Why Kad Lookup Fails

Hun J. Kang, Eric Chan-Tin, Nicholas J. Hopper, Yongdae Kim
University of Minnesota - Twin Cities
{hkang,dchantin,hopper,kyd}@cs.umn.edu

Abstract

A Distributed Hash Table (DHT) is a structured overlay network service that provides a decentralized lookup for mapping objects to locations. In this paper, we study the lookup performance of locating nodes responsible for replicated information in Kad – one of the largest DHT networks existing currently. Throughout the measurement study, we found that Kad lookups locate only 18% of nodes storing replicated data. This failure leads to limited reliability and an inefficient use of resources during lookups. Ironically, we found that this poor performance is due to the high level of routing table similarity, despite the relatively high churn rate in the network. We propose solutions which either exploit the high routing table similarity or avoid the duplicate returns using multiple target keys.

1 Introduction

A Distributed Hash Table (DHT) is a structured overlay network protocol that provides a decentralized lookup service mapping objects to peers. In a large peer-to-peer (P2P) network, this service can also provide means of organizing and locating peers for use in higher-level applications. This potential to be used as a fundamental building block for large-scale distributed systems has led to an enormous body of work on designing highly scalable DHTs. Despite this, only a handful of DHTs have been deployed on the Internet-scale: Kad, Azureus [1], and Mainline [5], all of which are based on the Kademlia protocol [6]. These widely deployed DHTs inherently face diverse user behaviors and dynamic situations that can affect DHTs' performance. Therefore, there have been studies measuring various DHT's aspects including node distribution, user behaviors, and dynamics of peer participation called churn [8, 10, 13]. Some studies [9, 12] also looked at the performance of lookups which are fundamental functions of DHTs. However, most of the previous work have focused on reducing lookup delay time.

In this paper, we study Kad lookup performance regarding reliability and efficiency in the use of resources. In Kad, object information is stored at multiple nodes (called *replica roots*). Therefore, a peer can retrieve the information once it finds at least one replica root. However, we observe that *8% of searching peers cannot find any replica roots immediately after publishing*, which means they are unable to retrieve the information. Even worse, *25% of searching peers fail to locate the information 10 hours after storing the information*. This poor performance is due to inconsistency between storing and searching lookups; Kad lookups for the same objects map to an inconsistent set of nodes. From our measurement, only 18% of replica roots located by storing and searching lookups are the same on average. Moreover, this lookup inconsistency causes an inefficient use of resources. We also find that *45% of replica roots are never located and thus used by any searching peers for rare objects*. Furthermore, when many peers search for popular information stored by many peers, *85% of replica roots are never used and only a small number of the roots suffer the burden of most requests*. Therefore, we can see that Kad lookups are not reliable and waste resources such as bandwidth and storage for unused replica roots.

Why are the nodes located by publishing and searching lookups inconsistent? Past studies [2, 12] on Kademlia-based networks have claimed that lookup results are different because routing tables are inconsistent due to dynamic node participation (*churn*) and slow routing table convergence. We question this claim and examine entries in routing tables of nodes around a certain key space. Surprisingly, the routing table entries are much more similar among the nodes than expected. Therefore, these nodes return a similar list of their neighbors to be contacted when they receive requests for the key. However, the Kad lookup algorithm does not consider this high level of similarity in routing table entries. As a result, this duplicate contact list limits the unique number of located replica roots around the key.

The consistent lookups enable reliable information search although some copies of the information are not available due to node churn or failure. Then they can also provide the same level of reliability with the smaller number of required replica roots compared to inconsistent lookups, which means efficiently use resources such as bandwidth and storage. Furthermore, consistent lookups locating multiple replica roots provide a way to load-balancing. Therefore, we propose algorithms considering the routing table similarity in Kad and show how improved lookup consistency affects the performance. These solutions can improve lookup consistency up to 90% (and 80%) and eventually lead to guaranteeing reliable lookup results while providing efficient resource use and load-balancing. Our solutions are completely compatible with existing Kad clients, and thus incrementally deployable.

2 Background

Kad is a Kademlia-based DHT for P2P file sharing. It is widely deployed with more than 1.5 million simultaneous users [8] and is connected to the popular eDonkey file sharing network. The aMule and eMule clients are the two most popular clients used to connect to the Kad network. We examine the performance of Kad using aMule (at the time of writing, we used aMule version 2.1.3), a popular cross-platform open-source project. The other client, eMule, also has a similar design and implementation.

Kad organizes participating peers into an overlay network and forms a key space of 128-bit quantifiers among peers. (We interchangeably use a peer and a node in this paper.) It "virtually" places a peer onto a position in the key space by assigning a node identifier (Kad ID) to the peer. The distance between two positions in the key space is defined as the value of a bitwise XOR on their corresponding keys. In this sense, the more prefix bits are matched between two keys, the smaller the distance is. Based on this definition, we say that a node is "close" (or "near") to another node or a key if the corresponding XOR distance is small in the key space. Each node takes responsibility for objects whose keys are near its Kad ID.

As a building block for the file sharing, Kad provides two fundamental operations: PUT to store the binding in the form of (*key, value*) and GET to retrieve *value* with *key*. These operations can be used for storing and retrieving objects for file information. For simplicity, we only consider keyword objects in this paper because almost the same operations are performed in the same way for other objects such as file objects. Consider a file to be shared, its keyword, and keyword objects (or bindings) where *key* is the hash of the keyword and *value* is the meta data for the file at a node responsible for the key. Peers who own the file *publish* the object so that any user can search the file with the keyword and retrieve the meta data. From the information in the meta data, users interested in the file can download it. Because a peer responsible for the object might not be available, Kad uses the data replication approach; the binding is stored at r nodes (referred to as *replica roots* and r is 10 in aMule). To prevent binding information from being stored at arbitrary locations, Kad has a "search tolerance" that limits the set of potential replica roots for a target.

In both PUT and GET operations, a Kad lookup for a target key (T) performs the process of locating nodes which are responsible for T (i.e., nodes near T). Kad lookup is mainly composed to two phases (called *Phase1* and *Phase2* in convenience). In *Phase1*, a peer finds a route to T. A querying node Q initially picks α nodes (*contacts*) which have the longest matched prefix bit length to T from its routing table and, "queries" those contacts by sending KADEMLIA_REQ messages for T (α is 3 in Kad). Each of queried nodes selects β contacts closest to the target from its routing table, and returns those contacts in a KADEMLIA_RES message (β is 2 in GET and 4 in PUT). Once a queried node sends a KADEMLIA_RES responding to a KADEMLIA_REQ, it is referred to as a "located" node. Q "learns" the returned contacts from queried nodes and picks the α closest contacts from its learned nodes. This lookup step of learning and querying is performed by querying node Q, thus Kad lookup is called *iterative*. The querying node can approach to the node closest to T by repeating lookup steps until it cannot find any nodes closer to T than those it has already learned. The number of lookup steps is bounded to $O(\log N)$ where N is the number of nodes in the Kad network. In *Phase2*, Q locates more nodes near T by querying already learned nodes to publish binding information to multiple nodes or to search the information from those nodes. Publish requests (PUBLISH_REQ) and search requests (SEARCH_REQ) are sent only in *Phase2*, which is an efficient strategy because the replica roots exist near the target and search nodes can locate the replica roots with high probability. This process repeats until termination conditions are reached – a specific amount of binding information are obtained, a predetermined number of responses are received, or a time-out occurs. More detailed description of Kad lookup and an illustration are provided in [4,9].

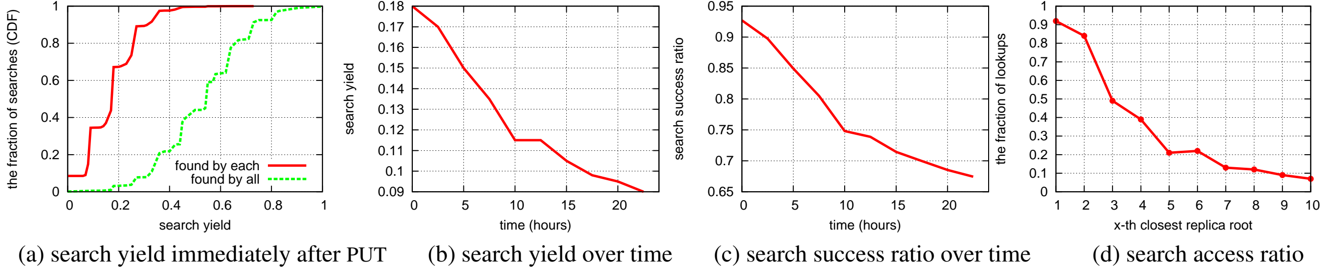
3 Evaluation of Kad Lookup Performance

In this section, we evaluate the performance of Kad focusing on the consistency between lookups through a measurement study. We first describe the experimental setup of our measurements. We then examine the inconsistency problem between publishing and searching lookups and show how this lookup inconsistency affects the Kad lookup performance in reachability and efficiency in the use of resource.

3.1 Experimental Setup

We ran a Kad node using an aMule client on machines having static IP addresses without a firewall or a NAT. Kad IDs of the peers were randomly selected so that the IDs were uniformly distributed over the Kad key space. A publishing peer shared a file in the following format "*keywordU.extension*" (e.g., "*as3d1f0goa.zx2cv7bn*"), where *keywordU* is a 10-byte randomly-generated keyword, and *extension* is a fixed string among all our file names, used for identifying our published files. This allows us to publish and search keyword objects of the files not duplicated with existing ones. For each experiment, one node published a file and 32 nodes searched for that file by using *keywordU*. We ran nodes which had different Kad IDs and were bootstrapped from different nodes in the Kad network to avoid measuring the performance in a particular key space. We repeated the experiments with more than 30,000 file names.

In order to empirically evaluate the lookup performance, we define the following metrics.



(a) search yield immediately after PUT (b) search yield over time (c) search success ratio over time (d) search access ratio

Figure 1. Performance of lookup

Search yield measures the fraction of replica roots found by a GET lookup process following a PUT operation, implying how "reliably" a node can search a desired file, and is calculated as

$$\frac{\text{number of the replica roots located by a GET lookup}}{\text{number of published replica roots}}.$$

Search success ratio is the fraction of GET operations which retrieve a *value* for a *key* from any replica roots located by a search lookup (referred to as successful searches), implying whether a node can find a desired object or not, and is calculated as

$$\frac{\text{number of successful searches}}{\text{number of total searches}}.$$

Search access ratio measures the fraction of GET lookups which finds a particular replica root, implying how likely the replica root is to be found through lookups with a corresponding key. It is calculated as (for each replica root)

$$\frac{\text{number of searches which locate a replica root}}{\text{number of total searches for the corresponding key}}.$$

For load balancing, the distribution of search access ratios among replica roots should not be skewed.

3.2 Performance Results

We evaluate the lookup ability to locate replica roots by measuring the search yield. Then, we show how search yield affects the Kad lookup performance by examining the search success ratio and search access ratio.

Figure 1(a) shows the distribution of the search yield immediately after PUT operations ("found by each" line). The average search yield is about 18%, meaning that only one or two replica roots are found by a GET lookup (because the replication factor is 10 in aMule). In addition, about 80% of the total lookups locate fewer than 3 replica roots (25% search yield). This result is quite disappointing, since this means that one cannot find a published file 80% of the time when these three nodes leave the network, even though 7 more replica roots exist. Figure 1(b) shows that the search yield continuously decreases over time during a day from 18% to 9%, which means nodes are less likely to find a desired file as time goes by.

This low search yield directly implies poor Kad lookup performance. A search is successful unless the search lookup is not able to find any replica roots (i.e., unless the search yield is 0). This is because binding information can be retrieved from any located replica root. Figure 1(c) shows the search success ratio over time. Immediately after publishing a file, the search success ratio is 92% implying that 8% of the time we cannot find a published file. This result matches the statistics in Figure 1(a) that 8% of searches have a 0 search yield. This result is somewhat surprising since we expected that i) there exists at least 10 replica roots near the target, and ii) DHT routing should guarantee to find a published file. Even worse, the search success ratio continuously decreases over time during a day from 92% to 67% before re-publishing occurs. This degradation of the search success ratio over time is caused by churn in the network. In Kad, no other peers take over the file binding information stored in a node when the node leaves the Kad network. The mechanism to mitigate this problem caused by churning is that the publishing peer performs PUT every 24 hours for keyword objects.

Because GET lookups are able to find a small fraction of replica roots, there must be unused replica roots as shown in Figure 1(a). In "found by all" line, 55% of replica roots are found by 32 lookups on average, so 45% of replica roots are never located by any lookup. From this fact, we can conjecture that the replica roots found by each GET lookup are not disjointed. This inference can be checked in Figure 1(d) showing the search access ratio of each replica root. In this figure, nodes in the X-axis are sorted by distance to a target and we can easily see that most of lookups locate the two closest replica roots, but the other replica roots are not contacted by lookups. This distribution of the search access ratios indicates that the load of replica roots is highly unbalanced. Overall, the current Kad lookup process cannot efficiently locate more than two replica roots. Thus, resources such as storage and network bandwidth are uselessly wasted for storing and retrieving replicated binding information.

4 Analysis of Poor Lookup Performance

In the previous section, we showed that the poor performance of Kad lookups (18% search yield) is due to the inconsistent lookup results. In this section, we analyze the root causes of these lookup inconsistencies. Previous studies [2, 12] of Kademlia-based networks have blamed membership churn, an inherent part of every file-sharing appli-

cation, as the main contributing factor to these performance issues. These studies claim that network churn leads to routing table inconsistencies as well as slow routing table convergence. These factors then lead to non-uniform lookup results [2, 12]. We question this claim and identify the underlying reasons for the lookup inconsistency in Kad. First, we analyze the entries within routing tables, specifically focusing on consistency and responsiveness. Next, we dissect the poor performance of Kad lookups based upon characteristics of routing table entries.

4.1 Characterizing Routing Table Entries

In this subsection, we empirically characterize routing table entries in Kad. We first explain the distribution of nodes in the key space, and then examine consistency and responsiveness. By consistency we mean how similar the routing tables of nodes around a target ID are, and by responsiveness we mean how well entries in the routing tables respond when searching nodes query them.

Node Distribution. Kad is known to have 1.5 million concurrent nodes with IDs uniformly distributed [12]. Because we know the key space is uniformly populated and we know the general size of the network, we can derive n_L, the expected number of nodes that exactly match L prefix bits with the target key. Let N be the number of nodes in the network and n'_L be the expected number of nodes which match at least L prefix bits with the target key. Then, the expected match between any target and the closest node to that target is $2^{\log_2 N}$ bits. n'_L increases exponentially as L decreases (nodes are further from the target). Thus, n'_L and n_L can be computed as follows:

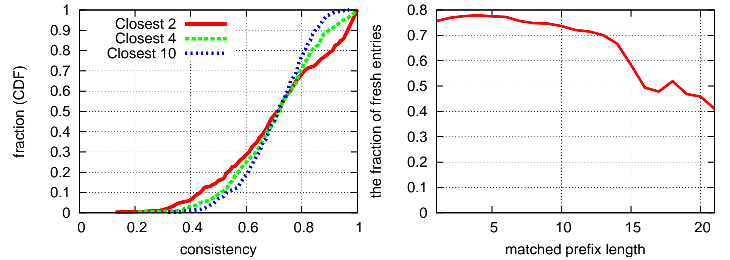
$$n'_L = 2^{\log_2 N - L} \qquad n_L = n'_L - n'_{L+1} = 2^{\log_2 N - L - 1}$$

When N is 1.5 million, the expected number of nodes for each matched prefix length is as follows:

L	21	20	19	18	17	16
n_L	0.35	0.71	1.43	2.86	5.72	11.44
n'_L	0.71	1.43	2.86	5.72	11.44	22.88

Routing Table Collection. To further study Kad, we collected routing table entries of peers located around given targets. We built a crawler that, given a target T, will crawl the Kad network looking for all the nodes close to T. If a node matches at least 16 bits with T, its routing table is polled. The number 16 is chosen empirically since there should be about 23 nodes at more than or equal to 16 bit matched prefix length in Kad (more than twice the number of replica roots). Those nodes are the ones "close" to T.

Polling routing tables can be performed by sending the same node multiple KADEMLIA_REQ messages for different target IDs. Each node will then return the routing table entries that are closest to these target IDs. A node's whole routing table can thus be obtained by sending many KADEMLIA_REQ. For every node found or polled, a HELLO_REQ is sent to determine whether that node is alive. For this study, we select more than 600 random target IDs and retrieve the routing tables of approximately 10,000 distinct Kad peers. We then examine the two properties mentioned above: consistency and responsiveness.



(a) Similarity among all nodes (b) Response ratio of nodes

Figure 2. Statistics on routing tables

View Similarity. We measure the similarity of routing tables. Let P be the set of peers close to the target ID T. A node A is added to P if the matched prefix length of A with T is at least 16 . We define a peer's view v to T as the set of k closest entries in the peer's routing table. This is because when queried, peers select the k *closest* entries from their routing tables and return them. We selected 2, 4, and 10 as k because 2 is the number of contacts returned in SEARCH_REQ, 4 for PUBLISH_REQ and 10 for FIND_NODE.

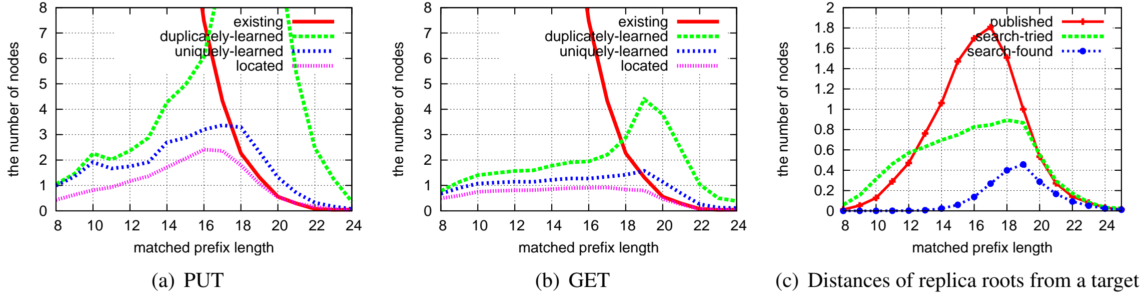
We measure the distance d (or the difference) between views (v_x, v_y) of two peers x and y in P as

$$d(v_x, v_y) = \frac{|v_x - v_y| + |v_y - v_x|}{|v_x| + |v_y|}$$

where $|v_x|$ is the number of entries in v_x. $d(v_x, v_y)$ is 1 when all entries are different and 0 when they are the same. The similarity of views to the target is defined as $1 - dissimilarity$ where *dissimilarity* is the average distance among the views of peers in P. Then, the level of this similarity indicates how similar close-to-T entries in the routing tables of nodes around the target T are. For simplicity, we call this *the similarity of routing table entries*.

Figure 2(a) shows that the average similarity of routing table entries is 70% based on comparisons of all nodes in P. This means that among any two routing tables of nodes in P, close to T, 70% of entries are identical. Therefore, peers return similar and duplicate entries when a searching node queries them for T. The high similarity values indicate that the closest node has a similar view to a target with the other close nodes in P.

Responsiveness. In Figure 2(c), we examine the number of responsive (live) contacts normalized by the total number of contacts close to a given target key. The result shows that around 80% of the entries in the routing tables respond to our requests, up to a matched prefix length of 15. The fraction of responsive contacts decreases as the matched prefix length increases because in the current aMule/eMule implementations, peers do not check the liveness of other peers close to its Kad ID as often as nodes further away [12].



(a) PUT (b) GET (c) Distances of replica roots from a target

Figure 3. Number of nodes at each distance from a target

4.2 Analysis of Lookup Inconsistency

In the previous subsection, we observed that the routing table entries of nodes are similar and only half of the nodes near a specific ID are alive. From this observation, we investigate why Kad lookups are inconsistent and then present analytical results.

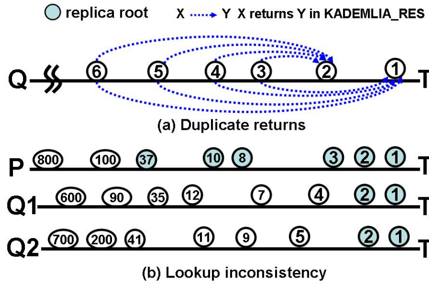


Figure 4. Illustration of how a lookup can be inconsistent

We explain why Kad lookups are inconsistent using an example, shown in Figure 4. A number (say k) in a circle means that the node is the $k-th$ closest node to the target key T in the network. Only nodes located by the querying nodes are shown. We first see how the high level of the routing table similarity affects the ability of locating nodes close to T. Peers close to T have similar close-to-T contacts in their routing tables. Thus, the same contacts are returned multiple times in KADEMLIA_RES messages and the number of learned nodes is small. In Figure 4(a), node Q learns only the two closest nodes because all queried nodes return node 1 and node 2.

The failure to locate nodes close to a target causes inconsistency between lookups for PUT and GET. A publishing node only finds a small fraction of the nodes close to the target. In Figure 4(b), node P locates three closest nodes (nodes 1, 2, and 3) as well as less useful nodes farther from the target T. Node P then publishes to the r "closest" nodes among these located nodes, assuming that those nodes are the very closest to the target ($r = 10$ but only 6 nodes are shown in the figure). Note that some replica roots (e.g. node 37) are actually far from T and many closer nodes exist. Similarly, searching nodes ($Q1$ and $Q2$) find only a subset of the actual closest nodes. These querying nodes then send SEARCH_REQ to the located nodes (referred to as "search-tried"). However, only a small fraction the search-tried nodes are replica roots (referred to as "search-found"). From this example, we can clearly see that the querying nodes will obtain binding information only from the two closest nodes (node 1 and node 2) out of 10 replica roots.

We next present analytical results supporting our reasoning for inconsistent Kad lookups. Figures 3(a) and (b) show the average number of different types of nodes at each matched prefix length for PUT and GET, respectively. The "existing" line shows the number of nodes found by our crawler at each prefix length and matches with the expected numbers provided in the previous subsection. The "duplicately-learned" line shows the total number of nodes learned by a searching node including duplicates and the "uniquely-learned" line represents the distinct number of nodes found without duplicates. When a node is included in 3 KADEMLIA_RES messages, it is counted as 3 in the "duplicately-learned" line and 1 in the "uniquely-learned" line. We can see that some nodes very close to T are duplicately returned when a querying node sends KADEMLIA_REQ messages. In other words, the number of "uniquely-learned" nodes is much smaller than the number of "duplicately-learned" nodes when they are very close to T. For instance, there is one existing node at 20 matched prefix length (in "uniquely-learned" line), and it is returned to a querying node 5 times in PUT and 3.8 times in GET ("duplicately-learned" lines). To further compound the issue, the number of "located" nodes is half that of "uniquely-learned" nodes because, on average, 50% of the entries in the routing tables are stale. In other words, half of the learned contacts no longer exist in the network. As a result, a PUT lookup locates only 8.3 nodes and a GET lookup finds only 4.5 nodes out of the 23 live nodes which have more than 16 matched prefix length with the target. Thus, we can see that the duplicate contact lists and stale (dead) routing table entries cause a Kad lookup to locate only a small number of the existing nodes close to the target.

Since the closest nodes are not located, PUT and GET operations are inadvertently performed far from the target. Figure 3(c) shows the average number of "published" (denoted as p_L), "search-tried" (denoted as s_L), and "search-found" (denoted as f_L) nodes for each matched prefix length L. We clearly see that more than half of the nodes

which are "published" and "search-tried" match less than 17 bits with the target key. We can formulate the expected number of replica roots $E[f_L]$ located by a GET lookup for each L. Let N be the number of nodes in the network and n_L be the expected number of nodes which match L prefix bits with the target key. Then f_L is computed as follows:

$$E[f_L] = s_L * \frac{p_L}{n_L} = s_L * \frac{p_L}{2^{\log_2 N - L - 1}}$$

The computed values of $E[f_L]$ match with f_L from the experiments shown in Figure 3. From the formula, $E[f_L]$ is inversely proportional to L because n_L increases exponentially. Thus, although a GET lookup is able to find some of the closest nodes to a target, not all of these nodes are replica roots because a PUT operation publishes binding information to some nodes really far from the target as well as nodes close to the target. For a GET lookup to find all the replica roots, that is, all the nodes located by PUT, the GET operation has to contact exactly the same nodes – this is highly unlikely. This is the reason for the lookup inconsistency between PUT and GET operations.

5 Improvements

We already saw how the lookup inconsistency problem affects the lookup performance in Section 3. This problem limits the lookup reliability and wastes resources. In this section, we describe several possible solutions to increase lookup consistency. Then, we see how well the proposed solutions improve Kad lookup performance. Moreover, we evaluate the overhead of the new improvements.

5.1 Solutions

Tuning Kad parameters. Tuning parameters on Kad lookups can be a trivial attempt to improve Kad lookup performance. The number of *replica roots* ($r = 10$) can be increased. Although this change could slightly improve performance, it will still be ineffective because close nodes are not located and the replica roots that are far from the target will still exist. The timeout value (t = 3 seconds) for each request can also be decreased. We do not believe this will be useful either since this change results in more queries being sent and more duplicates being received. The number of returned contacts in each KADEMLIA_RES can also be increased ($\beta = 2$ for GET and $\beta = 4$ for PUT). Suppose that 20 contacts are returned in each KADEMLIA_RES. Then, 20 nodes close to a target can be located (if all contacts are alive) even though returned contacts are duplicated. However, this increases the size of messages by an order of 10 for GET (5 for PUT). Finally, the number of contacts queried at each iteration ($\alpha = 3$) can be increased. This would increase the number of contacts queried at each iteration step, thus, increasing the ability to find more replica roots. However, this approach will result in more messages sent and even more duplicate contacts received.

Querying only the closest node (*Fix1*.) A solution of querying only the closest node exploits the high similarity in routing table entries. After finding the closest node to a particular target, a peer asks for its 20 contacts closest to the target. From our experimental results, a lookup finds the closest node with 90% probability, and always locates one of the nodes which matches at least 16 prefix bits with the target. Therefore, the expected search yield is $0.9 \times 1 + 0.1 \times 0.7 = 0.97$ (90% chance of finding the closest node from Figure 1(d), 10% chance of not finding the closest node, and 70% similarity among routing table entries from Section 4). We note that this simple solution comes as a direct result of our measurements and analysis.

Avoiding duplicates by changing target IDs (*Fix2*.)

Because of the routing table similarity, duplicate contacts are returned from queried nodes and this eventually limits the number of located nodes close to a target. To address this problem, we propose *Fix2* that can locate enough nodes closest to a target.

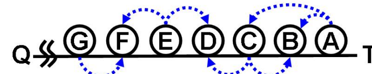
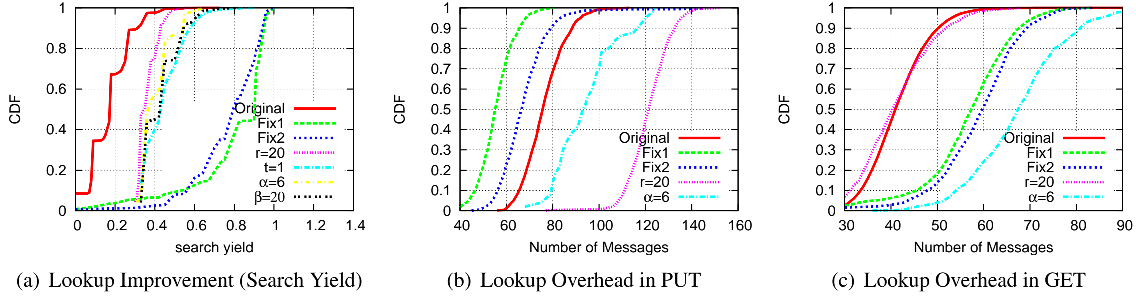


Figure 5. Lookup algorithm for *Fix2*

Our new lookup algorithm is illustrated in Figure 5 in which peer Q attempts to locate nodes surrounding target T. Assume that nodes (A, B,..., F) close to target T have the same entries around T in their routing tables and all entries exist in the network. We define KADEMLIA_REQ by adding a target notation; KADEMLIA_REQ (T) is a request to ask a queried node to select β contacts closest to target T, and return them in KADEMLIA_RES. In the original Kad, Q receives duplicate contacts when it sends KADEMLIA_REQ (T) to multiple nodes. In a current Kad GET lookup (β = 2), the only three contacts (A, B, and C) would be returned. However, *Fix2* can learn more contacts by manipulating target identifiers in KADEMLIA_REQ. Once the closest node A is located (i.e., *Phase2* is initiated – see Section 2), Q sends KADEMLIA_REQ by replacing the target ID with other learned node IDs ($\{B$, C,..., $F\}$). In other words, Q sends KADEMLIA_REQ (T') instead of KADEMLIA_REQ (T) where $T' \in \{B$, C,..., $F\}$. Then, the queried nodes return contacts (neighbors) closest to themselves. In this way, Q can locate most of the nodes close to the "real" target T.

In order to effectively exploit *Fix2*, we separate the lookup procedures for PUT and GET. These operations have different requirements according to their individual purposes; while GET requires a low delay in order to satisfy users, PUT requires publishing the file information where other peers can easily find it (it does not require a low delay). However, Kad has identical lookup algorithms for both PUT and GET, where a publishing peer starts PUT as soon as *Phase2* is initiated even when most of the close nodes are not located. This causes the copies of bindings to be stored



(a) Lookup Improvement (Search Yield) (b) Lookup Overhead in PUT (c) Lookup Overhead in GET

Figure 6. Lookup improvement

far from the target. Therefore, we modify only a PUT lookup to delay sending PUBLISH_REQ until enough nodes close to the target are located while GET is performed without delay. In our implementation, we wait one minute (the average time to send the last PUBLISH_REQ is 50 seconds in our experiments) before performing a PUT operation expecting that most of the close nodes are located during that time.

5.2 Performance Comparisons

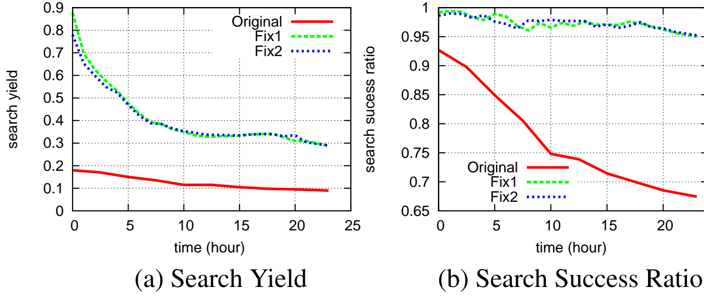
We next compare the performance improvement of the proposed algorithms. With the results obtained from the same experiments explained in Section 3, we show that our solutions significantly improve lookup performance.

Search yield can be used to clearly explain the lookup consistency problem. Figure 6(a) shows the search yield for each solution. Simply tuning parameters (number of replica roots, timeout value, α, β) exhibit search yields of $35\% \sim 42\%$. *Fix1* has an improvement of 90%, on average, which is slightly less than expected because some replica roots leave the network or do not respond to the GET requests. *Fix2* improves the search yield to 80%, on average, but provides more reliable and consistent results. For a search yield of 0.4, 99% of *Fix2* lookups have higher search yields compared to 95% of *Fix1* lookups. Since *Fix1* relies only on the closest node, the lookup results may be different when the closest node is different (due to churn). This can be observed when a new node closer to the target churns in because it could have different routing table entries from the other nodes close to it.

We next look at the overhead in the number of messages sent for both PUT and GET operations. The number of messages sent by each algorithm for PUT is shown in Figure 6(b). *Fix1* and *Fix2* use 72% and 85% fewer messages respectively because the current Kad lookup contacts more nodes than the proposed algorithms. After reaching the node closest to a target, the current Kad lookup locates only a small fraction of "close" nodes in *Phase2* (the number of nodes found within the search tolerance is fewer than 10). Thus, the querying node repeats *Phase1* again and contacts nodes further from the target until it can find more than 10 nodes within the search tolerance. The overhead for parameter tunings is higher than the original Kad implementation, as expected. Increasing the number of replica roots implies that 20 replica roots need to be found. Since it is already difficult (having to restart *Phase1*) to find 10 replica roots, it is even more difficult to find 20 replica roots – thus, the number of messages sent in PUT is much higher than for *Original*. Contacting more nodes at each iteration (increasing α from 3 to 6) increases the number of messages sent, and shortening the timeout (from 3 to 1) incurs a similar overhead. However, we observe that the overhead is not as high as increasing the number of replica roots because when r is increased, *Phase1* is restarted a couple of times – the Kad lookup process has difficulties locating 10 replica roots, thus trying to locate 20 replica roots means that *Phase1* has to take place more times.

The message overhead for GET operations is shown in Figure 6(c). *Fix1* and *Fix2* sent $1.45 \sim 1.5$ times more messages than the current Kad lookup. In the current Kad implementation, only a few contacts out of the learned nodes are queried during *Phase2* – thus, few KADEMLIA_REQ and SEARCH_REQ are sent. Even if the original Kad lookup implementation is altered to send more requests, this would not increase the search yield due to the number of messages wasted in contacting far away nodes from the target because of duplicate answers. Increasing the number of replica roots to 20 uses roughly the same number of messages as *Original* for GET because increasing the number of replica roots does not affect the search lookup process. Increasing the number of returned contacts (α), however, does increase the number of messages sent in GET because 6 nodes are queried instead of 3 nodes (a shorter timeout has a similar overhead). The overhead due to this tweaking is even higher than *Fix1* or *Fix2* because our algorithms increase α only after finding the closest node.

Fix1 and *Fix2* produce much higher performance than solutions changing parameters. Moreover, the overhead of these two solutions are lower than *Original* for PUT and slightly higher for GET. The overhead for the other solutions are much higher. We next compare only these two algorithms, *Fix1* and *Fix2*, as they are the most promising ones. Figure 7(a) shows that the search yield of the algorithms decreases as time goes on because of churns. However, it is still higher than the original Kad lookup. Although they show a very similar performance level, the variation of per-



(a) Search Yield (b) Search Success Ratio

Figure 7. Lookup performance over time

formance in *Fix1* is slightly higher than *Fix2* due to the possibility of a closer node churning in. Due to the high search yield, both *Fix1* and *Fix2* enable a peer to successfully find a desired object at any time with a higher probability than the original Kad lookup. In Figure 7(b), the search success ratios for our proposed algorithms are almost 1 after publishing while the ratio for the original Kad is 0.92. Even after 20 hours, the ratios for the solutions are 0.96 while the ratio for the original Kad is 0.68.

Overall, *Fix1* and *Fix2* significantly improve the performance of the Kad lookup process with little overhead in terms of extra messages sent compared to the other possible algorithms and the original one. *Fix1* is simple and can be used in an environment with a high routing table consistency. The downside of *Fix1* is that it is not as reliable as *Fix2* in some cases. Suppose that a new node joins – and becomes the closest node, but its routing table entries close to the target are not replica roots which were routing table entries of the "old" closest node. Then, a GET operation might not be able to find these replica roots. However, a querying client can locate most of the closest nodes around a target in *Fix2* even though the "old" closest node leaves the network or a joining node becomes the closest node. Therefore, *Fix2* can be used for applications which require strong reliability and robustness.

6 Object Popularity and Load Balancing

Many peers publish or search popular objects (or keywords such as "love") and some nodes responsible for the objects receive a large number of requests. To examine severity of this load balancing issue, we perform experiments on the lookup for popular objects in Kad network. The experiments are composed of two steps: i) finding the most of replica roots of popular objects in the network using our crawler and ii) examining the number of the replica roots located by Kad lookups. We select objects whose name match with keywords extracted from the 100 most popular items in Pirate Bay [14] on April 5, 2009. We modify our crawler used for collecting routing table entries so that it could send SEARCH_REQ. We consider a node to be a replica root if it returns binding information matching with a particular target keyword. Then, we run 420 clients to search bindings for the objects using those keywords.

We evaluate Kad lookup performance by investigating the number of replica roots located by Kad searches. First, we examine if a client was able to retrieve bindings. In the experiments, each client could find at least one replica root and retrieve binding information – the search success ratio was 1. Next, we discuss if Kad lookups use resources efficiently. Figure 8(a) shows the average number of the replica roots located by all clients at each prefix matched length. The "existing" line represents the actual replica roots observed by our crawler. The "distinctly-found" line indicates the number of the unique replica roots, but the "duplicately-found" line includes duplicates. For example, when one replica root is located by 10 clients, it is counted as 1 in the "distinctly-found" line but as 10 in the "duplicately-found" line. Overall, our results indicate that 85% of all replica roots were not located during search lookups, and, therefore, never provide the bindings to the clients. Our crawler found a total of 598 replica roots for each keyword on average. However, our clients located only 93 replica roots during the searches, which was only 15% of the total replica roots. Furthermore, we could observe a load-balancing problem in Kad lookups. Most of the "unlocated" replica roots are far from the target (low matched prefix length). At 11 matched prefix length, only 10 out of 121 replica roots were located. On the other hand, nodes close to the target were always located but received requests from many clients. At more than or equal to 20 matched prefix length ("20+" in the figure), there were only 1.4 unique replica roots (in the both "existing" and "'distinctly-found" lines) implying that all those replica roots were located by clients. However, there were 201 "duplicate-found" roots, which means that one replica root received search requests from 141 clients, on average.

To better illustrate the load-balancing problem, we define the average lookup overhead of replica roots at L prefix matched length as:

$$Load_L = \frac{\text{number of duplicately-found replica roots}}{\text{number of existing replica roots}}$$

A high $Load_L$ value means that there are numerous nodes at matched prefix length L which received search requests. The "real" line in Figure 8(d) shows the load for the above experiments. The load was high for the high matched prefix length (replica roots close to the target) while the load was close to 0 for nodes far from the target (low matched prefix length). This result indicates that i) Kad is not using replica roots efficiently, and ii) the nodes closest to the target suffer the burden for most of the search requests.

This problem can be explained by two factors in Kad. First, a querying node sends SEARCH_REQ starting from the closest node to nodes far from the target, thus, the closest node would receive most of the requests. Secondly, due to

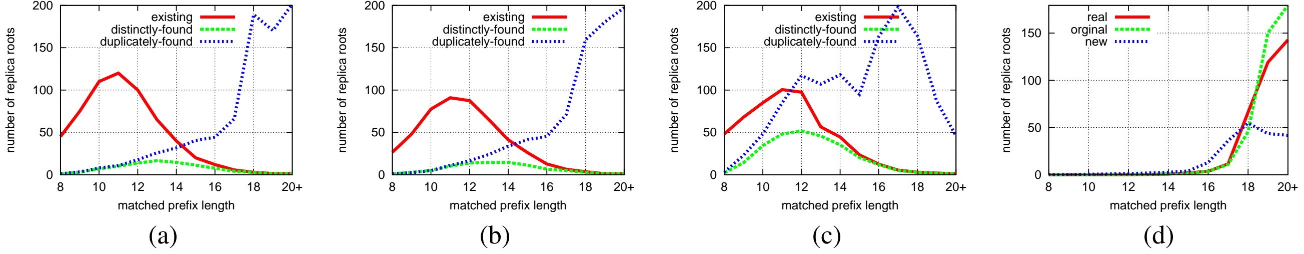


Figure 8. (a) Lookup with real popular objects (b) Original Kad lookup for our objects (c) New Kad lookup for our objects (d) Load for each prefix bit for real popular objects and our objects

the termination condition in Kad, the search stops if 300 results (objects) are received (recall that a replica root can return more than one result). Although there are more replica roots storing the binding information for a certain object, the search process stops without contacting these replica roots because 300 objects have been returned by the few replica roots contacted.

To address this load-balancing problem, we propose a new solution which satisfies the following requirements: i) balance the load for search lookups, and ii) produce a high search yield for both rare and popular objects. A description of the solution is as follows. A querying node attempts to retrieve the binding information starting far from the target ID. Suppose that querying node Q sends a KADEMLIA_REQ to node A, which is within the search-tolerance for target T. In addition to returning a list of peers (containing nodes closest to T that A knows about), A sends a piggybacked bit informing Q whether it has binding information for T, that is, whether A is a replica root for T. If A sends such a bit, Q then sends a SEARCH_REQ with a list of keywords to A and the latter returns any binding of objects matching all the keywords. When many replica roots publish popular objects, Q has a chance to retrieve enough bindings from replica roots that are not close to T. Thus, Q does not have to contact replica roots close to the target. This lookup can reduce the load on the closest nodes to a target with only a 1-bit communication overhead.

To exploit the new lookup solution, it is important to decide where to publish objects, that is, which nodes will be replica roots. Some nodes very close to a target ID should clearly be replica roots. This guarantees a high search yield even if only a small number of nodes publish the same objects ("rare" objects) because the closest nodes are almost always found as we have previously shown. Moreover, it is desirable that nodes far from the target be replica roots so that they can provide binding information earlier in the lookup process. This lessens the burden on the load for the closest replica roots and provides a shorter GET delay to querying nodes. In the new PUT operation, a publishing peer locates most of the closest nodes using *Fix2* and obtains a node index by sorting these nodes based on their distance to a target ID. The publishing node then sends the i-th closest node a PUBLISH_REQ with probability $p = \frac{1}{i-4}$.This heuristic guarantees that objects are published to the five closest nodes and to nodes further from the target.

We implemented our proposed solution and ran experiments to determine if it met our requirements for both PUT and GET. The same experiments from Section 3 were performed with the new solution. We repeated the experiments changing the number of files to be published, but we only present experiment results similar to those of the real network when the original Kad lookups were used. We observed a search success ratio of 62% for rare objects and almost 100% for popular objects. We next looked at whether our algorithm mitigated the load-balancing problem or not. In the experiment, 500 nodes published about 2150 different files with the same keyword, and another 500 nodes searched those files with that keyword. The experiments were repeated with 50 different keywords.

To show that our experiments emulated real popular objects in Kad, we tested both the original Kad lookup algorithm and our solution for comparison. In Figure 8(d), the "original" line show the results obtained from using the original Kad algorithm. As expected, these results were similar to what we obtained from the real network. The number of replica roots located by our proposed Kad lookup solution is shown in Figure 8(c). More replica roots were found (both the "duplicately-found" and "distinctly-found" lines) farther from a target than for the original Kad lookup. At 11 matched prefix length, 48 out of 101 replica roots were located using our solution while only 10 out of 91 replica roots were located using the original algorithm. The "new" line in Figure 8(d) shows that the load was shared more evenly across all the replica roots for our solution. At more than or equal to 20 matched prefix bit, the load decreased by 22%. In summary, our experimental results show that the proposed solution guarantees a high search yield for both rare and popular objects, and can further mitigate the load balancing problem in lookups for popular objects.

7 Related Work

Kad is a DHT based on the Kademlia protocol [6] that uses a different lookup strategy than other DHTs such as Chord [11] and Pastry [7]. The main difference between

Chord and Kademlia is that Chord has a root for every key (node ID). When a querying node finds that root, it can locate most of the replica roots. Every node keeps track of its next closest node (successor). In Pastry [7], each node has an ID and the node with the ID numerically closest to the key is in charge. Since each node also keeps track of its neighbors, once the closest node is found, the other replica roots can also be found. Thus, Chord and Pastry do not suffer from the same problems as Kad. We note that just replacing the Kad algorithm with Chord or Pastry is not a suitable solution as Kad contains some intrinsic properties, inherited from Kademlia, that neither Chord nor Pastry possesses – for example, Kad IDs are symmetric whereas Chord IDs are not. The Pastry algorithm can return nodes far from the target due to the switch in distance metrics. Moreover, Kad is widely used by over 1.5 million concurrent users whereas it was never shown that Chord or Pastry can work on large-scale networks.

Since Kad is one of the largest deployed P2P networks, several studies have measured various properties and features of the Kad network. Steiner et al [8, 10] crawled the whole Kad network, estimated the network size, and showed the distribution of node IDs over the Kad key space. More recently in [9], the authors analyzed the Kad lookup latency and proposed changing the configuration parameters (timeout, α, β) to improve the latency. Our work differs in that we measured the lookup performance in terms of reliability and load-balancing, and identified some fundamental causes of the poor performance.

Stutzbach et al. [12] and Falkner et al. [2] studied networks based on the Kademlia DHT algorithm by using eMule and Azureus clients, respectively. They argued that the lookup inconsistency problem is caused by churn and slow routing table convergence. However, our detailed analysis on lookups clearly shows that the lookup inconsistency problem is caused by the lookup algorithm which cannot consider duplicate returns from nodes with consistent views in the routing tables. Furthermore, the authors proposed changing the number of replica roots as a solution. Our experiments indicate that just increasing the replication factor is not an efficient solution. We propose two incrementally-deployable algorithms which significantly improve the lookup performance, and a solution to mitigate the load-balancing problem. Thus, prior work on the lookup inconsistency is incomplete and limited.

Freedman et al. [3] considered the problems in DHTs (Kad included) due to non-transitivity in the Internet. However, non-transitivity will only impact the lookup performance in a small way since, in essence, it can be considered a form of churn in the network. We already accounted for churn in our analysis and showed that churn is only a minor factor in the poor Kad lookup performance.

8 Conclusion

Should node churn be blamed as the root cause of poor lookup performance for file sharing? In this paper we examined why the Kad network exhibits poor performance during search and publish operations. The poor performance comes from the fact that the Kad network works too well in some sense. As we have shown, the level of similarity among nodes' routing tables in the Kad network is much higher than expected. Because of this high level of consistency, many of the same duplicated peers are returned during lookups. Thus, during a search, the number of unique nodes found close to a target ID is very limited. We have also observed that Kad suffers from a load-balancing problem during lookups for popular objects. Our proposed algorithms significantly improve Kad lookup performance while simultaneously balancing lookup load. Our solutions are completely compatible with existing Kad clients and thus incrementally deployable.

Acknowledgments. This work was funded by the NSF under grant CNS-0716025. We thank Peng Wang and James Tyra for the discussion on the performance of Kad in the early phase of the paper.

References

[1] Azureus. http://azureus.sourceforge.net.

[2] J. Falkner, M. Piatek, J. John, A. Krishnamurthy, and T. Anderson. Profiling a Million User DHT. In *IMC*, 2007.

[3] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-Transitive Connectivity and DHTs. In *USENIX WORLDS*, 2005.

[4] H. J. Kang, E. Chan-Tin, N. Hopper, and Y. Kim. Why Kad Lookup Fails. Technical Report 09-019, University of Minnesota, 2009.

[5] Mainline. http://www.bittorrent.com.

[6] P. Maymounkov and D. Mazíeres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *IPTPS*, 2001.

[7] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[8] M. Steiner, E. W. Biersack, and T. En-Najjary. Actively Monitoring Peers in KAD. In *IPTPS*, 2007.

[9] M. Steiner, D. Carra, and E. W. Biersack. Faster Content Access in KAD. In *IPTPS*, 2008.

[10] M. Steiner, T. En-Najjary, and E. W. Biersack. A Global View of Kad. In *IMC*, New York, NY, USA, 2007. ACM.

[11] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.

[12] D. Stutzbach and R. Rejaie. Improving Lookup Performance Over a Widely-Deployed DHT. In *INFOCOM*, 2006.

[13] D. Stutzbach and R. Rejaie. Understanding Churn in Peer-to-Peer Networks. In *IMC*, 2006.

[14] The Pirate Bay. http://thepiratebay.org.