Sub-Second Lookups on a Large-Scale Kademlia-Based Overlay

Raul Jimenez, Flutra Osmani and Björn Knutsson
KTH Royal Institute of Technology
School of Information and Communication Technology
Telecommunication Systems Laboratory (TSLab)
{rauljc, flutrao, bkn}@kth.se

***Abstract*—Previous studies of large-scale (multimillion node) Kademlia-based DHTs have shown poor performance, measured in seconds; in contrast to the far more optimistic results from theoretical analysis, simulations and testbeds.**

In this paper, we unexpectedly find that in the Mainline BitTorrent DHT (MDHT), probably the largest DHT overlay on the Internet, many lookups already yield results in less than a second, albeit not consistently. With our backwards-compatible modifications, we show that not only can we reduce median latencies to between 100 and 200 ms, but also consistently achieve sub-second lookups.

These results suggest that it is possible to deploy latency-sensitive applications on top of large-scale DHT overlays on the Internet, contrary to what some might have concluded based on previous results reported in the literature.

I. Introduction

Over the years, distributed hash tables (DHTs) have been extensively studied, but it is only in the last few years that multimillion node DHT overlays have been deployed on the Internet. To our knowledge, only three DHT overlays (all of them based on Kademlia [1]) consist of more than one million nodes: Mainline DHT (MDHT), Azureus DHT (ADHT), and KAD. The first two are independently used as trackers (peer discovery mechanisms) by BitTorrent [2], while KAD is used both for content search and peer discovery in eMule (a widely used file-sharing application).

KAD has been thoroughly studied [3], [4], [5]. Stutzbach and Rejaie [3] reduced median lookup latency to approximately 2 seconds (with a few lookups taking up to 70 seconds). Steiner et al. [4] focused solely on lookup parameters, providing useful correlations between parameter values and lookup latency. Their modest achieved lookup performance (lowest median lookup latency at 1.5 seconds), authors discovered, was not due to shortcomings in Kademlia or the KAD protocol but due to limitations in eMule's software architecture.

In this paper, we focus on Mainline DHT which, with up to 9.5 million nodes[1], is probably the largest DHT overlay ever deployed on the Internet [6]. In 2007, a study of the then most popular node implementation in MDHT reported median lookup latencies around one minute [7] and, to our knowledge, no systematic attempts to improve lookup performance have been reported.

Lookup latency results for MDHT, KAD and ADHT [7], [8] are disappointing considering the rather promising latency figures —in the order of milliseconds— previously reported in studies using simulators and testbeds [9], [10]. Our main goal is to improve lookup performance in MDHT, thus closing the gap between simulators and real-world deployments.

To measure and compare performance, we developed a profiling toolkit able to measure different node properties (lookup performance and cost among others) by parsing the network traffic generated during our experiments. This toolkit is capable of profiling any MDHT node, including closed-source, without the need of its instrumentation.

We profiled the closed-source μTorrent (also called UTorrent) implementation, currently the most prevalent MDHT implementation with 60% of the nodes in the overlay. Our results show that UTorrent's performance is unexpectedly good, with median lookup latencies well under one second.

UTorrent's performance does not, however, fulfill the demands of latency-sensitive applications such as the system that motivated this work (see Section II) because more than a quarter of its lookups take over a second. Thus, in an attempt to further reduce lookup latency, we developed our own MDHT node implementations.

In this paper, we show that our best node implementations achieve median lookup latencies below 200 ms and sub-second latencies in almost every single lookup, meeting our system's latency requirements.

The rest of the paper is organized as follows. The background is presented in Section II. Section III introduces the profiling toolkit. Section IV describes our MDHT node implementations while Section V discusses routing modifications. Section VI presents the experimental setup, Sections VII and VIII report the results obtained, Section IX briefly presents related work, and Section X concludes.

II. Background

The work presented in this paper is part of the P2P-Next project[2]. This project's main aim is to build a fully-distributed content distribution system capable of streaming live and on-demand video. Unlike file sharing applications, this is an interactive application, and thus reducing perceived latency

[1] A real-time estimation is available at http://dsn.tm.uni-karlsruhe.de/english/2936.php (June 2011)

[2] http://p2p-next.org/ (June 2011)

(e.g., the time it takes to start playback of a video after the user selects it) to a level acceptable by users is of great importance [11].

This system uses BitTorrent [2] as transport protocol and a DHT-based mechanism to find *BitTorrent peers* in a *swarm*. To avoid confusion, the following terms are defined here: *BitTorrent peers* (or simply *peers*) are entities exchanging data using the BitTorrent protocol; a *swarm* is a set of peers participating in the distribution of a given piece of content; and *DHT nodes* (or *nodes* for short) are entities participating in the DHT overlay and whose main task is to keep a list of peers for each swarm.

Our work is not, however, restricted to BitTorrent or video delivery. One can imagine more demanding systems, for instance, a DHT-based web service capable of returning services (e.g. a web page) quickly and frequently. CoralCDN [12] is a good example of such a service, although its scale is much smaller.

Our hope is that our results will encourage researchers and developers to deploy new large-scale DHT-based applications on the Internet.

A. Kademlia

Kademlia [1] belongs to the class of prefix-matching DHTs, which also includes other DHTs like Tapestry [13] and Pastry [14].

In Kademlia, each node and object are assigned a unique identifier from the 160-bit key space, respectively known as *nodeID* and *objectID*. Pairs of (*objectID*, *value*) are stored on nodes whose *nodeID* are closest to the *objectID*, where closeness is determined by performing an XOR bit-wise operation. In BitTorrent, an *objectID* is a swarm identifier (called *infohash*) and a *value* is a list of peers participating in a swarm.

A lookup traverses a number of nodes in the DHT overlay, each hop progressing closer to the target *objectID*. Each node maintains a tree-based routing table, containing $O(log\ n)$ *contacts* (references to nodes in the overlay), such that the total number of lookup hops does not exceed $O(log\ n)$, where n is the network size. The routing table is organized in *buckets*, where each bucket contains up to *k* contacts sharing some common prefix with the routing table's owner. Each contact in the bucket is represented by the triple *(nodeID, IP address, port)*.

New nodes are discovered opportunistically and inserted into appropriate buckets as a side effect of incoming queries and outgoing messages. To prevent stale entries in the routing table, Kademlia replaces stale contacts —nodes that have been idle for longer than a predefined period of time and fail to reply to active pings— with newly discovered nodes.

To locate nodes close to a given *objectID*, the node performing the lookup uses iterative lookup from start to finish. This node queries nodes from its routing table whose identifiers have shorter XOR distances to the *objectID*, and waits for responses. The newly discovered nodes —included in the responses— are then queried during the next lookup step. Kademlia makes use of parallel routing to send several parallel lookup requests, in order to decrease latency and the impact of timeouts. Lookup terminates when the closest nodes to the target are located.

B. Improving Lookup Performance

Given Kademlia's iterative lookup, lookup performance can be greatly enhanced by modifying the initiating node alone, without the need of changing any other node in the overlay. Thus, modified nodes can be deployed at any moment, setting the path for experimentation and incremental deployment of "better", yet backward-compatible, node implementations.

Researchers have proposed various approaches to increase overall lookup performance in iterative DHTs, while keeping costs relatively low. Parallel lookups and multiple replicas are two parameters that have often been fine-tuned to reduce the probability of lookup failures and alleviate the problem of stale contacts in routing tables, which in turn, increase DHT performance. Various bucket sizes, various-length prefix matching (known as symbol size) and reduced —usually RTT-based— timeout values have also been investigated as means of improving the overall performance.

We discuss some of these improvements in detail in Section IV and V, where we present the modifications we have deployed and measured.

III. Profiling MDHT Nodes

In a DHT overlay, nodes are independent entities that collaborate with each other in order to build a distributed service. A DHT protocol defines the interaction between nodes, but provides significant latitude in how to implement it. Indeed, the Mainline DHT protocol specification [15] leaves many blanks for the implementer to fill in as best as he can.

It follows naturally that many different node implementations will coexist in the MDHT overlay. Some, developed by commercial entities (e.g., Mainline and UTorrent), others cooperatively as open source projects (e.g., Transmission and KTorrent). Even though they have been developed to coexist, significant differences in their behavior can be observed, parts just accidents of separate development, others the result of making different trade-offs.

From our initial studies of Mainline DHT, we had observed diversity in the existing MDHT node implementations. We also recognized that our efforts to improve the performance of MDHT nodes would likely make use of the latitude afforded by the protocol specifications, and thus it was of critical importance that we be able to study the impact of our modifications. To this end, we built a toolkit for profiling and analyzing the behavior and performance of MDHT nodes.

A. Profiling Tools

Instrumenting an open source DHT node is a common approach to measure its performance. The instrumented node would join an overlay, perform lookups, and log performance measurements.

It is, however, unpractical to instrument nodes whose source code is unavailable. In MDHT, UTorrent is by far the most

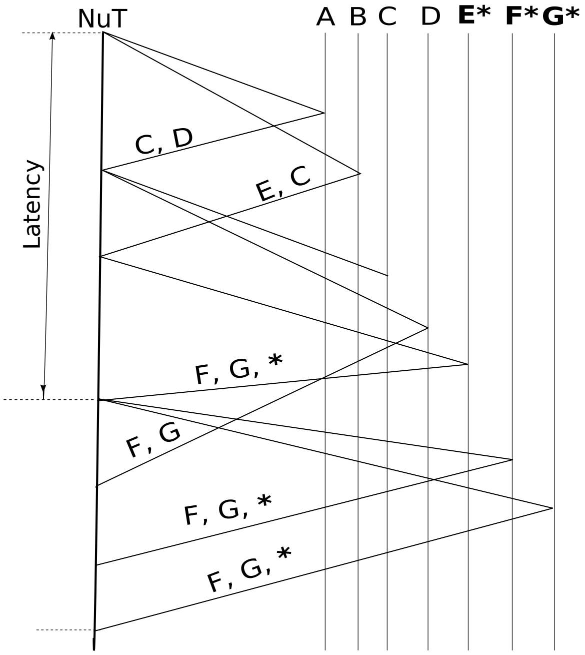

Fig. 1. Lookup performed by node under test (NuT). Letters A–G represent nodes in the overlay. Values are represented with "*".

popular node implementation (2.7 out of 4.4 million nodes according to our results presented in Section VIII). Given that UTorrent's source code is closed, we devised a different approach.

Our toolkit uses a black-box approach: an MDHT node is commanded to perform lookup operations (by using the node's GUI or API), while simultaneously capturing its network traffic. Figure 1 illustrates a lookup performed by the *node under test* (NuT). This node joins the MDHT overlay and is under our control (we can command it to perform lookups); the rest of the nodes shown in the figure (A–G) are a minute fraction of the millions of MDHT nodes —over which we have no control.

Whenever NuT sends or receives a message, the data packet is captured. When the experiment is over, all captured packets can be parsed to measure lookup latency and cost, among other properties.

The toolkit's core, written in Python, provides modules to read network captures and decode MDHT messages. Tools to analyze and manipulate messages are also available, as are the plotting modules that can produce various graphs. Along with the rest of the software presented in this paper, we have released the profiling toolkit under the GNU LGPL 2.1 license.

We use the toolkit in Sections VII and VIII to illustrate our results, and as will be seen, it is already capable of measuring and displaying many interesting properties of MDHT nodes. New measurements and presentations are easily added as plug-ins, using existing analysis and presentation plug-ins as templates.

B. Profiling Metrics

In theoretical analysis and simulations of DHTs, lookup performance is often measured in routing hops between the initiator —node performing the lookup— and the node closest to the target key. Although our profiling toolkit can measure hops, we find it more appropriate to measure lookup latency because that is the parameter determining whether a DHT is suitable for latency-sensitive applications.

In this paper, we define **lookup latency** as the time elapsed between the first lookup query is sent and the first response containing values is received.

Figure 1 illustrates a full lookup performed by the node under test. NuT starts the lookup by selecting the nodes closest to the target in its routing table (A and B) and sending lookup queries to them. NuT receives a response from A containing nodes (C and D) closer to the target but no values. Then, NuT sends queries to nodes C and D. The lookup continues until NuT receives a response containing values (*) from E. At this point we consider the goal achieved and we record the lookup latency, although the lookup can progress further to obtain more values associated with the key, as we will discuss in Section VIII.

We define **lookup cost** as the number of lookup queries sent before receiving a value (including queries sent but not yet replied). In the example above, lookup cost is five queries. At the time values are retrieved (E's response contains values), three queries were replied (A, B and E), D replies shortly after, and C never replies (this query would eventually trigger a timeout).

Finally, we define **maintenance cost** as the total number of maintenance queries —`ping` and `find_node` messages— sent by the node under test. These queries are sent to detect stale entries in the routing table and find replacements for these entries. As we later propose modifications to the routing table management, which is the source of maintenance traffic, we will also measure their impact on maintenance cost.

IV. Implementing MDHT Nodes

We have developed a flexible framework based on a plug-in architecture, capable of creating different MDHT nodes. The central part of the architecture handles the interaction between network, API, and plug-ins; while the plug-ins contain the actual policy implementation. There are two categories of plug-in modules: *routing modules* and *lookup modules*. The policies are concentrated on these plug-ins (e.g., all algorithms and parameters related to routing table management are exclusively located in routing modules), simplifying their modification. This architecture allows us to quickly implement different routing table and lookup configurations, and compare them against each other.

The combination of the core, a routing module and a lookup module forms a fully-functional MDHT node, which can be deployed and further analyzed with the profiling tools described in Section III-A.

Although this paper examines only two lookup and four routing modules, several additional modules have been designed, implemented and measured. The modules presented here have been chosen due to their characteristics and their effects on lookup performance and cost.

Even though our plug-in architecture allows us to freely modify lookup modules, we observe that merely adjusting well-know lookup parameters can dramatically improve lookup performance.

These lookup parameters are known as α and β. The α parameter determines how many lookup queries are sent in parallel at the beginning of the lookup, while β is the number of maximum queries sent when a response is received. Figure 1 is an example of a lookup with both parameters set to two.

In this paper, we describe and measure two lookup modules:

- **Standard Lookup** Since the protocol specification does not specify parameters such as α and timeout values, we have resorted to an analysis of UTorrent's lookup behavior. According to our observations, UTorrent's value for α and β are four and one, respectively. Our *standard lookup* implements the same parameters.
- **Aggressive Lookup** In our *aggressive lookup* module, β is set to three while α remains four.

Our routing modules introduce much deeper modifications to the original MDHT routing table management specified in BEP5 (BitTorrent Enhancement Proposal 5) [15]. These modifications are detailed in the next section.

V. Routing Modules

Although some of the previous studies on Kademlia performance have considered modifications on routing table management, most of them estimate the performance gain assuming that all nodes implement them.

We do not propose global modifications where all nodes in the overlay must be modified to obtain benefits. Instead, we propose modifications that benefit the nodes implementing them, regardless of whether other nodes in the overlay implement these modifications or not.

To our knowledge, this is the first attempt to deploy alternative routing table management implementations on an existing multimillion overlay on the Internet, and then measure their effects on lookup latency.

A. Standard Routing Table Management (BEP5)

The *BEP5* routing module aims to implement the routing table management specified in the BEP5 specifications [15] as rigorously as possible. The specifications define bucket size k to be 8. The routing table management mechanism is summarized next.

When a message is received, query or response, the appropriate bucket is updated. If there is already an entry corresponding to this node in the bucket, the entry is updated. Otherwise, three scenarios are possible: (1) if the bucket is full of *good* nodes, the new node is simply discarded; (2) if there is a *bad* node inside the bucket, the new node simply replaces it; (3) if there are *questionable* nodes inside the bucket, they are pinged; if any of them fail to respond after two ping attempts, they will be replaced.

According to the specifications, nodes are defined as *good* nodes if they respond to queries or they have been seen alive in the last 15 minutes. Nodes which have not been seen alive in the last 15 minutes become *questionable*. Nodes that failed to respond to multiple consecutive queries (we chose this value to be two) are defined as *bad* nodes.

Buckets are usually kept fresh as a side effect of lookup traffic. Buckets which have not been opportunistically refreshed in the last 15 minutes are refreshed by performing a maintenance lookup. Maintenance lookups are similar to normal lookups but they use `find_node` messages instead of `get_peers`.

B. Nice Routing Table Management (NICE)

The *NICE* routing module attempts to improve the quality of the routing table by continuously refreshing nodes in the routing table and checking their connectivity. While, as our results show, this quality improvement directly reduces our nodes' lookup latency, we expect other nodes to be also benefited as a side effect. We plan to measure this indirect benefit in future work.

The refresh task is regularly triggered (every 6 seconds in NICE). Each time it is triggered, it selects a bucket and pings the most stale node in the bucket. This continuous refresh guarantees that each bucket must have at least one contact that was recently refreshed and no contacts that have not been refreshed for more than 15 minutes. As a side benefit, this makes maintenance traffic smooth and predictable, with a maximum maintenance traffic of 10 queries per minute.

This module also actively probes nodes to detect and remove nodes with connectivity issues from the routing table. In particular, we implement the quarantine mechanism we previously proposed [16] where nodes are only added to the routing table after a 3 minute period. This quarantine period is mainly aimed at detecting DHT nodes with limited connectivity (probably caused by nodes behind NAT and firewall devices) which cause widespread connectivity artifacts in Mainline DHT, hindering performance.

C. NICE + Low-RTT Bias (NRTT)

In Kademlia, any node falling within the region covered by a bucket is eligible to be added to that bucket. Kademlia follows a simple but powerful strategy of preferring nodes that are already in the bucket over newly discovered candidates. The reasoning is that this policy leads to more stable routing tables [1].

Having stable contacts in the routing table benefits lookups by reducing the probability of sending lookup queries to nodes that are no longer available. Likewise, if the round trip time (RTT) to these nodes is low, then the corresponding lookup queries will be quickly responded, reducing lookup latency.

The impact of low-RTT bias in routing tables has been previously discussed [17], [10] but never deployed on a large-scale overlay.

The *NRTT* module is an implementation of the NICE module plus low-RTT bias. While NICE follows Kademlia's rules regarding node replacement —i.e. nodes cannot be replaced unless they fail to respond to queries— NRTT introduces the possibility of replacing an existing node with a recently discovered node, if the RTT of the incoming node is lower than that of the existing node.

D. NRTT + 128-bucket (NR128)

Another approach to improve performance is to reduce the number of lookup hops. The most extensive study of bucket modifications in Kademlia [3] considered two options: (1) adding more buckets to the routing table and (2) enlarging existing buckets. Their theoretical analysis concluded that, while both approaches offer comparable hop reduction on average, increasing bucket size is simpler to implement, has lower maintenance cost, and improves resistance to churn as a side effect. Finally, they showed that performance improves logarithmically with bucket size.

Enlarging buckets is simple but costly because maintenance traffic grows linearly with bucket size. That is, if one is to enlarge all buckets equally. But not all the buckets are equal when it comes to lookup performance.

Given the structure of a Kademlia routing table, on average, the first bucket is used in half of the lookups, the second bucket in a quarter of the lookups, and so forth. In the *NR128* routing module, buckets are enlarged proportionally to the probability of them being used in a given lookup. The first buckets hold 128, 64, 32, and 16 nodes respectively, while the rest of the bucket sizes remain at 8 nodes. To our knowledge, this technique has not been proposed before.

The expected result is that, while half of the lookups are bootstrapped by a 128-bucket, and more than nine in ten by an enlarged bucket, maintenance traffic merely doubles compared to NICE (20 queries per minute).

VI. Experimental Setup

To measure and understand the behavior of UTorrent and of our own implementations, we have run numerous experiments in a variety of configurations, both sequential and parallel. Our final configuration is one in which we ran all implementations in parallel, providing the same experimental conditions to all nodes being compared, on a large number of freshly acquired torrent *infohashes* (see below). The experiment we document in this paper is neither the best nor the worst but rather representative, as the results we obtained are very consistent between different runs.

The experiment, which started on March 26, 2011 and ran over 80 hours, tested all our eight (two times four) implementations and UTorrent version 2.2 (build 23703)[3]. A very simple coordination script is used to command our nodes under test; a Python interface is used for our MDHT node implementations and an HTTP interface is used for UTorrent.

Each node under test joins the multimillion-node MDHT overlay. Upon joining the overlay, the lookup rounds begin. In each round, a random NuT sequence is generated. Every 10 seconds, the next NuT in the sequence is commanded to perform a lookup on an identifier that is randomly selected from its list of infohashes and then remove the infohash from its list. All nine NuTs perform one lookup per round, until every NuT has emptied its list of infohashes —i.e. when every NuT has performed a lookup for every infohash. The experiment is then considered complete and the captured traffic is ready to be parsed by our profiling toolkit.

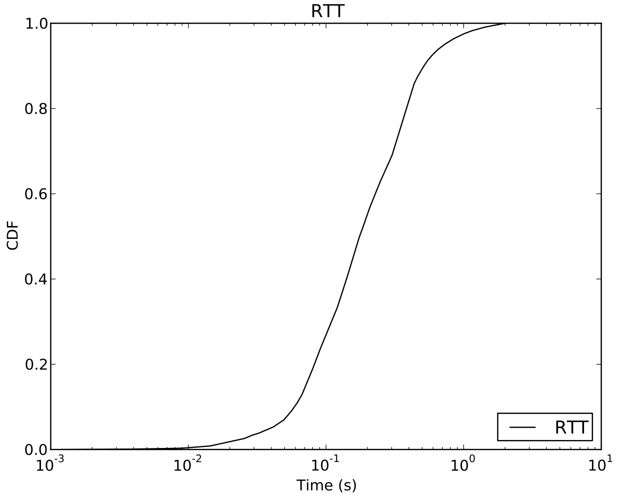

Fig. 2. RTT to nodes in the MDHT overlay (Queries that have not been replied within 2 seconds are considered timed-out, thus excluded from this graph.)

Experiments were not CPU-bound, and were run on a system with a P4@3.0GHz, 3GB RAM and running Windows 7. Regarding network latency, as shown in Figure 2, the RTTs from the nodes under test to other MDHT nodes were mainly concentrated between 100 and 300 ms, with very few RTTs over one second (2nd percentile: 2.13 ms, 25th percentile: 94.8 ms, median: 175.2 ms, 75th percentile: 343.6 ms, 98th percentile: 1093.9 ms).

Infohashes can be obtained from various sources, and can even be generated by us. We are, however, specifically interested in active swarms under “real world” conditions. This has led us to obtain infohashes from one of the most popular BitTorrent sites on the Internet, `thepiratebay.org`. This site has a “top” page with the most popular content organized in categories. We have extracted all infohashes from these categories, obtaining a total of 3078 infohashes.

It should be noted that our MDHT node implementations neither download, nor offer for upload, any content associated with these infohashes. UTorrent is given only 3 seconds to initiate the download —triggering a DHT lookup as a side-effect— before being instructed to stop its download, thus leaving no time for any data transfer. We have observed that the DHT lookup progresses normally despite the stop command.

[3] Downloaded from http://www.utorrent.com/downloads/

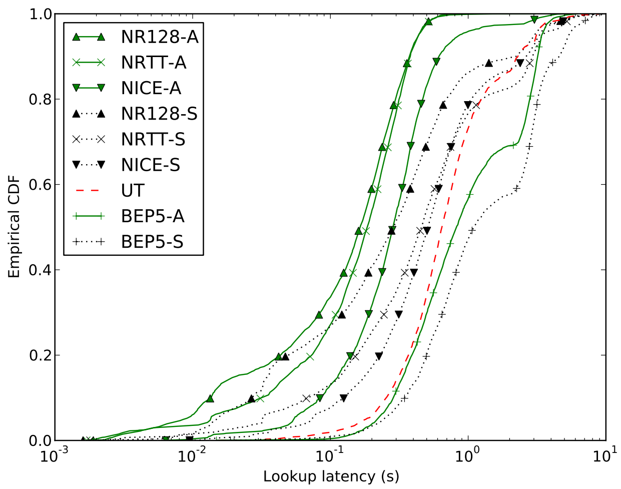

Fig. 3. Lookup latency when retrieving a value from the MDHT overlay

VII. Experimental Results

Each of the eight node implementations we have studied is one of the combinations of our four routing and two lookup modules described previously, and the names we use for them reflect the components combined. For instance, the NICE-S node implementation uses the NICE routing module plus the standard (S) lookup module. Similarly, we will use “wildcards”. For example, *-S indicates all nodes using the standard lookup module, and NICE-* all nodes using the NICE routing module. UTorrent is simply referred to as UT.

A. Lookup Latency

Figure 3 shows the empirical cumulative distribution function (CDF) of lookup latency when retrieving a value from the overlay, as defined in Section III-B.

In Table I we document lookup latencies for the 50 (median), 75, 98, and 99 percentiles. We expect these figures to be valuable for those interested in large-scale latency-sensitive systems, whose requirements usually specify a maximum latency for a large fraction of the operations.

Since our BEP5 routing module follows the MDHT specifications published by the creators of UTorrent (BitTorrent, Inc.) and our standard module implements the same lookup parameters as UTorrent, we expected that BEP5-S would perform similarly to UTorrent. As our measurements reveal, this is not the case. UT performs significantly better than our BEP5-S, and even outperforms our BEP5-A, which we expected to beat UT by using more aggressive lookups. This fact suggests undocumented enhancements in UTorrent’s routing table management.

We see that the aggressive lookup module consistently yielded lower median lookup latency, but more importantly, drastically reduced the worst-case latencies, as seen in the 98th and 99th percentile columns of Table I.

Nodes implementing the NICE routing module perform better than BEP5 and also UTorrent. We believe that this is due to an improvement in the quality of the initiator’s routing table caused by our constant refresh strategy and mechanisms to detect and avoid nodes with connectivity limitations.

TABLE I
Lookup latency (in ms)

Node	median	75th p.	98th p.	99th p.
UT	647	1047	3736	5140
BEP5-S	1105	3011	6828	7540
NICE-S	510	877	4468	5488
NRTT-S	459	928	5060	5737
NR128-S	286	589	4375	5343
BEP5-A	825	2601	3840	4168
NICE-A	284	420	2619	3247
NRTT-A	185	291	512	566
NR128-A	164	269	506	566

The performance gain from the addition of low-RTT bias (NICE vs. NRTT) is uneven. NRTT-A performs significantly better than NICE-A, but using standard lookups, the difference is less pronounced. This is due to standard lookups not being able to take full advantage of low-RTT contacts by rapidly fanning out. The comparison between NRTT-S and NRTT-A illustrates this point, where the worst-case latency is an order of magnitude lower for NRTT-A.

When examining the routing table, we find that NICE-* nodes have contacts with RTTs in the 100–300 ms range, while NRTT-* nodes have contacts whose RTTs are lower than 20 ms.

Conversely, we see that the impact of enlarged routing tables in NR128-variants is the opposite to that in NRTT. The small performance gain from NRTT-A to NR128-A may be a sign that the maximum performance has been reached already. Indeed, NR128-A’s median lookup latency is, in fact, lower than the median RTT to MDHT nodes.

NR128-A, our best performing node implementation, achieves a median lookup latency of 164 ms. While median lookup latency is important, many latency-sensitive applications are more concerned with the worst-case performance, and treat lookup latency above a narrow threshold as failure.

Where previous measurements of large-scale Kademlia-based overlays report long tails with worst-case latencies in the tens of seconds, our NRTT-A and NR128-A consistently achieve sub-second lookups, with almost 98% finishing in less than 500 ms. More importantly, assuming a hard lookup deadline of 1 second, less than five out of over three thousand lookups would fail using any of these two implementations.

B. Lookup Cost

Lookup cost, defined in Section III-B, is also an important characteristic to measure. As Figure 4 shows, implementations using the aggressive lookup module require more lookup queries, thus increasing the lookup cost.

Lookup cost in UTorrent and our *-S nodes are very similar, as we expected. Among them, NRTT-S is slightly more expensive than the rest, which is caused by a more intensive query burst, due to the fan-out effect discussed in the previous section. Conversely, NR128-S has the lowest lookup

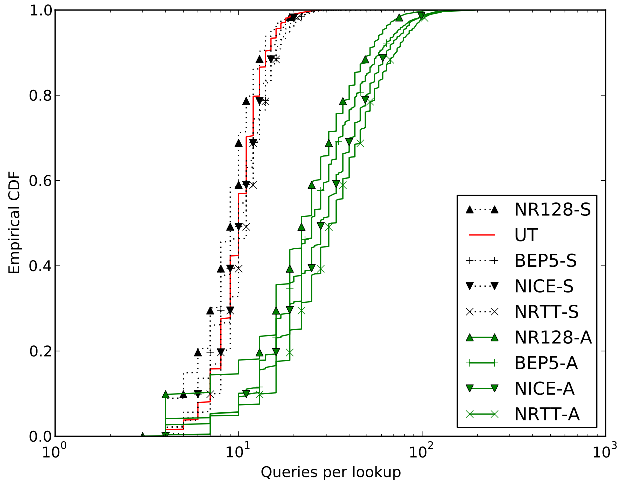

Fig. 4. Lookup cost. (We recognize that distinguishing between individual lines in this graph is hard, but the difference between standard (*-S and UT) and aggressive (*-A) lookup is clear. Also notice the tendency of NR128-* and NRTT-* to have lower and higher cost than the rest, respectively.)

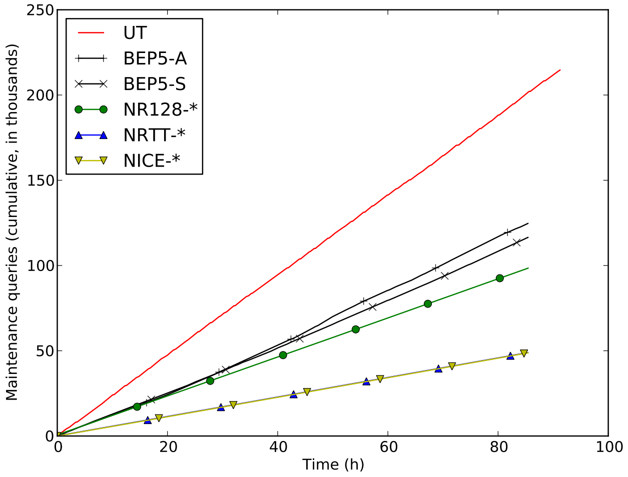

Fig. 5. Cumulative maintenance traffic during the entire experiment

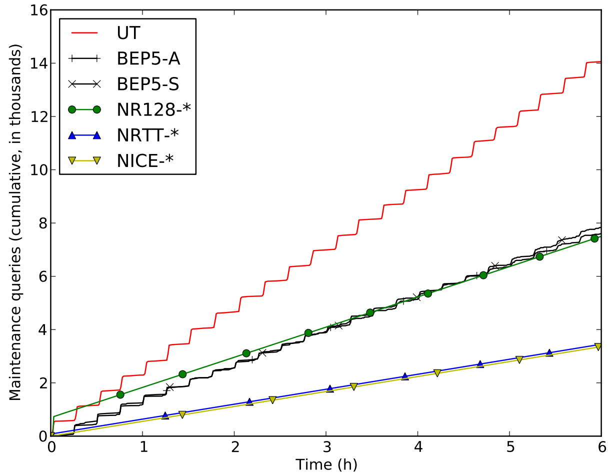

Fig. 6. Cumulative maintenance traffic during the first 6 hours

cost, which comes as no surprise since its enlarged buckets will reduce the number of hops required.

We predictably see similar relationships between the *-A nodes, but with higher average lookup costs across the board.

C. Maintenance Cost

Figure 5 depicts the cumulative maintenance queries sent over time. The obtained results confirm that all our MDHT node implementations generate less maintenance traffic, by a considerable margin, than UTorrent.

As mentioned earlier, we believe that UTorrent's unexpectedly good lookup performance is due to modifications to its routing table management, compared to the specification. This would go a long way towards explaining why we observe much more maintenance traffic than for our BEP5-* implementations.

Figure 6 shows only the first 6 hours of the experiment, revealing a peculiar stair-like pattern in UT and BEP5-*. Every 15 minutes, UTorrent triggers a burst of maintenance messages, approximately 600 messages for a period of 1–2 minutes, and few or no queries between bursts. We see a similar pattern initially in our own BEP5-* implementations, but they quickly flatten out. This observation suggests that while the initial occurrence is an artifact of the specification, the continued behavior is due to the way UTorrent implements its internal synchronization mechanism, causing maintenance message bursts.

All our MDHT node implementations, regardless of the modifications they include, drastically reduce maintenance traffic compared to UTorrent. BEP5-S and BEP5-A have irregular maintenance traffic patterns while the rest were designed to have very regular traffic patterns.

The enlarged bucket implementations (NR128-*) generate twice the maintenance traffic of NICE-* and NRTT-* (whose lines overlap), but still generate less traffic than BEP5-* in the long run.

D. Trade-offs

In comparing our different implementations, we have explored different trade-offs between performance and cost. We do, however, also see that some benefits can be gained at zero, or even negative, cost. For instance, NR128-S is better than UTorrent in all aspects, with significantly lower maintenance cost, lower lookup cost and median lookup latencies less than 50% of UTorrent's. Both NR128-S and UTorrent suffer from long tails, however, with 10% and 14%, respectively, of the lookups taking more than 2 seconds.

While achieving better performance at lower cost is certainly desirable, our target applications have very strict latency requirements. We are thus forced to go a step further, and

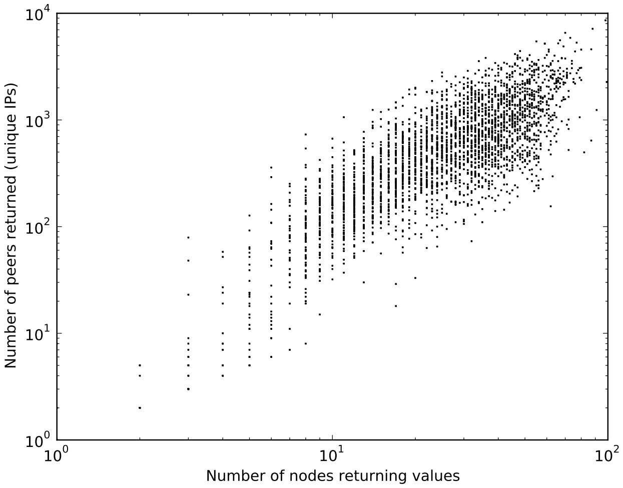

Fig. 7. Peers versus nodes returning values

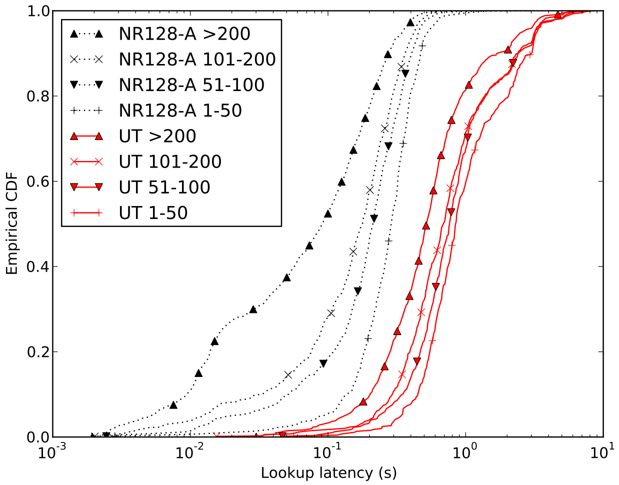

Fig. 8. Lookup latency versus swarm size for UTorrent and NR128-A. Both implementations perform better on larger swarms.

carefully explore what trade-offs we can make to meet these requirements, even at sometimes significantly higher costs.

Specifically, our low-RTT bias nodes (NRTT-*) achieve a noticeable performance improvement while keeping the same maintenance cost and just a small increase in lookup cost, which we attribute to being able to more rapidly fan out queries, compared to NICE-*. Finally, enlarging buckets improves lookup performance while slightly reducing lookup cost, which might be suitable when lookup cost dominates over maintenance cost. For example, a system where lookups are performed very frequently.

VIII. Additional Results

Our primary goal was to reduce the time until our node under test received the first value, but since we have not changed the way lookups terminate, they will continue until they reach the node closest to the key. Our toolkit continued to capture information about this phase of the lookups as well.

The modifications we have made have implications not only for the first phase, analyzed in the previous section, but for the complete lookup. In this section, we will present our analysis and summarize our results as they apply to the whole lookup.

A. Lookup Latency Versus Swarm Size

In principle, in a Kademlia-based DHT, only a fixed number of nodes need to store the values corresponding to a given key, regardless of the size of DHT or the popularity of the key. In practice, we find that popular keys in MDHT tend to have values distributed among a large number of nodes, while less popular keys are less widely dispersed.

In Figure 7 we plot the number of nodes returning values (replicas) against number of unique values stored (swarm size). We see that as swarm size increases, the number of replicas found increases as well.

This has no impact on the time it takes to reach the node closest to the key, but has a significant impact on lookup latency, as defined by us. Not only because there are more replicas to find, but also because having more replicas increase the chance that at least one of them is close (low RTT) to the node under test.

We thus see a relationship between swarm size and lookup latency. Figure 8 illustrates that lookup latency is lower when looking up popular infohashes (large swarms). In NR128-A, for instance, median lookup latency for swarms with more than 200 peers is 92 ms versus 289 ms for swarms with 50 peers or less (521 ms vs. 848 ms in UTorrent).

We draw two conclusions from these observations. First, users should expect significantly lower latency when looking up popular keys (i.e., popular content). And second, our techniques yield medians well under half second even for small swarms.

B. Reaching the Closest Node

In this paper, we have focused on a more user-centric metric of DHT performance, the time to find values. Another metric that has been widely used and studied in DHTs is the time to reach the closest node to the target key [3]. For completeness, and to allow our results to be easily compared to previous work, we plot our results according to this metric in Figure 9.

Using this metric, our NR128-A implementation still achieves sub-second results, with a median of 455 ms and 92.8% of its lookups reaching the closest node within a second.

C. Queries & Responses

The Internet is a pretty hostile environment, and many issues that normally would not arise in a testbed or simulator will impede performance when the same code is deployed “in the wild”. As an example, Table II presents information about lookup traffic obtained in our experiments. The *queries* columns show the number of queries generated, and *responses* the responses received, both the absolute number and as

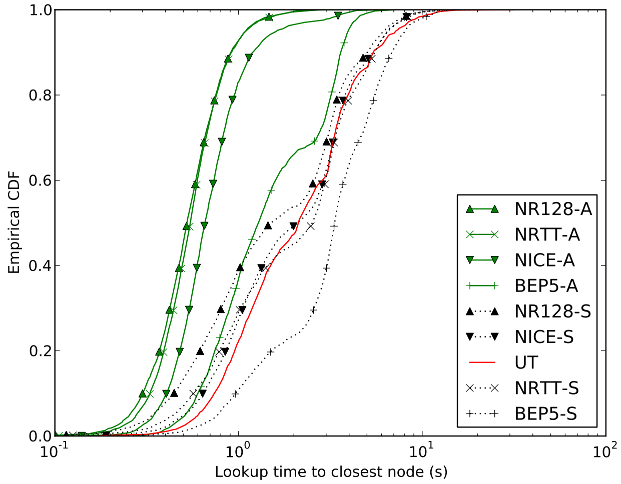

Fig. 9. Lookup latency to reach the closest node to the key

TABLE II
LOOKUP QUERIES AND RESPONSES

Label	Queries	Responses (%)
UT	92,450	52,378 (57)
BEP5-S	67,454	36,361 (54)
NICE-S	68,923	43,937 (64)
NRTT-S	70,234	44,515 (63)
NR128-S	64,488	39,633 (61)
BEP5-A	198,015	116,070 (59)
NICE-A	260,849	166,026 (64)
NRTT-A	281,335	183,175 (65)
NR128-A	221,543	140,025 (63)

TABLE III
IMPLEMENTATION MARKET SHARE

Implementation	Nodes (unique IPs)	Percentage
UT	2,663,538	60.0
LT	324,122	7.3
TR	7,666	0.2
Other versions	4,813	0.1
No version	1,441,899	32.5
Total	4,442,038	100.0

a percentage of queries issued, with the remainder being timeouts.

As can be seen, for BEP5-* and UT, less than 60% of queries receive responses, the equivalent, one could say, of more than a 40% packet loss ratio. In previous work [16], we characterized these connectivity artifacts and proposed mechanisms to identify and filter out nodes with connectivity issues. In this paper, some of these mechanisms have been implemented, improving the quality of our own routing tables. We believe that these improvements explain why NICE-*, NRTT-* and NR128-* consistently see a higher response rate than BEP5-* and UT. We plan to analyze the impact of these routing policies on the quality of routing tables in future work.

D. Implementation Market Share

Mainline DHT messages have an optional field where the sender can indicate its version in a four-character string. The first two characters indicate the client —UTorrent nodes identify themselves as "UT"— and the other two, the version number. The client labels reported by nodes are presented in Table III.

During the course of this experiment, the nodes under test have exchanged messages with over four million nodes (unique IP addresses) in the MDHT overlay. We have identified 2.6 out of 4.4 million (60%) as UTorrent nodes, far ahead of the second most common node implementation, libtorrent. It is also noteworthy that about one third of the nodes did not include this optional field in their messages.

IX. Related Work

Li et al. [18] simulated several DHTs under intensive churn and lookup workloads, in order to understand and compare the effects of different design properties and parameter values on performance and cost. The study revealed that, under intensive churn, Kademlia's capacity of performing parallel lookups reduces the effect of timeouts compared to other DHT designs studied. In their simulation results, Kademlia achieved a median lookup latency of 450 ms with the best parameter settings.

Kaune et al. [10] proposed a routing table with a bias towards geographically close nodes, called *proximity neighbour selection (PNS)*. Although their goal was to reduce inter-ISP traffic in Kademlia, they observed that PNS also reduced lookup latency in their simulations from 800 to 250 ms.

Other non-Kademlia-based systems have been studied. Rhea et al. [19] showed that an overlay deployed on 300 PlanetLab hosts can achieve low lookup latencies (median under 200 ms and 99th percentile under 400 ms). Dabek et al. [20] achieved median lookup latencies between 100–300 ms on an overlay with 180 test-bed hosts.

Crosby and Wallach [7] measured lookup performance in two Kademlia-based large-scale overlays on the Internet, reporting a median lookup latency of around one minute in Mainline DHT and two minutes in Azureus DHT. They argue that one of the causes of such performance is the existence of dead nodes (non-responding nodes) in routing tables combined with very long timeouts. Falkner et al. [8] reduced ADHT's median lookup latency from 127 to 13 seconds by increasing the lookup cost three-fold.

Stutzbach and Rejaie [3] modified eMule's implementation of KAD to increase lookup parallelism. Their experiments revealed that lookup cost increased considerably while lookup latency improved only slightly. Their best median lookup latency was around 2 seconds.

Steiner et al. [4] also tried to improve lookup performance by modifying eMule's lookup parameters. Although they discovered that eMule's design limited their modifications' impact, they achieved median lookup latencies of 1.5 seconds on the KAD overlay.

X. Conclusion

In this paper, we have shown that it is possible for a node participating in a multimillion-node Kademlia-based overlay to consistently perform sub-second lookups. We have also analyzed the impact of each proposed modification on performance, lookup cost, and maintenance cost, exposing the trade-offs involved. Additionally, we observed a phenomenon relevant for applications using the overlay: the more popular a key is, the faster the lookup.

In our efforts to accomplish the goal of supporting latency-sensitive applications using Mainline DHT, we have also produced other noteworthy secondary results, including, but not limited to: **(1)** a profiling toolkit that allows us to analyze MDHT messages exchanged between the node under study and other MDHT nodes, without code instrumentation; **(2)** the deployment and measurement of three modifications to routing table management (NICE, NRTT, NR128); and **(3)** an infrastructure to rapidly implement and deploy those modifications in the form of plug-ins.

Our initial study of MDHT node implementations revealed that UTorrent is the most common implementation currently in use, with a measured “market share” of 60%, making UTorrent a good candidate as the state-of-the-art benchmark for us to beat.

Our most aggressive implementation (NR128-A) not only beats UTorrent, but also steals its lunch money. Not only is our median lookup latency almost four times lower than UTorrent’s, but, most importantly for our purposes, just 0.1% of NR128-A’s lookups need over a second versus over 27% of UTorrent’s. While this comes at a higher lookup cost (220%), when we consider both lookup and maintenance traffic, our implementation actually generates substantially less traffic than UTorrent.

Amongst our less aggressive lookup implementations, NR128-S needs slightly less queries per lookup, half the maintenance traffic, and still its median lookup latency is less than half of UTorrent’s, beating it in all three metrics.

We hope that others will find our results useful in designing, evaluating, and improving applications deployed on top of large-scale DHT overlays on the Internet. All the source code described in this paper is available on-line at: http://people.kth.se/~rauljc/p2p11/.

Acknowledgment

The authors would like to thank Rebecca Hincks, Amir H. Payberah, and the anonymous reviewers for their valuable comments on our drafts.

The research leading to these results has received funding from the Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 216217 (P2P-Next).

References

[1] P. Maymounkov and D. Mazieres, “Kademlia: A peer-to-peer information system based on the XOR metric,” in *Proceedings of the 1st International Workshop on Peer-to Peer Systems (IPTPS02)*, 2002, pp. 53–65.

[2] B. Cohen, “Incentives Build Robustness in BitTorrent,” in *Workshop on Economics of Peer-to-Peer Systems*, vol. 6. Berkeley, CA, USA, 2003.

[3] D. Stutzbach and R. Rejaie, “Improving Lookup Performance Over a Widely-Deployed DHT,” in *INFOCOM*. IEEE, 2006.

[4] M. Steiner, D. Carra, and E. W. Biersack, “Evaluating and improving the content access in KAD,” *Springer ”Journal of Peer-to-Peer Networks and Applications”, Vol 2*, 2009.

[5] M. Steiner, T. En-Najjary, and E. W. Biersack, “A global view of kad,” in *IMC ’07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. New York, NY, USA: ACM, 2007, pp. 117–122.

[6] K. Junemann, P. Andelfinger, J. Dinger, and H. Hartenstein, “BitMON: A Tool for Automated Monitoring of the BitTorrent DHT,” in *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*. IEEE, 2010, pp. 1–2.

[7] S. A. Crosby and D. S. Wallach, “An analysis of bittorrent’s two kademlia-based dhts,” 2007.

[8] J. Falkner, M. Piatek, J. P. John, A. Krishnamurthy, and T. Anderson, “Profiling a million user DHT,” in *IMC ’07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. New York, NY, USA: ACM, 2007, pp. 129–134.

[9] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek, “Comparing the performance of distributed hash tables under churn,” in *In Proc. IPTPS*, 2004.

[10] S. Kaune, T. Lauinger, A. Kovacevic, and K. Pussep, “Embracing the peer next door: Proximity in kademlia,” in *Eighth International Conference on Peer-to-Peer Computing (P2P’08)*, 2008, p. 343–350.

[11] A. Bakker, R. Petrocco, M. Dale, J. Gerber, V. Grishchenko, D. Rabaioli, and J. Pouwelse, “Online video using bittorrent and html5 applied to wikipedia,” in *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, 8 2010, pp. 1 –2.

[12] M. J. Freedman, “Experiences with coralcdn: a five-year operational view,” in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI’10. Berkeley, CA, USA: USENIX Association, 2010, pp. 7–7. [Online]. Available: http://portal.acm.org/citation.cfm?id=1855711.1855718

[13] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz, “Tapestry: a resilient global-scale overlay for service deployment,” *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, 2004.

[14] A. Rowstron and P. Druschel, “P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” *In: Middleware*, pp. 329–350, 2001.

[15] A. Loewenstern, “BitTorrent Enhancement Proposal 5 (BEP5): DHT Protocol,” 2008.

[16] R. Jimenez, F. Osmani, and B. Knutsson, “Connectivity properties of Mainline BitTorrent DHT nodes,” in *9th International Conference on Peer-to-Peer Computing 2009*, Seattle, Washington, USA, 9 2009.

[17] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica, “The impact of DHT routing geometry on resilience and proximity,” in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 2003, pp. 381–394.

[18] J. Li, J. Stribling, R. Morris, M. F. Kaashoek, and T. M. Gil, “A performance vs. cost framework for evaluating DHT design tradeoffs under churn,” in *INFOCOM*, 2005, pp. 225–236.

[19] S. Rhea, B. Chun, J. Kubiatowicz, and S. Shenker, “Fixing the embarrassing slowness of OpenDHT on PlanetLab,” in *Proc. of the Second USENIX Workshop on Real, Large Distributed Systems*, 2005, pp. 25–30.

[20] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris, “Designing a dht for low latency and high throughput,” in *IN PROCEEDINGS OF THE 1ST NSDI*, 2004, pp. 85–98.