Profiling a Million User DHT

Jarret Falkner Michael Piatek John P. John Arvind Krishnamurthy Thomas Anderson
University of Washington

ABSTRACT

Distributed hash tables (DHTs) provide scalable, key-based lookup of objects in dynamic network environments. Although DHTs have been studied extensively from an analytical perspective, only recently have wide deployments enabled empirical examination. This paper reports measurements of the Azureus BitTorrent client's DHT, which is in active use by more than 1 million nodes on a daily basis. The Azureus DHT operates on untrusted, unreliable end-hosts, offering a glimpse into the implementation challenges associated with making structured overlays work in practice. Our measurements provide characterizations of churn, overhead, and performance in this environment. We leverage these measurements to drive the design of a modified DHT lookup algorithm that reduces median DHT lookup time by an order of magnitude for a nominal increase in overhead.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Measurement, Performance

1. INTRODUCTION

Distributed hash tables provide scalable, key based lookup of objects in dynamic network environments. As its name implies, a DHT exports a hash table interface. Nodes can insert {key, value} pairs and retrieve values for a provided key. In support of this simple API, the underlying DHT layer manages the details of scalable routing and coping with dynamic membership.

As a distributed systems building primitive, DHTs have proven remarkably versatile. Researchers have leveraged scalable lookup to design distributed filesystems, content distribution networks, location sensing systems, and rendezvous-based communication infrastructures. These projects demonstrate the range of services that DHTs can support, but each is predicated on the existence of a robust DHT substrate.

To date, researchers have lacked a comprehensive framework for verifying the robustness of their DHT designs. Wide-area testbeds such as PlanetLab provide realism in terms of network conditions [2, 8], and simulators enable evaluation of large-scale settings [3, 5], but neither approach provides a complete picture of how a DHT would behave when operated in the wild, i.e., the performance of a wide-area *and* large-scale deployment under realistic workloads.

Recently, large-scale, wide-area DHT deployments have emerged that enable measurements that can fill in this missing operational knowledge. DHTs are widely used by several peer-to-peer filesharing networks with users numbering in the millions, session times varying from minutes to days, and connectivity ranging from a dialup modem to 100 Mb fiber. This paper reports on profiling one of these networks, the DHT underlying the Azureus BitTorrent client. Azureus is the most widely used implementation of the popular BitTorrent protocol [7], and our measurements indicate that its DHT supports more than a million concurrent users. Using instrumented clients at diverse vantage points, we gather detailed traces of DHT activity that we both make available to the community as well as analyze to obtain the following results.

- As with many P2P systems, we find evidence that session times are short on average, but heavy tailed.
- Despite short session times, lookups are robust, in part because long-lived nodes shoulder a relatively large amount of routing traffic.
- The network's view of the closest node for a given key is partially inconsistent in both the short and long term, motivating redundant storage and refresh. However, both the number of replicas and the refresh rate are currently more conservative than necessary for the observed Azureus workload, contributing to high maintenance overhead.
- Although the core DHT operation of replica lookup depends on many per-node properties, we identify per-message response probability as a relatively stable global property that distills many other workload aspects, including churn, routing table freshness, and per-node resource limits.
- Using aggregate response probability measurements, an individual node can reduce lookup time by an order of magnitude at the cost of a marginal increase in overhead.

IMC'07, October 24-26, 2007, San Diego, California, USA.

2. BACKGROUND

We report on measurements of the Kademlia-based DHT used by the Azureus BitTorrent client.[1] There are three pieces of relevant context: 1) the uses of Kademlia by Azureus, 2) the properties of the Kademlia DHT, and 3) the key Kademlia parameters that influence measured data. We provide background on each of these in turn.

Azureus maintains a DHT to address a scalability bottleneck. The BitTorrent protocol defines the wire-level details of distributing data in a *swarm*, an overlay mesh of randomly connected end-hosts [1]. To join a swarm, a new client first needs a list of peers already participating in the distribution. Originally, this source of initial peers was provided by a centralized coordinator which dispensed a partial, random list of current participants upon request. Although data distribution in BitTorrent is decentralized, this central coordinator was recognized as a scalability bottleneck that also singlehandedly controlled availability; if the coordinator was inoperable, new users could not be bootstrapped into the mesh.

Azureus uses its DHT to provide a backup source of peers should the centralized coordinator become unavailable. The DHT also enables distribution without specifying *any* central coordinator. In addition to contacting the coordinator, a new Azureus user also performs a DHT `get` operation for the swarm's key, determined by hashing its metadata. This `get` returns a set of Azureus peers. The new peer appends itself to this list by performing a corresponding `put` operation, which is refreshed every four hours as long as the client is alive. In addition, Azureus exposes the DHT interface through its plugin API, supporting additional services such as per-swarm chat and ratings.

The Azureus DHT is based on Kademlia. As a DHT, Kademlia has a `put`/`get` API and provides a logarithmic bound in the total number of nodes contacted during each of these operations. In Kademlia, each node is assigned a unique identifier in a 160 bit key space. In Azureus, this identifier is the SHA-1 hash of a peer's IP address and DHT port. Each node also maintains a local routing table containing a set of buckets populated with other nodes that are "closer" to ranges of the key space. Kademlia defines proximity on the basis of the XOR metric, i.e., the distance between keys X and Y is simply the integer value of X$\oplus$Y. The routing table is constructed so that each lookup reduces the XOR distance to the target key by 1/2, providing logarithmic lookup (for details, see [6]). Both the `put` and `get` operations are iterative, performing successive queries of intermediate nodes until no closer contacts can be obtained.

Although conceptually straightforward, implementations of the Kademlia algorithm need to cope with several challenges of operating on unreliable end-hosts.

- **Churn:** Nodes arrive and depart rapidly, making routing table entries stale. To address this, each routing table bucket responsible for a certain key space range contains redundant peers. In Azureus, buckets are of size 20. This redundancy is used to conduct parallel lookups. Instead of querying a single peer at each successive routing intermediary, a set of intermediaries are probed. To limit resource consumption, the number of concurrent probes is limited to 10 in Azureus.
- **Consistency:** Churn results in routing tables that are not only stale, but also inconsistent. Ideally, a value need only be stored on the single node closest to the key. In practice, that node may depart prematurely. Further, concurrent lookups from distinct regions of the overlay might not have a consistent view of which node is closest. Azureus addresses inconsistency via replication. Each value is replicated on the 20 nodes closest to the key from the perspective of the client performing the `put`.
- **Failures:** Kademlia uses UDP for message transport, requiring application-level detection of both message and node failures. The default implementation relies on 20 second timeouts for message expiry and considers a node down once it fails to respond to two back-to-back messages. To detect failures in the absence of network activity, clients lookup a random key every 5 minutes.

The solutions adopted by Azureus to these challenges carry tradeoffs. For example, failures can be detected more rapidly through aggressive probing of routing table entries, improving consistency but increasing overhead. These tradeoffs are often pinned to parameters that could benefit from measurement, e.g., unnecessarily long timeouts degrade performance. In the next section, we elucidate these tradeoffs, characterizing them in practice through measurement.

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

3. MEASUREMENTS

This section reports profiling results of the Azureus DHT collected during February–May, 2007 by 250 PlanetLab and 8 UW vantage points. First, we present a measured distribution of node session times in the DHT, discovering a spectrum of liveness ranging from minutes to days. We then synthesize these results with measurements of DHT maintenance overhead, providing a breakdown of message types and a coarse estimate of the time required for new nodes to percolate throughout the DHT's routing tables. Next, we examine the inconsistency of DHT routing tables as well as their evolution over hours. Finally, we characterize the probability of a DHT node replying to a protocol message as a function of the time since the request was sent. In each case, we refrain from fitting statistical distributions to data, instead opting to make available the raw samples obtained in our traces to the community.

3.1 Approximating session times

The distribution of node session times is central to many design decisions underlying DHT implementations, e.g., session lengths determine the rate at which routing table entries need to be probed for freshness. Obtaining an accurate measure of session times requires a random sampling method and knowledge about when a sampled node joins and departs. Unfortunately, the DHT does not expose this information. Instead, we provide an estimate of session times. To gather a set of candidate nodes, we instrumented DHT clients to perform random `get` requests, collecting a set of 300,000 nodes drawn from the tables returned by those requests. Immediately after observing a node, we issued a heartbeat message, repeating this probe every 2.5 minutes. Nodes failing to respond to two consecutive probes are considered down. This check was continued for 48 hours after first observing a node.

Figure 1 shows the cumulative fraction of observed nodes remaining responsive over time. Surprisingly, roughly 46% of peers obtained from routing table entries did not respond to even immediate probes, suggesting that a significant frac-

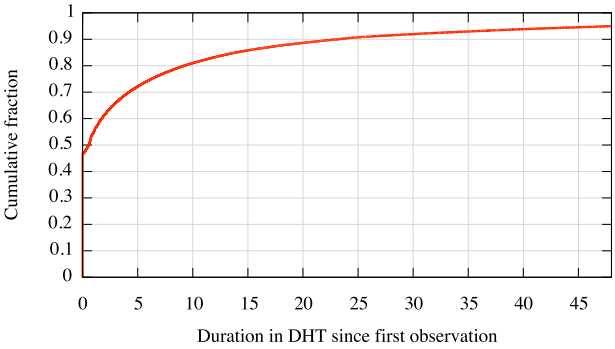


Figure 1: Freshness of DHT routing table entries and persistence for 48 hours after first observation.

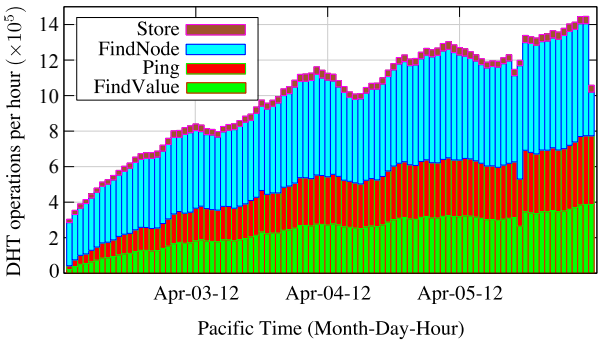


Figure 2: Two day trace of DHT messages received by 125 vantage points on PlanetLab. Bars give the absolute number of each measure type over a one hour period.

tion of routing table entries are stale and point to inactive nodes. After 48 hours, roughly 5% remained active, suggesting a minority of highly available peers. The remaining peers are spread over a spectrum, suggesting that session times in the Azureus DHT are on the order of hours. Because we sample existing nodes in the DHT, we cannot recover the precise duration of their participation before our initial observation, nor can we account for the potential bias of long-lived nodes being more prevalent in routing tables, which may skew our measurements by over-representing long-lived nodes. This bias is independently noteworthy for its impact on node bootstrapping time, the topic we examine next.

3.2 Bootstrapping and overhead

DHTs operating on unreliable end-hosts are faced with a design tradeoff when choosing how best to *bootstrap* new nodes into the overlay, i.e., incorporate their resources. New nodes are likely to be short-lived, and immediately incorporating them into the overlay risks polluting routing tables with short-lived nodes and storing data on transient replicas. Alternatively, incorporating only those nodes believed to be stable requires predicting longevity, a challenging task with transient participants and local history alone.

Azureus adopts the former method, incorporating new nodes into the DHT immediately. However, their responsibility is initially limited by the slow update rate of routing tables in practice. When a node joins, it performs a lookup of its identifier to populate its routing table and notify others of its presence in the system. However, Azureus peers preferentially respond to routing requests with longer-lived entries in their routing tables. Thus, while new nodes can immediately serve as replicas for their range of the key space, they tend to percolate slowly into the routing tables of distant nodes.

This slow bootstrapping time is manifested in our traces of DHT client activity, which we collected from 125 PlanetLab vantage points running the original Azureus client implementation. Each vantage point joined the Azureus DHT simultaneously, logging messages over a two day period. No lookups or storage requests were sent beyond those issued to maintain routing table freshness or migrate values to newly joined nodes. Figure 2 gives the absolute number of each Kademlia message type received over one hour intervals. For the first few hours, the newly joined nodes receive few DHT messages. Load increases steadily over a period of several days, indicating the time scale at which routing tables update in practice. Our trace also suggests that DHT maintenance overhead dominates resource consumption for the Azureus workload; routing table updates (FindNode) and liveness checks (Ping) comprise 81% of messages.

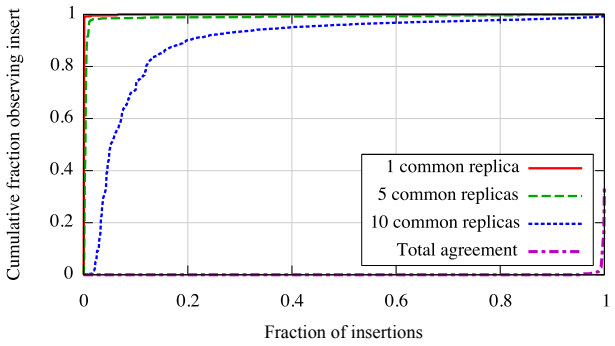
3.3 Consistency and persistence

Apart from scalability, DHTs attempt to provide an appealing consistency guarantee; a `get` operation will return a value if and only if the {key, value} pair exists in the DHT. Unfortunately, this strong guarantee on consistency can only be made in the absence of churn. Over long periods, data replication and routing table redundancy are intended to foster, but not guarantee, consistency under realistic operating conditions.

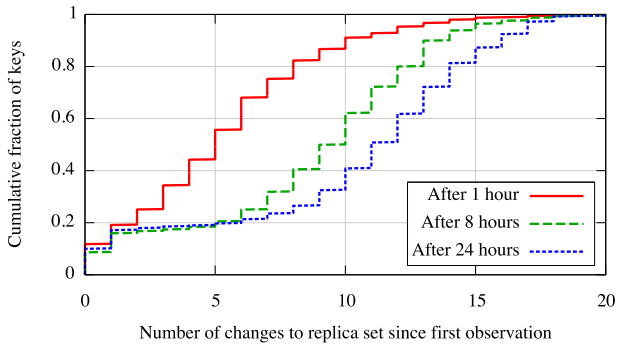
Promoting consistency through replication depends on a number of implementation details. Each value in Azureus is replicated at the 20 nodes closest to the target key. We refer to this set of replicas as R. Replication compensates for the inconsistent view of R in the short-term arising from slow convergence of routing tables; we find that membership in R varies greatly from different vantage points in the DHT. Replication also mitigates the impact of unexpected failures. As membership in R changes under churn, values ideally migrate from node to node, remaining in the set of 20 closest nodes at any point in time. In practice, slow failure detection and unexpected failures shift the burden of ensuring data persistence from the DHT to the publishing node, which typically performs periodic `puts` to refresh data.

A familiar tradeoff arises from a replication-based approach. A large replica set compensates for inconsistent routing and reduces the need to frequently refresh data for persistence. This comes at an increased cost for `puts` in terms of time, nodes contacted, and storage. The replication factor controls this tradeoff, but estimating consistency in terms of replication requires workload data.

We study this tradeoff for the Azureus workload in terms of both short-term and long-term consistency. In the short-term, developers need an estimate of the replication required for a `put` to be visible to the entire overlay. Over the long-term, developers need an estimate of the rate of change of the replica set to determine how often data should be republished to ensure persistence.



(a) Short-term consistency



(b) Evolution of replica set

Figure 3: Profiling routing table consistency. *Left:* The cumulative fraction of vantage points observing a DHT put immediately after insertion. *Right:* Evolution of replication set over time from the perspective of a single vantage point.

Short-term consistency: To measure short-term consistency, we performed `puts` of random keys into the DHT, measuring the visibility of each insertion from 250 PlanetLab vantage points. Each insert was performed by a randomly chosen PlanetLab host. The remaining vantage points performed a `get` operation immediately after the insert completed, each returning a set of replicas—the 20 nodes closest to the key from an individual vantage point's perspective. With consistent routing tables, each of these sets would be identical. In practice, different vantage points observe different sets of replicas due to churn and slow routing table convergence.

In the absence of perfect consistency, how many replicas are needed for inserts to be widely visible? Figure 3(a) shows the fraction of vantage points observing an insert immediately after it completes, aggregated over 931 random `puts`. Each line corresponds to a given level of commonality among replica sets. For example, the long, flat shape of *5 common replicas* indicates that for the vast majority of insertions (length of x-axis), at least 5 replicas are visible to most vantage points (height on y-axis). 99% of vantage points observe at least one replica out of 20 for 99% of inserts, and 98% of vantage points observe at least 5 replicas for 98% of inserts. However, this quickly falls off as the desired commonality in replica set increases. In the extreme case of total replica set agreement, for 95% percent of measured `puts`, the inserting node does not share the same 20 replicas with any vantage point.

Long-term consistency: Data persistence is easily attainable if the DHT exhibits both a consistent view of the replica set in the short-term and stable membership in the replica set over the long-term. Under changing replica set membership, a straightforward method to promote data persistence is to have nodes periodically reinsert data into the DHT. How often to perform such refreshes depends largely on churn. We examine the changing membership of the closest replica set from 125 PlanetLab vantage points. Each vantage point selected a random key and a 1, 8, or 24 hour interval. To measure the evolution of the replica set, we consider the change in membership between the replicas returned by the initial insert and those returned by a lookup after the chosen time interval. Figure 3(b) summarizes the change in replica set for 639, 616, and 503 keys for 1, 8, and 24 hour intervals, respectively. These results suggest that periodic insertions can be performed at the granularity of hours with little impact on data persistence. After one hour, more than 98% of replica sets shared at least five members with those of the initial insert.

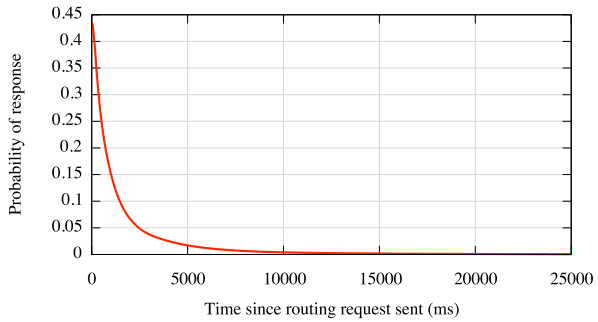


Figure 4: The probability of receiving a reply to a DHT message as a function of time after sending.

3.4 Response probability

The Azureus DHT uses UDP for message transport, as its routing table redundancy and size would otherwise necessitate hundreds of TCP connections per-node. To limit resource consumption, Azureus restricts the number of outstanding messages and imposes processing rate-limits on incoming messages. The combination of unreliable message delivery and limits on resource consumption severely hinders performance. Marking outstanding messages as expired is necessary to avoid blocking on the hard outstanding message limit, but rate-limited processing inflates message RTTs, requiring lengthy timeouts.

For a single node seeking to improve response time, disabling local rate-limits is straightforward. However, choosing an appropriate timeout for message responses requires workload data. Figure 4 shows the probability of receiving a response to an outstanding message as a function of the time since it was sent, measured over 45 million messages sent during our trace from PlanetLab and UW vantage points. Although most are received after just one sec-

ond, some responses arrive much later with the response behavior influenced by a multitude of factors such as network delays, routing table staleness, and rate limits. While the response time for an individual message could vary significantly, we observed that the response probability distribution is remarkably stable over both short and long time scales and across all vantage points.[2] We leverage this stability in the next section to construct an improved lookup algorithm parameterized by our measurements.

4. IMPROVING PERFORMANCE

The performance of the Azureus DHT is largely controlled by the parameters that impact routing delay: lookup parallelism, message timeouts, and rate-limits on message processing. The default Azureus implementation adopts settings for these parameters tuned for its use of the DHT largely as a redundant source of peers in large file downloads. DHT lookups in this environment need not be fast. Downloads typically take many minutes or even hours, and the extra time spent acquiring DHT peers has a marginal impact on end-to-end download time in the common case. For Azureus, limiting resource consumption is much more important than improving performance. A DHT node rapidly dispatching hundreds of UDP packets to many end-hosts concurrently may trigger intrusion detection systems or denial-of-service filtering rules. However, this is precisely the behavior of a latency-sensitive DHT lookup with high parallelism.

The original implementation of Azureus adopts a design intended to limit resource consumption in its DHT. However, part of the promise of DHTs is their potential to serve as a generic distributed systems building block, independent of an individual service [9]. To fully deliver on that promise, DHTs should be adaptable, exposing the performance / overhead tradeoff to developers. This section reports on measurements of the adaptability of the Azureus DHT, examining the question: through local modifications only, can the existing Azureus DHT be adapted to support latency sensitive services? Our preliminary results suggest that it can, potentially allowing future developers to leverage the large-scale Azureus DHT to build diverse distributed services.

The original implementation of DHT lookup frames resource control in terms of 1) hard bounds on outstanding messages and 2) rate-limits on message processing. Clearly, disabling processing rate-limits locally improves performance, and we do so in our modified implementation. For random key lookups, this reduced median per-message response time from 642 to 281 milliseconds. Adjusting rate-limits provides direct control over the performance / overhead tradeoff at the granularity of per-hop *messages*, but adjusting this tradeoff at the granularity of end-to-end *lookups* requires eliminating hard bounds on outstanding messages.

Controlling resource consumption through bounds on outstanding messages requires a method of expiring dead messages. Because DHT messages are small, typically fitting in a single UDP packet, nodes rely on timeouts for expiry rather than more sophisticated methods that may, for instance, infer loss from out-of-order delivery. Selecting a timeout value itself presents a tradeoff: too low a value results in redundant messages while too high a value degrades performance, blocking while an almost certainly failed message times out. The response times shown in Figure 4 suggest that some messages may arrive after tens of seconds, and in line with a resource-conserving approach, Azureus adopts a 20 second message timeout. This timeout, in combination with a limit of 10 on the number of outstanding messages and low response probability borne out of stale routing tables (Figures 1, 4) results in high end-to-end DHT lookup times in Azureus. The *Original* lines of Figure 5 give the CDF of time and resource consumption for 3,007 random DHT `get` operations from PlanetLab and UW vantage points. Median lookup time is 127 seconds.

Lengthy lookup times are the result of conservative timeouts. Rather than attempting to tune the timeout value for performance, we eliminate timeouts altogether. We instead elect to parameterize our revised lookup algorithm in terms of the number of *expected* responses. Measured response probability data (Figure 4) allows us to compute the expected number of message replies at a given instant, i.e., the sum of the response probabilities of all outstanding messages. We use a threshold on this expectation to implicitly define the rate at which outgoing messages will be sent, generalizing rate-limits on local queues as well as hard bounds on outstanding messages. Our revised lookup algorithm computes the number of expected replies every 100 milliseconds, issuing new messages whenever the expectation drops below a specified parallelism threshold.

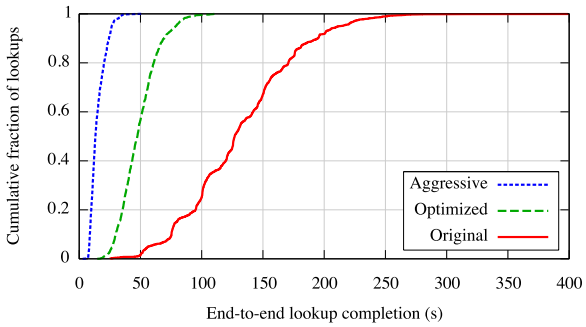
Parameterizing lookup in terms of expected message responses more accurately reflects the intuition behind parallel searches—the natural knob for tuning the performance / overhead tradeoff in Kademlia-based DHTs. Because of the interplay between parallelism, timeouts, and outstanding message bounds, tuning performance in the original Azureus DHT implementation requires adjusting all three quantities. Further, these adjustments depend on the number of expected failures. By leveraging aggregate profile data, we can eliminate multi-variable tuning, instead presenting developers with a smooth tradeoff in terms of number of expected results.

We explore the impact of our revised algorithm on performance and overhead in Figure 5. The *aggressive* lines correspond to conducting, in expectation, 6 parallel lookups—essentially issuing messages as quickly as possible. This results in median end-to-end lookup time dropping from 127 seconds to 13. With aggressive parallelism, the total number of nodes contacted increases from 52 to 150. By eliminating timeouts, our revised lookup algorithm can reduce lookup time by an order of magnitude for only a threefold increase in the number of nodes contacted. Our algorithm also improves performance even while controlling resource consumption. The *optimized* results correspond to a parallelism threshold of 0.8—simply compensating for expected response failures. In this case, median lookup time decreases from 127 seconds to 47 while increasing contacted nodes only slightly from 52 to 60.

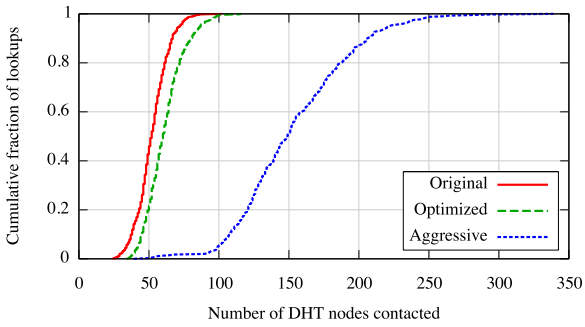
5. RELATED WORK

DHTs have seen a large body of work in recent years. Steiner et al. report heavy tailed session times on the Kademlia-based DHT used by eMule, another file-sharing network [10]. We observe a similar distribution of session times (live nodes in Figure 1), but find that many routing table entries in the Azureus Kademlia DHT are stale. We attribute this to the

[2]We do not present the data here due to space limitations.



(a) Completion times for DHT lookup algorithms



(b) Cost in terms of number of nodes contacted

Figure 5: End-to-end lookup performance. With aggressive parallelism, median lookup time is reduced by an order of magnitude.

fundamentally different use of eMule's DHT, which provides for the network's core search functionality and is heavily used as a result. Because Kademlia lookups trigger routing table updates, comparatively heavy use has the side-effect of promoting routing table freshness. Our work exposes the need to explicitly and aggressively refresh routing tables in the absence of demand or under heavy churn.

A separate analysis of eMule presented Stutzbach et al. [11] finds that aggressive parallel lookups increase overhead without significantly increasing performance. We find exactly the opposite. Due to the prevalence of stale routing table entries resulting from a fundamentally different workload and refresh policy, increasing parallelism improves lookup performance in Azureus substantially. The relative effectiveness of coping with stale routing tables through parallel lookups (as in Azureus) or aggressive refresh (as in eMule) remains an open question for future work. But, by identifying message response probability as the key parameter for tuning the performance / overhead tradeoff, we develop a revised lookup algorithm that can dynamically adapt parallelism via local observations of response probability in either case.

Ledlie et al. leverage the Azureus DHT to examine the operation of network coordinate systems in the wild, finding that using coordinate information can improve performance when lookups do not experience timeouts [4]. Our work is complimentary, demonstrating that the dominant influence on performance in practice is failures and timeout-based resource control. Our revised lookup algorithm avoids reliance on timeouts, clearing the way for further optimizations such as network coordinates.

6. CONCLUSION

We have reported profiling measurements of a large-scale DHT operating on end-hosts. The unpredictability of this environment is manifested in heavy tailed session times, inconsistent routing tables, and high overhead. Still, DHT operation is robust, leveraging wide data replication and routing redundancy. These techniques give rise to tradeoffs between availability, overhead, and performance, which we have identified. While the Azureus DHT as implemented trades off performance for availability and reduced overhead, surprisingly, individual clients are not forced to do the same. Through local modifications only, we demonstrate that a node prioritizing performance can improve response time by an order of magnitude, although it must expend more resources to do so. Still, our analysis is not exhaustive, and so we make available the traces we obtained during our measurements to promote wider understanding of end-host DHTs in the wild.

7. REFERENCES

[1] B. Cohen. Incentives build robustness in BitTorrent. In *Proc. of P2P-ECON*, 2003.

[2] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and DHTs. In *Proc. of WORLDS*, 2005.

[3] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proc. of SIGCOMM*, 2003.

[4] J. Ledlie, P. Gardner, and M. Seltzer. Network coordinates in the wild. In *Proc. of NSDI*, 2007.

[5] 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 *Proc. of INFOCOM*, 2005.

[6] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. of IPTPS*, 2002.

[7] M. Piatek, T. Isdal, A. Krishnamurthy, and T. Anderson. Do incentives build robustness in BitTorrent? In *Proc. of NSDI*, April 2007.

[8] S. Rhea, B.-G. Chun, J. Kubiatowicz, and S. Shenker. Fixing the embarrassing slowness of OpenDHT on PlanetLab. In *Proc. of WORLDS*, 2005.

[9] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proc. of SIGCOMM*, 2005.

[10] M. Steiner, E. W. Biersack, and T. Ennajjary. Actively monitoring peers in KAD. In *Proc. of IPTPS*, 2007.

[11] D. Stutzbach and R. Rejaie. Improving lookup performance over a widely-deployed DHT. In *Proc. of INFOCOM*, 2006.