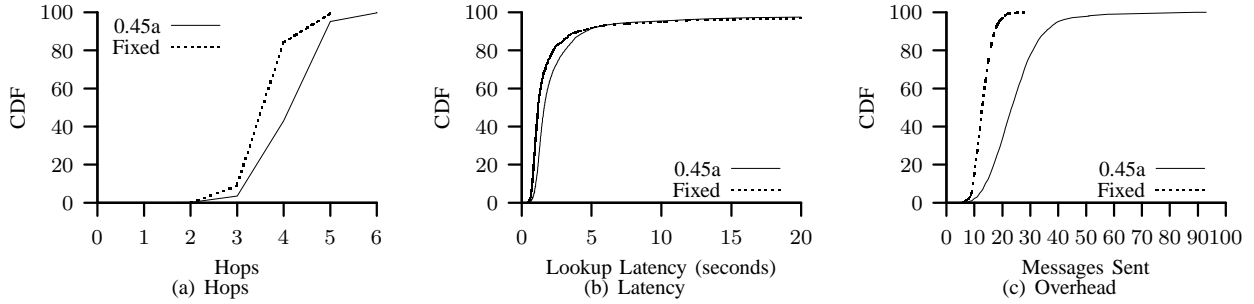

(a) Hops   (b) Latency   (c) Overhead

Fig. 7.   Performance improvement gained by fixing the bugs in eMule 0.45a

it is significantly less likely to get stuck waiting for timeouts to occur. However, as we show in Section III, few contacts in Kad are stale. This explains why loose parallelism does not show much performance improvement for this network.

To examine the communication overhead of parallel lookup, Figure 6(b) shows the number of packets sent as a function of α for the two approaches. In both cases, the overhead increases roughly linearly with α, with the loose approach generating roughly twice as many messages as the strict approach. Given that for fixed α the performance of strict and loose parallelism are quite similar, strict parallelism is the better choice for the current Kad network. To directly compare the two, Figure 6(c) factors out α by plotting the lookup hops as a function of the overhead. This figure shows that asymptotically the performance of strict and loose parallelism are surprisingly similar. A large number of messages represents the lower bound on lookup hops: no amount of increased parallelism of any kind will significantly improve performance. At the low-end, the two perform the same since the two approaches result in identical behavior for the special case $\alpha = 1$. However, the sweet-spot for strict parallelism ($\alpha = 3$) is significantly better than the sweet-spot for loose parallelism.

In summary, these observations show that strict parallelism with $\alpha = 3$ is a good choice for the current Kad network. Higher values of α and loose parallelism substantially increase overhead without much change in performance. Also, Figure 5 provides strong evidence for the correctness of our analysis in Section III.

Comparing with eMule: As part of creating kLookup, we also attempted to exactly reimplement eMule 0.46a's lookup algorithm. We validated this mode of kLookup by extending tcpdump to decode Kad packets and performing lookups for the same key using kLookup and eMule itself to verify their similarity. In the process of implementing eMule's lookup algorithm, we discovered a few bugs [17]–[19] which significantly degrade its efficiency[7].

As part of our study, we wanted to compare the performance of eMule's current lookup algorithm with and without the bugs, in the hope that it will be of use to the eMule developers. Again, we examine performance in terms of hops and latency, and overhead in terms of the number of messages. These experiments are based on more than one-thousand experiments using kLookup from unique, randomly-selected sources and destinations. With the bugs fixed, eMule's lookup algorithm is $\alpha = 3$ with loose parallelism.

Figure 7(a) presents a CDF of the number of hops to perform a lookup. The mean value is 3.59, somewhat worse than our analytically predicted value of 2.91. Without bugs, the number of hops drops to 3.08, which is closer to our predicted lower-bound. Figure 7(b) shows the latency of the two versions. In both cases, there is a significant tail (not shown) out to around 70 seconds. We see that the fixed version improves by around 1 second in most cases. The most striking difference however

---

[7]Our results are based on eMule version 0.46a, the most recent version available at the time of our study. We have been corresponding with the eMule developer team regarding these discoveries, and at least some of the reported bugs were corrected in 0.46b.
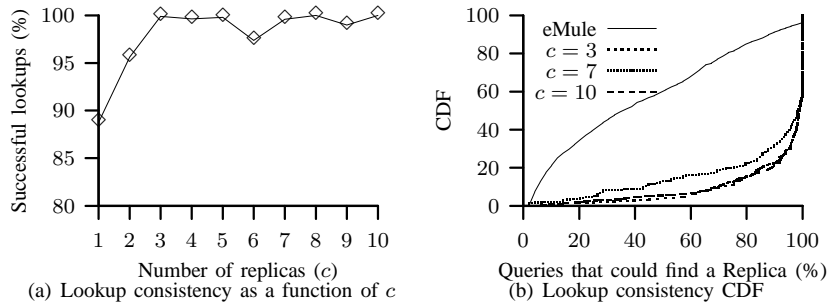
Fig. 8. Lookup Consistency

is in the overhead, as shown in 7(c). The fixed version uses roughly half as many messages on average.

VI. IMPROVING LOOKUP CONSISTENCY

Ideally, each peer in a DHT is responsible for a certain part of the DHT identifier space and lookups for any identifier should lead to the responsible peer. In practice, peer churn causes two types of inaccuracies in routing tables:

1) Peers may not yet have pointers to a recently arrived peer
2) Peers may have stale pointers to a recently departed peer

When routing tables are incorrect, it is possible for some parts of the identifier space to be unreachable for some peers. The extent of these problems is determined by how frequently the DHT validates its pointers, known as route stabilization [1], compared to the rate of churn in the system. One approach to minimize these problems is to increase the frequency of route stabilization. However, this significantly increases the bandwidth required for route maintenance.

A. Content Replication

An alternative approach is to map each identifier to the set of the c closest peers in the identifier space, rather than to only the single closest peer. The publishing operation performs a regular lookup, then searches the surrounding area to find the closest c peers. The search operation does the same, and as long as the two find any peer in common the search will succeed. Kademlia [4] takes this approach as a basic principle; however, it can be used in almost any DHT. For example, DHash [8] implements this technique over Chord. The parameter c must be chosen based on knowledge of the degree of routing table inaccuracy, to guarantee with high likelihood that multiple lookups will be able to find peers in common.

The key question is: *what is the right value of c to guarantee a certain level of reliability p?* In the following subsection, we use empirical techniques to answer this question for Kad.

B. Evaluating Lookup Consistency

To explore lookup consistency, we extended kLookup to locate the c closest points after its regular search has completed. To get an empirical measure for p, we use kLookup to perform 50 lookups to the same key, each from a different and random starting point in the Kad network. The first lookup emulates a publish operation which returns a set of peers to publish on. The following lookups to the same key emulate query operations, returning a set of peers in response to the actual query. Computing the fraction of queries that successfully find one of the target peers yields an empirical measure of the consistency, p, for that experiment. We perform the lookups as concurrently as possible to limit the effects of peer departure and arrival. For these experiments, we used strict parallelism with $\alpha = 3$. We conducted this experiment 20 times for each value of c in the interval $[1, 10]$ (*i.e.,* 1000 lookups per value of c: 20 experiments and 50 lookups per experiment).

We observed that for $c = 1$, the consistency is only 89%, meaning that 11% of the time queries fail to find the same "closest" peer as a publisher. To explore how many replicas are needed, Figure 8(a) plots p as a function of c. For the value $c = 3$, the consistency is over 99.9% across the twenty 50-lookup trials. For $c = 2$, the consistency is in between, at around 96%.

The above values are for finding *any* of the replicas. However, another issue regarding consistency is how effectively all of the replicas can be found. If one replica can always be located, but the others cannot be, then lookups will fail if the one easy-to-locate replica becomes unavailable. Therefore, for each replica we compute the number of lookups that found it, and plot it as a CDF in Figure 8(b). An ideal curve would be a vertical line at $x = 100\%$, indicating that every query found every replica. The Figure shows that the performance for the nearby-replication method is indeed good, with roughly 50% of queries able to find every replica, and 80–90% of queries able to find 80% of the replicas.

In summary, our results show that locating the three closest nodes after finding the closest peer is an effective way to cope with routing table inconsistencies. More importantly, we show that even routing table inconsistencies can be a considerable problem in practice with more than 11% of lookups failing when no replication is used.

Comparing with eMule: Currently, eMule uses a fuzzy algorithm which selects several peers as part of the endpoint set that are not necessarily the closest. In addition to our experiments for different values of c, we also conducted more than 60 experiments using eMule's algorithms for publishing and lookup. We found that eMule's approach produces 19

replicas on average and queries succeeds 99.9% of the time. While robust, this is 6.3 times more replicas than simply using $c = 3$. Furthermore, Figure 8(b) shows the CDF of the percentage of all replicas each lookup found. The performance is substantially worse than the nearest-c approach, with many replicas being found by only a few queries. For example, 50% of replicas could be found less than one-third of the time, compared to just 3% for $c = 3$. Additionally, some replicas were not found by *any* queries.

VII. Related Work

Early work on DHTs focused on introducing new DHTs [1]–[4] that each achieved $O(\log n)$ lookup hops using $O(\log n)$ state per peer. Initially, it was difficult to directly compare the performance of these DHTs, as each DHT has several tunable parameters, which might cause them to perform better or worse under different loads. For example, under low churn a DHT with a large routing table will perform better since it can achieve faster lookups and route maintenance is inexpensive. The same DHT will perform poorly under heavy churn.

Several studies [7], [10], [20]–[23] have attempted to address the issue of DHT performance under churn, in most cases using a simple Poisson model for session length. However, several measurement studies of peer-to-peer systems [24]–[28] show that session times are dramatically different from Poisson. In this study, we conduct experiments using the real Kad network, *i.e.,* under real churn.

Gummadi *et al.* [6] showed that DHTs can be broken into two components: geometry (or structure) and lookup strategy. Some DHT geometries provide greater routing flexibility than others in terms of neighbor selection or route selection. For example, in CAN a peer's neighbors are precisely defined by the geometry, while in Chord there are 2^{i-1} options for the ith neighbor, providing Chord substantially more flexibility in selecting neighbors. Their results show that more flexible systems, such as Chord and Kademlia, can achieve better performance. We utilize their division between geometry and lookup to study the lookup behavior in light of the geometry of the deployed Kad network.

Jinyang Li *et al.* [7] developed a performance-versus-cost framework (PVC) for comparing different DHTs. Their key observation is that for a given bandwidth usage, there is a minimum lookup latency that can be achieved over the entire space of DHT parameters, and vice versa. In PVC, they simulate each DHT using a wide variety of parameters and plot the best lookup latency each DHT can achieve within a given bandwidth constraint. This allows them to compare how different DHTs make the performance-versus-cost trade-off under a given load. They show that using large routing tables with infrequent stabilizations and parallel lookup achieves a better balance than other approaches, culminating in their later development of the Accordion DHT [5]. However, PVC can only draw conclusions about how well the DHTs respond to the simulated workload. While their work is useful for drawing inferences about design trade-offs, our work is aimed at optimizing tunable parameters in a DHT that is already deployed.

In summary, prior work on DHTs has been driven by analysis, simulation, and limited experiments. In each case, a model is used to approximate or estimate real-world behavior. This paper presents experiments on a deployed DHT that has approximately one million real users, and develops tools and techniques for improving its performance.

VIII. Conclusions and Future Work

This paper examines lookup performance over the Kad DHT network. We analytically derive new formulas for the expected hop count, taking into account random improvements, and demonstrate that Kademlia's use of k-buckets leads to significantly better performance than previously reported. We present new tools, kFetch and kLookup, to characterize the accuracy of routing tables in Kad, examine the impact of routing table accuracy on efficiency and consistency of the lookup operation, and experimentally verify our analysis. Furthermore, we explore two types of parallel lookup techniques and their impact on lookup efficiency and also examine the degree of replication needed to cope with routing inconsistency. While some of our empirical results are specific to Kad, our analysis applies to other prefix-matching DHTs such as Pastry and Tapestry and could be modified to handle other DHT geometries.

In our future work, we plan to measure the bandwidth eMule uses for route maintenance and study ways to maintain higher quality routing information at lower cost. We also plan to use our recent measurement-based characterization of churn in peer-to-peer systems [29] to determine the number of replicas needed to guarantee the availability of a piece of data within the network. This will include a mathematical analysis of the trade-off between republishing the data more frequently to a few peers versus publishing infrequently to many peers, followed by empirical experiments to validate our findings.

IX. Acknowledgments

References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications," *IEEE/ACM Transactions on Networking*, 2002.

[2] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," in *SIGCOMM*, 2001.

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

[4] P. Maymounkov and D. Mazieres, "Kademlia: A Peer-to-peer Information System Based on the XOR Metric," in *International Workshop on Peer-to-Peer Systems*, 2002.

[5] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek, "Bandwidth-efficient Management of DHT Routing Tables," in *Networked Systems Design and Implementation*, Boston, MA, May 2005.

[6] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica, "The Impact of DHT Routing Geometry on Resilience and Proximity," in *SIGCOMM*, Karlsruhe, Germany, Aug. 2003.

[7] J. Li, J. Stribling, F. Kaashoek, R. Morris, and T. Gil, "A Performance vs. Cost Framework for Evaluating DHT Design Tradeoffs under Churn," in *INFOCOM*, Miami, FL, Mar. 2005.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *SOSP*, Banff, AB, Canada, Oct. 2001.

[9] V. Ramasubramanian and E. G. Sirer, "The Design and Implementation of a Next Generation Name Service for the Internet," in *SIGCOMM*, Portland, OR, 2004.

[10] S. Rhea, D. Geels, and J. Kubiatowicz, "Handling Churn in a DHT," in *USENIX*, 2004, pp. 127–140.

[11] S. S. Lam and H. Liu, "Failure Recovery for Structured P2P Networks: Protocol Design and Performance Evaluation," in *SIGMETRICS*, New York, NY, June 2004.

[12] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A Resilient Global-Scale Overlay for Service Deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, Jan. 2004.

[13] B. Leong, B. Liskov, and E. D. Demaine, "EpiChord: Parallelizing the Chord Lookup Algorithm with Reactive Routing State Management," in *nternational Conference on Networks*, Nov. 2004.

[14] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris, "Designing a DHT for Low Latency and High Throughput," in *NSDI*, Berkeley, CA, 2004.

[15] D. D. Clark, "Design Philosophy of the DARPA Internet Protocols," *ACM SIGCOMM Computer Communication Review*, vol. 25, no. 1, Jan. 1995.

[16] D. Stutzbach and R. Rejaie, "Capturing Accurate Snapshots of the Gnutella Network," in *Global Internet Symposium*, Miami, FL, Mar. 2005, pp. 127–132.

[17] D. Stutzbach, "Csearch::jumpstart() Deletes Wrong Contacts," http://forum.emule-project.net/index.php?showtopic=81094, June 2005.

[18] ——, "Kad Lookups Start In Wrong Place," http://forum.emule-project.net/index.php?showtopic=81113, June 2005.

[19] ——, "M_best Not Initialized And Updated Properly," http://forum.emule-project.net/index.php?showtopic=81020, June 2005.

[20] S. Krishnamurthy, S. El-Ansary, E. Aurell, and S. Haridi, "A Statistical Theory of Chord under Churn," in *International Workshop on Peer-to-Peer Systems (IPTPS)*, Ithaca, NY, Feb. 2005.

[21] D. Liben-Nowell, H. Balakrishnan, and D. Karger, "Analysis of the Evolution of Peer-to-Peer Systems," in *Principles of Distributed Computing*, Monterey, CA, July 2002.

[22] J. Li, J. Stribling, T. M. Gil, R. Morris, and F. Kaashoek, "Comparing the performance of distributed hash tables under churn," in *International Workshop on Peer-to-Peer Systems*, 2004.

[23] R. Mahajan, M. Castro, and A. Rowstron, "Controlling the Cost of Reliability in Peer-to-Peer Overlays," in *International Workshop on Peer-to-Peer Systems*, 2003.

[24] S. Sen and J. Wang, "Analyzing Peer-To-Peer Traffic Across Large Networks," *IEEE/ACM Transactions on Networking*, vol. 12, no. 2, pp. 219–232, Apr. 2004.

[25] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan, "Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload," in *SOSP*, 2003.

[26] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, "The Bittorrent P2P File-sharing System: Measurements and Analysis," in *International Workshop on Peer-to-Peer Systems (IPTPS)*, Ithaca, NY, Feb. 2005.

[27] M. Izal, G. Urvoy-Keller, E. W. Biersack, P. A. Felber, A. A. Hamra, and L. Garces-Erice, "Dissecting BitTorrent: Five Months in a Torrent's Lifetime," in *PAM*, Apr. 2004.

[28] J. Chu, K. Labonte, and B. N. Levine, "Availability and Locality Measurements of Peer-to-Peer File Systems," in *ITCom: Scalability and Traffic Control in IP Networks II Conferences*, July 2002.

[29] D. Stutzbach and R. Rejaie, "Characterizing Churn in Peer-to-Peer Networks," University of Oregon, Eugene, OR, Tech. Rep. CIS-TR-2005-03, May 2005.