An Analysis of BitTorrent's Two Kademlia-Based DHTs

Scott A. Crosby and Dan S. Wallach
Department of Computer Science, Rice University
{scrosby, dwallach}@cs.rice.edu

Abstract

Despite interest in structured peer-to-peer overlays and their scalability to millions of nodes, few, if any, overlays operate at that scale. This paper considers the distributed hash table extensions supported by modern BitTorrent clients, which implement a Kademlia-style structured overlay network among millions of BitTorrent users. As there are two disjoint Kademlia-based DHTs in use, we collected two weeks of traces from each DHT. We examine churn, reachability, latency, and liveness of nodes in these overlays, and identify a variety of problems, such as median lookup times of over a minute. We show that Kademlia's choice of iterative routing and its lack of a preferential refresh of its local neighborhood cause correctness problems and poor performance. We also identify implementation bugs, design issues, and security concerns that limit the effectiveness of these DHTs and we offer possible solutions for their improvement.

1 Introduction

Overlay or peer-to-peer (p2p) networking technologies have been widely studied as mechanisms to support wide-scale, fault-tolerant, distributed storage and communication systems. By implementing a distributed hash table (DHT) abstraction, a wide variety of applications can be easily adapted to using the overlay network. Unfortunately, relatively little is known about such DHTs' real-world behavior with large numbers of users and real workloads.

While Kazaa does operate at this scale, and its user behavior has been well studied [20, 29], Kazaa does not use a "structured overlay," a logarithmic routing structure common to many p2p systems. Such structured overlays have generally only been evaluated in limited deployments or on PlanetLab [8], which does not necessarily represent what might be observed elsewhere [23].

BitTorrent is a p2p system allowing large numbers of users to share the burden of downloading very large files. The most notable feature of BitTorrent is its use of tit-for-tat trading to incentivize users to give each other higher bandwidth service [9]. The BitTorrent protocol uses a *central tracker* as a rendezvous point So that nodes interesed in the same file can find each other. As such, classic BitTorrent also lacks a structured overlay. However, the most popular BitTorrent clients support a *distributed tracker*, serving the same purpose as the central tracker, but implemented with a variant of the Kademlia [31] DHT. At the present time, there are two Kademlia overlays in use: one by Azureus clients[1] and one by many other clients including Mainline[2] and BitComet[3]. Each of these overlay networks include a million live nodes, providing an excellent opportunity for data collection and analysis of a real, deployed DHT on the Internet.

2 Related Work

P2p systems have been extensively studied in the literature. Risson and Moors [42] wrote an extensive survey with 363 citations. We limit our discussion to work most relevant to the DHTs used in BitTorrent.

Saroiu et al. [44] characterized the Gnutella and Napster systems, measuring bottleneck bandwidth and node lifetimes, and finding significant heterogeneity across the nodes.

Gummadi et al. [19] studied several routing geometries and the effectiveness of proximity neighbor/route selection on performance with up to 64,000 nodes. They did not assess the impact of dead nodes and the resultant timeouts, which we will show has a significant impact on performance.

Li et al. [28] compared 5 DHTs and their parameter choices by simulating each system and parameter vector on a 1024 node network. We will show how poor parameter selection has impacted the performance of BitTorrent's DHTs.

Rhea et al. [39] also considered performance issues including the use iterative routing, delay aware routing, and multiple gateways, with an emphasis on improving the performance of OpenDHT [41] on 300 PlanetLab [8] nodes.

Liang et al. [29] characterized the design of the two-level FastTrack overlay, used by Kazaa, Grokster and iMesh. This overlay had 3 million active users.

Skype also operates at this scale, but it encrypts all traffic, complicating its analysis [18]. Skype is believed to use a two-tiered overlay with the top tier fully connected [3].

Two other file-sharing systems use Kademlia. The now-defunct Overnet[4] used the DHT for searching. eMule's Kad network[5], also based on Kademlia, uses the DHT for keyword searching and is estimated to have 4 million nodes. Kad would also be suitable for analysis.

3 BitTorrent Background

BitTorrent was designed to incentivize a large collection of users, all wishing to download a single large file, to share their respective upstream bandwidths. Pouwelse et al. [36] present measurements of the millions of users who regularly trade files with BitTorrent,

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

[2]`http://www.bittorrent.com`

[3]`http://www.bitcomet.com`

[4]Overnet's web pages can now only be found on the Internet Archive. `http://web.archive.org/web/20060418040925/http://www.edonkey2000.com/documentation/how_on.html`

[5]`http://en.wikipedia.org/wiki/Kad_Network`

as does Legout et al. [26]. Izal et al. [24] measures the effectiveness of BitTorrent in a flash crowd. Qiu et al. [37] and Piccolo et al. [35] present fluid models of BitTorrent's behavior. Guo et al. [22] models and measures BitTorrent, identifying limitations caused by exponentially declining popularity. Bharambe et al. [2] simulates BitTorrent and show hows to improve its fairness in a heterogeneous environment.

Of course, users may have an incentive to freeload on the system, as was observed in Gnutella [1]. BitTorrent addresses this through a tit-for-tat system, where a node will preferentially spend its upstream bandwidth on peers which give it good downstream bandwidth [9]. While freeloading is still be possible, the download speed improves noticeably if a node acts as intended. (Incentives issues in p2p systems are widely studied [11, 14, 15, 17, 25, 32] but are not the main thrust of this paper.)

Otherwise, classic BitTorrent is fairly straightforward. A file to be shared is described by a *torrent*, typically distributed through web servers. A torrent file contains metadata, including cryptographic hashes, allowing the resulting file to be validated; a torrent also contains a pointer to a *central tracker*, whose sole purpose is to introduce nodes to their peers. Clients announce themselves to the tracker once every 30 minutes. The client only queries the tracker for the identities of new peers if its current peers are insufficient. As a result, a tracker, despite running on a single machine, can handle a very large number of concurrent nodes.

The set of peers working to download a particular torrent is called a *swarm*. Once a node has completely downloaded the file but has not yet left the swarm, it becomes a *seed*, sharing its bandwidth among all requesters. One or more seeding nodes are also typically set up by a document's publisher to get things started.

The tit-for-tat exchanges are based on 64KB-1024KB *pieces*. Peers will exchange information with each other about which pieces they have. Most BitTorrent clients implement a "rarest-piece first" policy that encourages broader distribution of rare pieces, making all pieces more uniformly available [27]. More recent research has shown free riding is practical [30] and that a peer can optimize its performance by carefully allocating its upload bandwidth [34].

3.1 Extensions

As BitTorrent usage has expanded, a number of extensions have been added to classic BitTorrent, with different clients supporting different extensions. The Message Stream Encryption / Protocol Encryption extension, for example, encrypts all data exchanges in the swarm and may defeat some traffic shaping devices. The most relevant extensions to our study both concern peer discovery: the Kademlia DHTs and the Peer Exchange Protocol (PEX). PEX is a gossip protocol; peers in a swarm exchange their peer lists, accelerating peer discovery and reducing tracker load. Fry and Reiter [16] have shown that a variant of PEX using random walks to discover new peers can be just as effective as using a tracker. The DHT extensions replace or supplement the central tracker, providing for node discovery and avoiding the single-point-of-failure of a central tracker.

BitTorrent clients implement two mutually incompatible DHTs. The Mainline DHT (hereafter, MDHT) is implemented by Mainline BitTorrent, μtorrent, BitLord and BitComet. The Azureus DHT (hereafter, ADHT) is implemented only by Azureus.

Both DHTs are based on Kademlia, but have non-trivial differences. Because the ADHT and MDHT are disjoint, and many users still use older clients that have no DHT support, completely distributed tracking would create a variety of interoperability problems. Unsurprisingly, virtually all torrents define a centralized tracker, allowing the DHT to operate as a backup to the central tracker, rather than depending on it for correctness.

4 Kademlia Background

Kademlia implements a tree-based routing topology based around an XOR distance metric where the XOR of two nodeIDs is used as the distance between them. Kademlia is based around a prefix-based routing table, similar to Pastry [43], with logarithmic lookup performance, but without any locality-based routing optimizations. A Kademlia routing table is parameterized by two numbers, 2^w, the width (i.e., w bits), and b, the bucket size.

For a node with a 160 bit id X, the routing table has $160/w$ layers starting at 0. Layer l contains $2^w - 1$ buckets with nodes whose IDs match X on the first $l \cdot w$ bits, and differ from X on the next w bits. Each bucket contains up to b most-recently-seen nodes. This description roughly covers Kademlia as it is implemented in Azureus and Mainline.

The full Kademlia design incorporates additional routing buckets that serve a purpose comparable to Pastry's leafset: identifying every node nearby. A Kademlia node selects the longest prefix of its nodeID such that the number of nodes sharing any longer prefix is less than b, and then tracks *all* nodes having that prefix. For correctness, a host *must never overestimate* the number of nodes sharing any longer prefix by counting dead nodes as alive, a task made more difficult in an environment with churn. To the best of our knowledge, *this additional routing table is not implemented by any BitTorrent client*, meaning that BitTorrent clients may not always be able to find the node whose nodeID is closest to a given lookupID (See also, Section 5.1).

New nodes for addition to the routing table are discovered opportunistically from incoming DHT queries and in replies to outgoing lookups. A new node replaces the oldest node in the appropriate routing bucket if the old node has not been seen alive in 15 minutes and does not reply to an active ping. To prevent staleness, Kademlia will route to a random ID in any bucket that has been quiescent for 15 minutes.

When routing to a lookupID, Kademlia uses iterative routing, where a host contacts peers with progressively smaller XOR distances to the lookupID in turn (see Figure 1). This contrasts with Pastry's recursive routing, where each host forwards the message along the chain to the destination (see Figure 2). Iterative routing can be performed concurrently, with multiple outstanding requests to decrease latency and reduce the impact of timeouts.

4.1 Implementation details

Mainline and Azureus have fairly faithful implementations of Kademlia, except for ommitting the extended routing table. As with the centralized tracker, a host queries

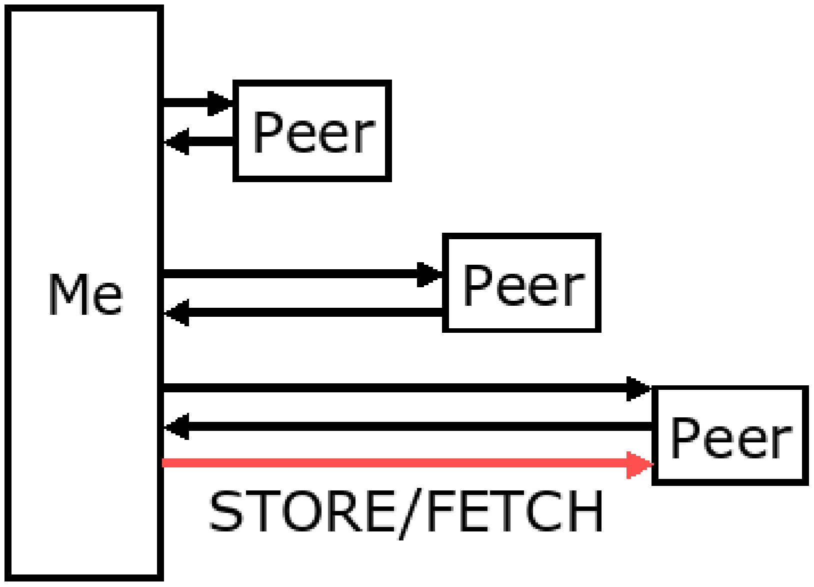

Figure 1: Iterative routing

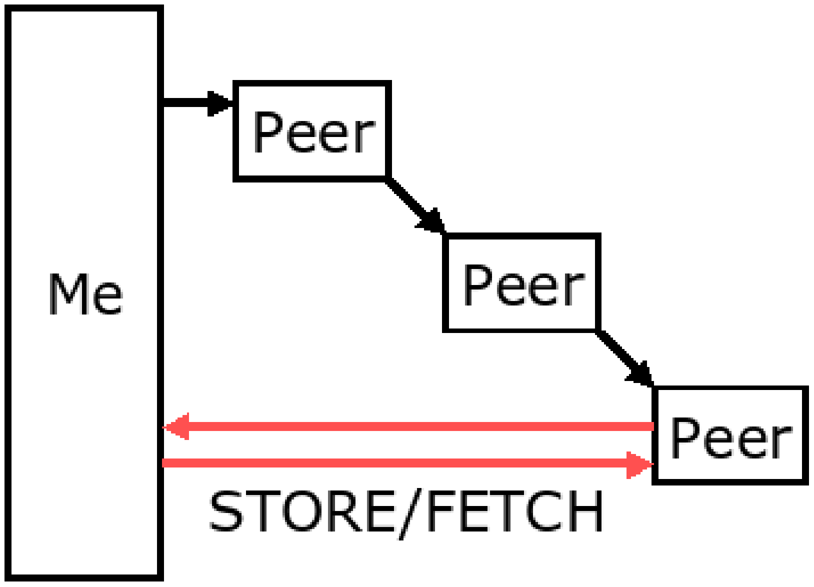

Figure 2: Recursive routing

the DHT every 30 minutes if it requires more peers for a swarm.

4.1.1 Mainline implementation

We studied version 4.20.9 of Mainline BitTorrent, which is open source and implemented in Python. Mainline, BitComet, and several other BitTorrent clients all use the same MDHT. MDHT implements Kademlia with a width of 2, $w = 1$, and $b = 8$ nodes in each bucket; keys are replicated on the three nodes with nodeID nearest the key with a 30-minute timeout, after which the key must be reinserted by its original publisher. There is no other migration of keys, so if all the nodes holding a key fail, the key will be lost.

Experimentally, we would like to make queries to random nodeIDs, but MDHT does not already make suitable queries that we can instrument. Instead, we modified our client to perform a lookup on a random ID every 10 minutes, accounting, in total, for 10% of all lookups our clients perform. An unmodified client that joined 3 swarms would generate a similar level of traffic.

If a torrent file defines a central tracker, Mainline will *never* use the MDHT for tracking. Unsurprisingly, very little traffic in MDHT appears to originate from Mainline clients. If a torrent uses the distributed tracker, it contains a subset of DHT peers, allowing Mainline to bootstrap itself into the DHT. Most of the clients that we have seen in MDHT appear to be BitComet (see Section 4.1.3, below).

4.1.2 Azureus implementation

We studied version 2.5.0.0 of Azureus which is open source and implemented in Java. ADHT implements Kademlia with a width of 16, $w = 4$, and $b = 20$ nodes in each bucket; keys are replicated on the 20 nodes with nodeID nearest the key. When doing lookups, Azureus uses 5-way concurrency. Unlike MDHT, ADHT is partially stateful; while keys must be refreshed every 8 hours, keys are also replicated and migrated to new hosts in response to node churn. Azureus will first use the central tracker, as specified in the torrent, and will only use the ADHT if the tracker is unreachable. Azureus supports PEX and a variety of other extensions.

Where Mainline refreshes quiescent buckets every 15 minutes, Azureus starts a refresh cycle 30 minutes after the prior one finishes and performs random lookups every 5 minutes when not refreshing buckets. To initially boostrap itself into ADHT, Azureus uses its regular swarming peers, which may be more robust than Mainline's use of the the torrent file to list bootstrapping peers.

4.1.3 Other implementations

BitComet versions newer than version .53 appear to use the centralized tracker in parallel with the MDHT. BitLord only supports the PEX extensions with other BitLord nodes and recent versions appears to support MDHT. μtorrent supports PEX, MDHT, and several other extensions.

None of these clients are open source, but they do interoperate correctly with

MDHT. As such, we assume they implement MDHT in a similar fashion to the Mainline client.

5 Design issues

We now consider general design issues that affect the performance of both MDHT and ADHT.

5.1 Identifying replica peers

In a DHT, it is useful for a node to be able to identify whether it is or is not responsible for a given key, particularly when supporting migration and replication. For efficiency, a node should not have to contact any other node to determine its responsibilities.

In Pastry, every node explicitly tracks (and regularly pings) its nearest neighbors, forming a *leafset*. So long as the replication factor is smaller than half the size of the leafset, every node can determine when a replica peer has failed, requiring the migration of data. Kademlia similarly stores keys on the n nearest nodes in XOR distance to the key. MDHT and ADHT implement Kademlia without its extended routing table (see Section 4), resulting in no easy way to unambiguously determine which peers are responsible for a key, complicating any replication or migration strategy.

Instead, they use Kademlia's iterative lookup algorithm to find the nearest node to an ID using a priority queue ordered by distance to the key. The queue is initialized from the routing table where the search begins. The nearest node in the queue is queried for the k closest nodes it knows to the key. The lookup terminates when no new nodes are found closer to the key. This algorithm can be extended to find the n nearest nodes to the key by modifying the termination condition so that the lookup terminates only after the n nearest nodes in the priority queue have all been contacted and no additional nearby nodes have been discovered.

The lookup has been proven correct, with the extended Kademlia routing table, when $k = n = b$ [31]. The correctness of lookup is unproven for Mainline and Azureus, without the extended routing table. Furthermore, as the number of dead nodes in the routing table increases, k is effectively less than n or b, and the lookup might then miss live nodes near the key. Such failures manifest themselves when we try to estimate the size of the DHT (see Section 7.9) and can be seen in ADHT's migration and replication system. MDHT does not perform migration at all.

5.2 Opportunism in Kademlia

Many distributed systems, including Kademlia, opportunistically piggyback maintenance traffic onto application traffic, saving the cost of separate traffic devoted solely to maintenance. Kademlia uses the senders of incoming queries as an opportunistic source of new nodes for refreshing routing tables. When available, Kademlia uses RPC timeout failures on outgoing messages in lieu of explicit *ping* messages for checking node liveness and keeping the routing table fresh. Unfortunately, iterative routing curtails both opportunities.

Consider: the top layer of a host's routing table can potentially contain a reference to any node in the overlay. Lower layers in a host's routing table have a prefix in common with the host's nodeID. As such, an outgoing query to a random nodeID will be unlikely to ever satisfy the prefix match for lower layers. The probability that such a random lookup will use a bucket in layer *l* in the local routing table is $1/2^{w \cdot l}$, decreasing exponentially as we get to lower layers.

A similar issue exists in learning new nodes, biasing node discovery toward the highest and lowest levels of the routing table while starving the middle. With iterative routing, the ID of the sender of a message is independent of the key being looked up. Since the senders' IDs are uniformly spread over the node ID space, they will be unlikely to be eligible for all but the highest layers of the routing table. Likewise, newly joining nodes look for their own nodeID first, populate their routing table as they search. As a side effect, they announce themselves to their future neighbors, and populate the lower levels of their neighbors' routing tables. No equivalent opportunistic method exists to populate the middle layers.

Minor modifications to Kademlia could address these concerns. For example, iterative routing could be enhanced with a feedback mechanism; whenever a node contacts a dead peer during a lookup it informs the supplier of that peer of the apparent death. Another fix would be for nodes interested in the death of a peer to cooperate by gossiping information about that peer [46]. Furthermore, iterative lookup requests could include some or all of the live nodes discovered earlier within the same lookup. (Such extensions, however, might allow for security attacks; a malicious node could return a list of other malicious nodes, polluting the victim node's routing table [6].)

Another alternative would be for Kademlia to adopt a recursive routing strategy, rather than its iterative strategy. Each node along the lookup path would then have an opportunity to discover dead nodes in its routing table. The odds of a node being queried at any level of its routing table would be uniform, so the opportunistic refreshing would have a similar effect across the entire routing table.

In Kademlia, opportunistic routing table refreshes depend on DHT application traffic to piggyback on. In the current DHT, there is little DHT application traffic to piggyback upon, only 2 messages per swarm per hour compared to a measured maintenance traffic of about one message per minute. Unless additional uses are found for the DHTs, explicit refresh traffic will continue to be necessary to prevent stale routing tables.

6 Evaluation Goals

There are many more questions we could study and analyze than we have space to present here. This section summarizes the research we chose to perform.

Do the DHTs work correctly? No. Mainline BitTorrent dead-ends its lookups 20% of the time and Azureus nodes reject half of the key store attempts. (See Sections 7.1 and 7.10.)

What is the DHT lookup performance? Both implementations are extremely slow, with median lookup times around a minute. (See Section 7.2.)

Why do lookups take over a minute? Lookups are slow because the client must wait for RPCs to timeout while contacting dead nodes. (See Section 7.3.) Dead nodes are commonly encountered in the area closest to the destination key.

What if the timeouts were eliminated? Lookup times would still be slow (typically 5-15 seconds). 10% of MDHT messages, for example, have round-trip times (RTT) of over 5 seconds. (See Section 7.5.)

Why are the routing tables full of dead nodes? Kademlia's use of iterative routing limits the ability for a node to opportunistically discover dead nodes in its routing table. (See Section 7.3.)

Do all nodes have the same observed RTT? Many nodes has sub-second RTTs with low variance, while most have much higher RTTs with noticably higher variance as well. (See Section 7.5.)

What if we used adaptive timeouts as suggested in Bamboo? Bamboo proposed using adaptive timeouts for a DHT using recursive routing. We evaluate their effectiveness in Section 7.6.

How effective is routing table maintenance? Azureus and Mainline are effective at finding nodes with an average of 10 and 4 hour uptime respectively, and the high layers of their routing tables are 80-90% live. (See Section 7.4.)

Does the network have connectivity artifacts? Both ADHT and MDHT observe one-way connectivity where 12% and 6% of nodes, respectively, send us messages but never reply once to our own messages. (See Section 7.11.)

What about bugs in the implementations? Azureus and Mainline both contain bugs that impact their performance. (See Section 7.1.)

What is the difference between MDHT and the Kademlia design? Mainline does not 'backtrack' if it encounters dead nodes and thus sometimes does not return the correct replication set. (See Sections 7.1 and 7.9.)

What are the differences between ADHT and the Kademlia design? Azureus contains a bug where it takes an hour to eject dead nodes. (See Section 7.4.) Furthermore, Azureus nodes will reject a key if they conclude they are not in the replication set. This conclusion is incorrectly reached about half of the time. (See Section 7.10.)

7 Measurements

To effectively measure MDHT and ADHT, we modified existing Mainline and Azureus clients to participate normally in the DHT, logging the events they observed. A single client will not see enough data to constitute a representative sample of DHT behavior. Instead of using many machines and running one client on each, we ran multiple independent instances of Mainline and Azureus on a single host and effectively performed a *Sybil attack* [13] giving us the same information without needing additional hardware. We collected about 16 days of measurements with 77 concurrent Mainline clients and about 23 days of measurements with 11 concurrent Azureus clients. For measuring routing table maintenance for Mainline, we used all 77 clients. For our other measurements, we use data from 11 clients.

7.1 Implementation bugs and changes

MDHT, as implemented by Mainline BitTorrent, has several curious bugs. The most serious is a routing table maintenance bug. When a contacted node fails to reply, the original Mainline blames an innocent node for the failure when the contacted node is not in the routing table. As a result, under normal operation, 85% of the nodes ejected from the routing table are not actually at fault. For our experiments, we repaired this bug, allowing us to better measure MDHT. When we do not refer to a specific version in our experimental results, we are referring to our *Fixed Mainline* implementation, as opposed to the *Original Mainline*. (We don't know if this bug is present in BitComet or other MDHT clients.)

A second serious bug occurs during a lookup, used to identify the nodes responsible for storing a key, among other things. Mainline performs iterative routing with 8-way concurrency, maintaining a priority queue of the 8 closest nodes which have yet been discovered. The lookup terminates when the 8 closest nodes have been contact or timed out. If, as often the case, some of these nodes are dead, Mainline does not backtrack to find the closest 8 *live* nodes; instead, the lookup terminates early. In our experiments, we observed a dead-end condition, where *all* 8 nodes are dead, on 20% of all randomly chosen queries. (Again, we don't know if other MDHT clients shares this bug.)

Finally, Kademlia specifies random queries, within each quiescent routing table bucket, to freshen the routing table. Mainline's implementation is incorrect; it considers a bucket active if a node has been added *or removed* within 15 minutes. Kademlia normally requires that *every* node in every bucket be pinged every 15 minutes. As a result, we would expect MDHT routing tables to have more stale nodes.

Azureus has a bug where it treats a remote node that regularly contacts it as alive even though it is unreachable. The routing table ping/replacement/refresh state machine likewise appears to be too optimistic about node liveliness; unless a node in the routing table is found to be dead in the course of a routing lookup or a regular refresh operation, Azureus will only explicitly ping it to verify liveness under uncommon conditions. This issue can be observed in our measurements of how long it takes for dead nodes to be removed from the routing table (see Section 7.4).

7.2 Lookup performance

Our first set of measurements concerns lookup performance on ADHT and MDHT. We performed 45k random lookups in ADHT and 26k random lookups in MDHT. A random lookup picks a key, at random, and locates the *k* nearest nodes to the key. In Figure 3, we show the CDF of the number of hops taken for each lookup. ADHT has a median of 26 hops and MDHT has a median of about 33 hops. ADHT is more tightly clustered because it uses a branching width of 16 so each hop matches at least 4 additional prefix bits. MDHT's branching width of 2 only guarantees one additional

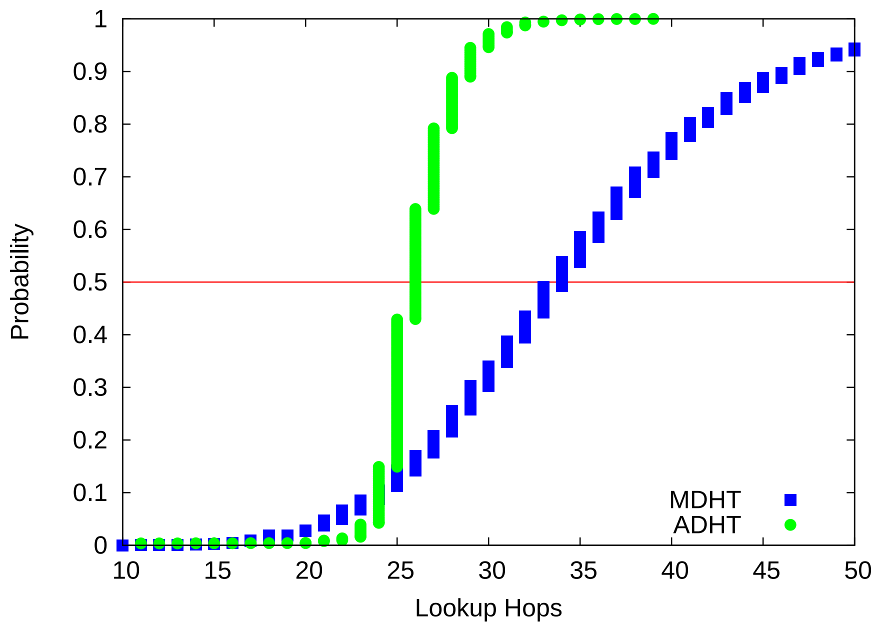

Figure 3: Hopcount CDF for random queries for MDHT and ADHT.

bit each hop. (In Section 7.9, we estimate similar numbers of nodes in each DHT, so population size differences do not figure into these graphs.)

Figure 4 shows the lookup latency on random queries. The performance is poor, with a median lookup latency exceeding a minute for both Mainline and Azureus, as a result of RPC timeouts contacting dead nodes. For both systems, 20% of the nodes contacted in a single lookup are dead on 95% of the lookups. Overall, about 40% of the nodes encountered are dead. Each dead node consumes one concurrent request slot for a 20 second timeout. Every 8 dead nodes encountered on an Mainline query will consume all the available slots resulting in a 20 second delay. This effect can be seen in the slope changes of the Mainline curve at 40, 60, and 80 seconds. Azureus uses the same timeout, but allows fewer concurrent requests, directly leading to the longer lookup latencies observed.

We wished to understand the potential lookup performance of these DHTs if the RPC timeout problem were eliminated, perhaps through shorter timeouts, more lookup concurrency, or fresher routing state. By estimating and artificially subtracting the timeouts, the resulting lookup times would improve to a median of 5 and 15 seconds for MDHT and ADHT, respectively. (See also Section 7.5.)

7.3 Location and causes of dead nodes

When a node performs an iterative DHT lookup, it maintains a queue of nodes as the search progresses(see Section 5.1). When that search encounters a dead node, we track

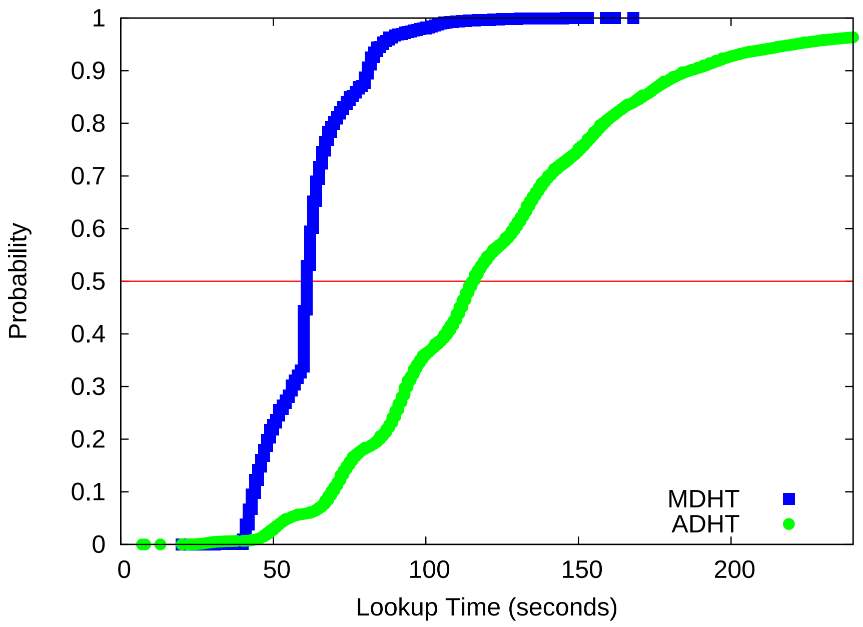

Figure 4: Lookup time CDF for random queries for MDHT and ADHT.

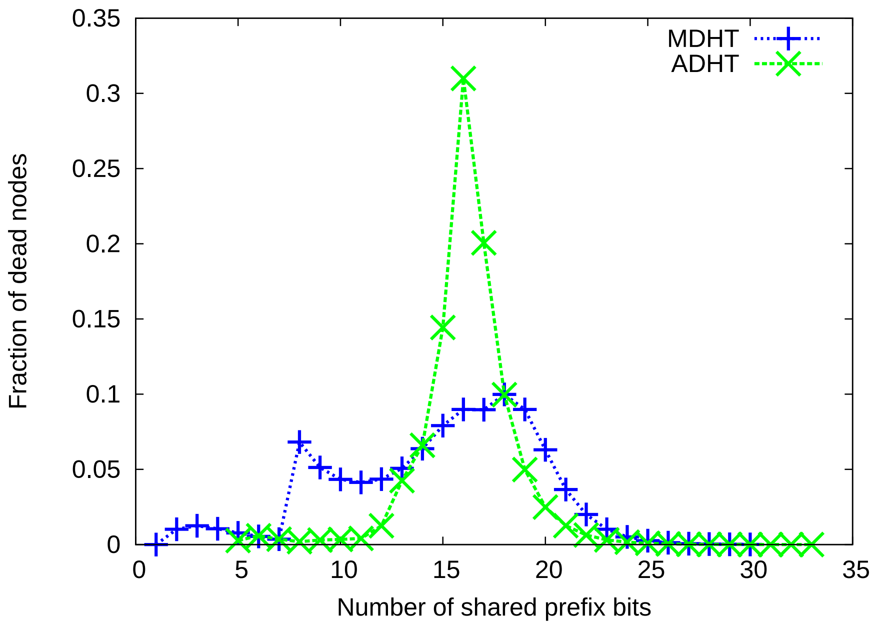

Figure 5: Histogram plotting the fraction of dead nodes encountered for different numbers of shared prefix bits.

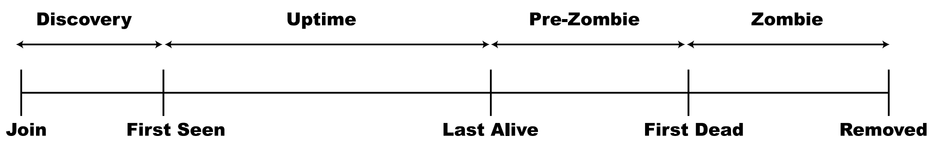

Figure 6: The stages in a node's lifespan.

	Bucket refr. per node/day	Churn per node/day	Discovery (min)	Uptime (min)	Prezombie (min)	Zombie (min)
Fixed Mainline	1373	493	56.89	258.69	12.73	3.43
Original Mainline	797	1110	8.90	79.82	8.08	1.61
Azureus	1662	1657	21.32	665.10	76.80	58.51

Table 1: Explicit bucket refreshes, node discovery, churn, and average node lifetimes for each DHT.

how many prefix bits it shares with the lookup key. Figure 5 shows a histogram of the dead nodes based upon the number of shared prefix bits. We can see that 75% of all dead nodes encountered in an ADHT lookup occur when the contacted node shares 15-18 bits of prefix, which corresponds to identifying the nodes in the replica set for the key.

This spike has two causes: First, the lowest levels of the routing table have more dead nodes; second, Azureus must contact and confirm the liveness of *all* nodes that are close enough to possibly be in the replica set. Mainline does not show a similar spike, most likely because it has a smaller replication factor and does not backtrack when encountering dead nodes at the end of a search. Instead, 20% of Mainline searches fail without finding the replica set.

To better understand the Azureus spike, we examined routing table liveness, layer by layer. For ADHT, the chance of a node in the routing table being alive drops from 70% for the first 3 layers to 20% thereafter. For MDHT, the first 15 levels of the routing table average an 85% chance of being alive. By layer 18, it drops to under 35%.

Clearly, both the MDHT and ADHT implementations have trouble keeping the lower-levels of the routing table populated with live nodes. A sensible solution would be to add more aggressive pinging on the extended routing table that neither MDHT nor ADHT implement. (See Section 4.)

7.4 Routing table maintenance

Any DHT design must deal with nodes that have limited lifetimes and unreliable behavior. In this section, we discuss how effective Azureus and Mainline are at discovering new nodes and ejecting dead nodes.

The lifespan of a node can be split into five events:

Join When a node first joins the system.

Seen When a node is first seen alive by us.

Last Alive When a node was last seen alive by us.

First Dead When a node was first found dead by us.

Removed When a node fails a few times and is removed from the table.

We refer to the four durations between these five events as the discovery time, uptime, prezombie time, and zombie time (see Figure 6). Nodes self-report their join times, so clock skew will create some error in the discovery time.

Some nodes die very young and we are unable to ever successfully contact them. We separately classify these infant mortality events: 13% of the nodes in ADHT and 57% of the nodes in MDHT suffer infant mortality and are not included in the subsequent figures.

We track nodes using our routing tables and Kademlia's normal node detection and elimination algorithms. Our measurements are based on nodes that are selected to be within our routing table. Neither Azureus nor Mainline selects nodes for inclusion in the routing table based on any criteria beyond its nodeID and reachability[6]. Typically, Mainline's routing table has about 150 nodes and the Azureus's has about 1000 nodes.

In Table 1 we show the average discovery time, uptime, and so forth of nodes in the Azureus and two Mainline clients routing tables. The difference between Fixed Mainline and Original Mainline is striking, with over twice the churn rate and a third of the expected uptime. Our bug repairs significantly improved Mainline DHT's ability to retain good nodes. The retention of more good nodes would lessen the need to find replacements for them, leading to the much longer node discovery time. Counterintuitively, fixing the bug also increases overhead. The high churn rate refreshes routing buckets without requiring explicit refreshes. The decrease in churn rate of about 600 nodes per measurement node per day is compensated for by an increase of 600 stale buckets instances for which Mainline performs a full lookup.

The uptime numbers we report exceed the 1-hour session duration reported by Saroiu et al. [44] or the 2.4 minute session length reported by Gummadi et al. [20]. This may be indicative of the stable nature of BitTorrent clients, which commonly run for hours to download large files.

7.5 Internet RTT measurements

In this section, we identify the cause of why median lookup performance would still be 5-15 seconds, even if timeouts were eliminated (discussed earlier in Section 7.2). We examine the effective RTT experienced by MDHT nodes contacting their peers through outgoing lookup and ping messages. Ping messages are sent to nodes in the local routing table. Lookup messages occur during random lookups and are sent to each node in the path.

In Figure 7 we show the complimentary CDF of the observed Internet RTTs for over a million lookup and ping messages done by both Fixed and Original Mainline. The

[6]Azureus has a BitTorrent extension to favor local nodes, on the same subnet, for high-speed sharing, but this has no impact on its Kademlia routing table.

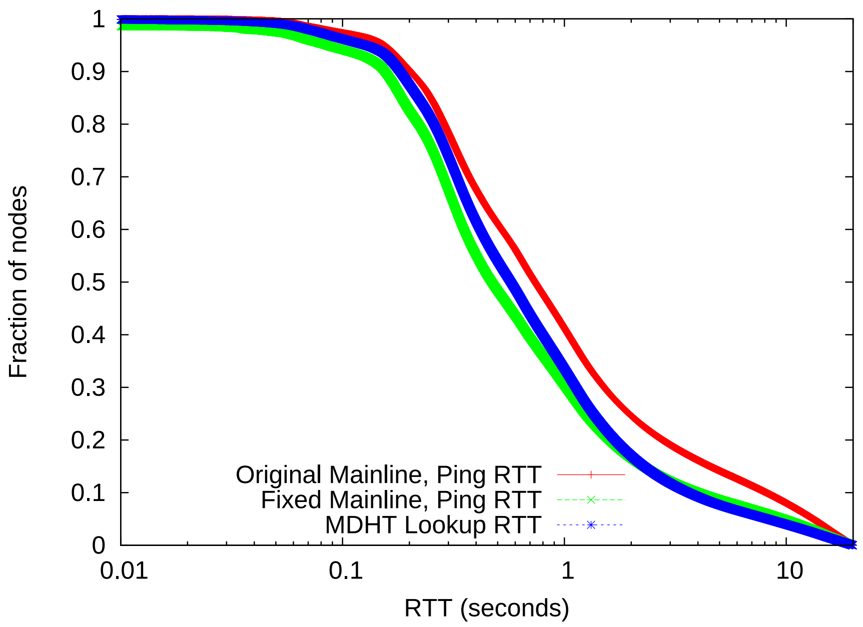

Figure 7: Complementary CDF of the probability that the observed RTT exceeds a threshold.

poorer quality of nodes in Original Mainline's routing table can be seen by comparing its ping RTT to Fixed Mainline. The MDHT lookup RTTs (i.e., RTTs for each node encountered during iterative lookups) are comparable to the Fixed Mainline ping RTTs. learned from other nodes on the network and sample remote routing tables, demonstrating that Fixed Mainline's routing table has a comparable distribution to other routing tables in the DHT.

The average MDHT response time for lookup messages was 1.6 seconds with 10% of hosts taking over 3.8 seconds to reply and 5% of hosts taking in excess of 8 seconds to reply. Overall, 59% of lookup messages received a reply.

Most of the RTT variation is from transient effects. This can be seen in Figure 8 as the difference between the minimum and average lookup RTTs for nodes that replied to at least 10 lookup messages. Even though most nodes have an average RTT of over a second, only 5% of nodes have a minimum observed RTT exceeding a second.

Not all of the nodes have the same RTT distribution. Of the 2.5M distinct nodes that replied to our lookup messages, we examine the mean (μ_{RTT}) and standard deviation (σ_{RTT}) of the RTT for the 19k nodes for which we had at least 30 replies. These popular nodes account for 2.1M or 20% of outgoing lookup messages and are much more likely to reply, with a 78.2% reply rate.

In Figure 9 we scatterplot μ_{RTT} versus σ_{RTT} for a random subset of the 19k nodes described above. Nodes with an average RTT under a second account for 42% of the peers in tight cluster of 'fast' nodes with a half second average RTT and a standard

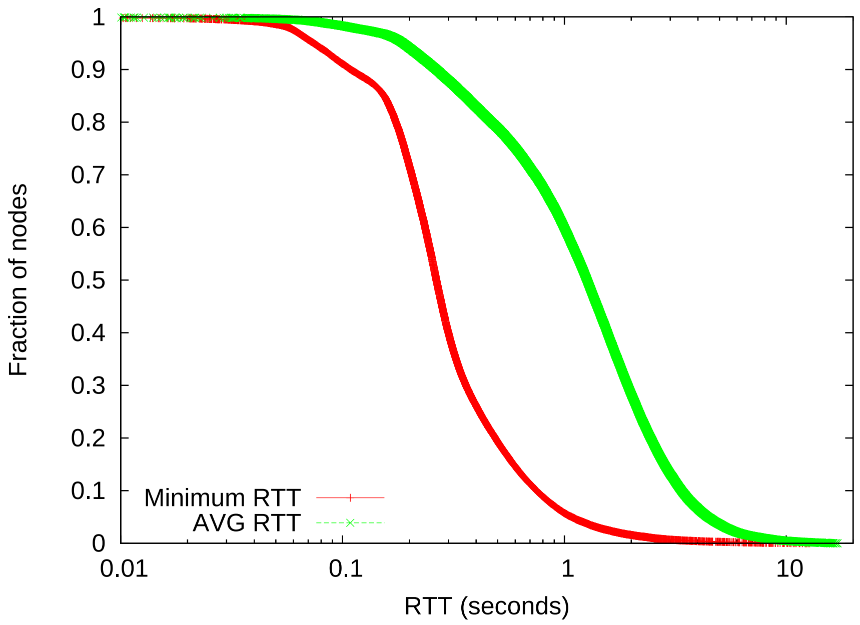

Figure 8: Complimentary CDF of Minimum and Average RTT for each host that replied to at least 10 MDHT lookup messages

deviation of .8 seconds. The remaining 'slow' nodes show a 2.1 second average RTT and a standard deviation of 2.8 seconds.

Overall, we observed a median RTT of 1.5 seconds, significantly more than the 100ms median Gnutella latency measured by Saroiu et.al. [45]. Directly comparing our measurements with the King dataset [21] is difficult, as 40% of our measurements exceeded their 1 second timeout limit. We conjecture that the latency we observe is because BitTorrent's tit-for-tat trading strategy encourages hosts to consume their upload bandwidth causing queuing delay and congestion. Furthermore, when a BitTorrent client is sharing files, bulk traffic must compete with lookup requests. Upload throughput could also be throttled by asymmetric links or traffic shaping devices. Perhaps future BitTorrent clients could implement techniques like TCP-Nice [47] to give preference to interactive traffic.

7.6 Timeout strategies

While more live routing tables and routes with fewer hops can reduce the odds of waiting on a dead node, and greater concurrency can sometimes hide this latency, another important tactic is to reduce the timeouts. Timeouts chosen based on the expected round-trip time can significantly reduce the latency for discovering dead nodes [5, 40]. The RTT can either be directly measured or estimated using network positioning algorithms like GNP [33] or Vivaldi [12]. Azureus already incorporates Vivaldi, but

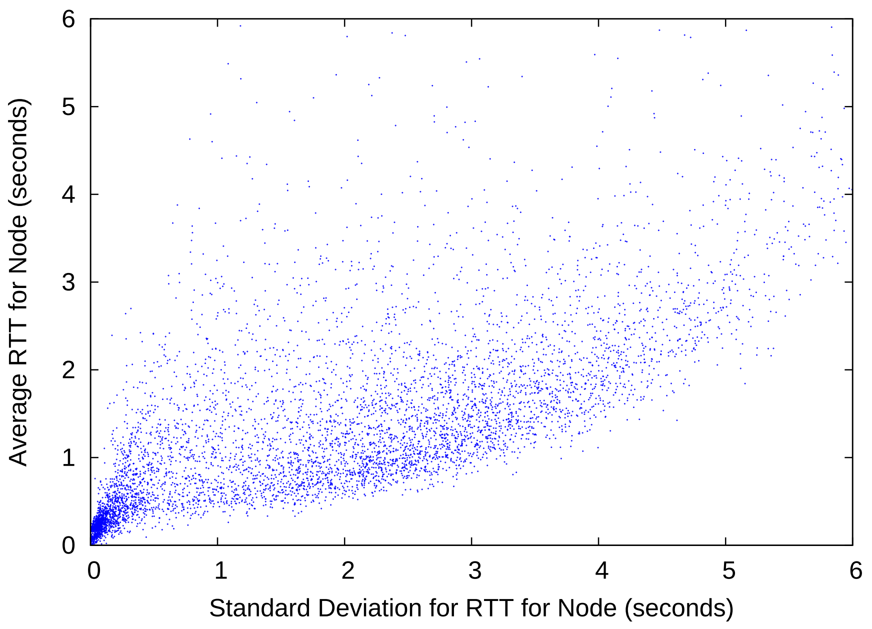

Figure 9: Scatter plot of RTT standard deviation and RTT average for 19k nodes from MDHT.

does not presently use it for this purpose. We further investigate the effect of different timeout strategies by using response times for MDHT lookup messages.

Table 2 compares the effectiveness several timeout strategies would have on the 2.1M outgoing messages sent to the 19k nodes who replied to at least 30 lookup MDHT lookup messages. The table shows the formula for computing the timeout, the average timeout calculated by the formula, the average response time, and the number false positives compared to the the baseline strategy **A** (i.e., live nodes that failed to reply before a more aggressive timeout caught them). For RPCs that fail or time out, they are considered in the response time column as if they had run for the duration of the timeout (i.e., timeouts bound the response time but can induce higher failure rates). The baseline strategy **A** has a response time, including errors, of 5.52 seconds and a failure rate of 21.8%. Strategy **C** was proposed by Bamboo [40] and inspired by TCP/IP, demonstrating the superiority of dynamic timeouts as it has half of the false positive rate of **B** while having a similar timeout duration and response time. Strategy **D** is a more aggressive variation on Bamboo's strategy, having an average response time of 1.22 seconds, but false positives on 9% of live nodes. This supports the strength of Bamboo's dynamic timeout mechanism.

Deploying these dynamic timeout strategies requires that the DHT use recursive routing, or if iterative routing is used, nodes must track the RTTs of nodes in their routing table to compute the necessary timeouts.

	Timeout Formula	Average Timeout	Response Time	False Positives
A	20	20	5.52	-
B	10	10	3.10	2.8%
C	$\mu_{\mathrm{RTT}} + 4 \cdot \sigma_{\mathrm{RTT}}$	8.8	3.03	1.4%
D	$\mu_{\mathrm{RTT}} + \sigma_{\mathrm{RTT}}$	3.1	1.22	9%

Table 2: Comparing timeout strategies. **A** represents the default Kademlia strategy. **C** represents Bamboo's adaptive strategy, where μ_{RTT} and σ_{RTT} represent average and standard deviation RTTs for the peer. All times are in seconds.

7.7 Spying on swarm sizes

Our measurement nodes participated normally in the DHT protocols, storing keys and responding to fetch requests. For the swarms where we were storing keys and acting as a tracker, we were able to monitor swarm membership and estimate the number of nodes in the tracked swarms. Our main ADHT dataset included fetch and store requests. Due to an oversight, our main MDHT dataset did not record those requests. In order to estimate the MDHT swarm sizes, we collected a new MDHT dataset over a 2 day period using 100 measurement nodes. In Figure 10, we plot a CCDF showing the fraction of swarms with at least a given number of nodes. These sizes are an *underestimate* of the true swarm size, because they only count visible nodes within that DHT. Azureus and Mainline clients, for example, are compatible with each other for the core tit-for-tat trading protocol, but are incompatible at the DHT layer. Likewise, older clients or clients with firewall issues might not connect to the DHT at all, despite participating in the core BitTorrent trading protocol.

ADHT swarms are much smaller than MDHT swarms because, in the default configuration, Azureus does not use the DHT for all torrents. BitComet uses the tracker in parallel with the DHT, and contributes the majority of the MDHT curve. We counted 3952 distinct swarms in our ADHT dataset and 386419 swarms in our MDHT dataset. Also, the distribution of swarm sizes follows a classic power-law distribution (i.e., the log-log plot is a straight line).

7.8 Flash crowds

Unlike MDHT, ADHT has provisions for handling nodes that are overloaded with requests or with stored data. ADHT will store values for a key in 10 alternate locations when the read rate exceeds 30 reads per minute or when the current node hits a global capacity limit of 64k keys or 4MB of storage. It also limits any one key to at most 4kB of storage or no more than 512 values, corresponding to a swarm with 512 members. Both types of migrations are rare. Approximately 2% of the time, one of our ADHT nodes was responsible for storing exactly one capacity-limit triggered migrated key.

Figure 11 plots the CDF of the inbound message rate bucketed into 5 second periods. The busiest 5 second period we observed is only four times busier than the median 5 second period. This is unsurprising, as BitTorrent clients are normally unsynchronized in their interactions with a tracker (whether central or distributed) and have a 30

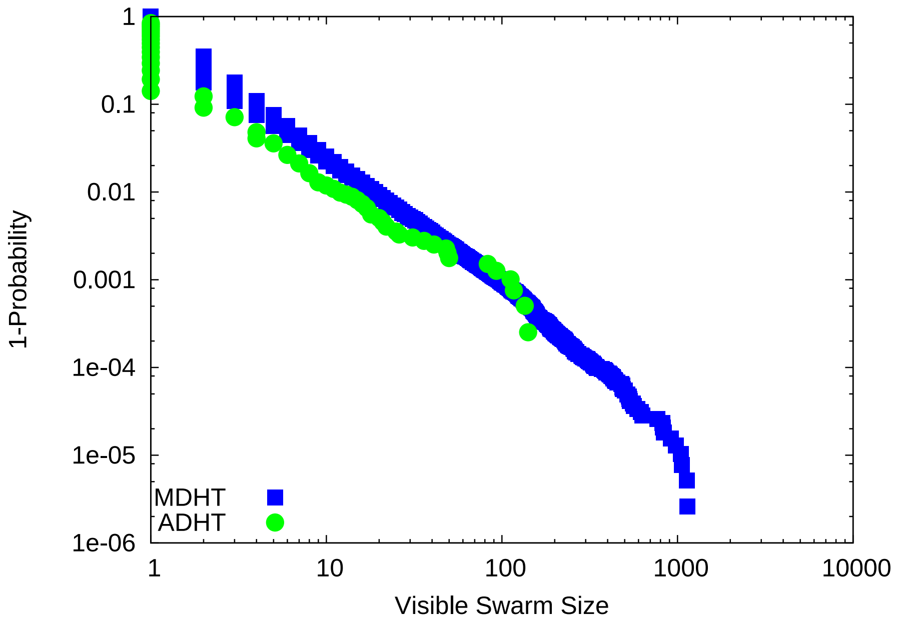

Figure 10: Visible swarm size CCDF (log-log scale).

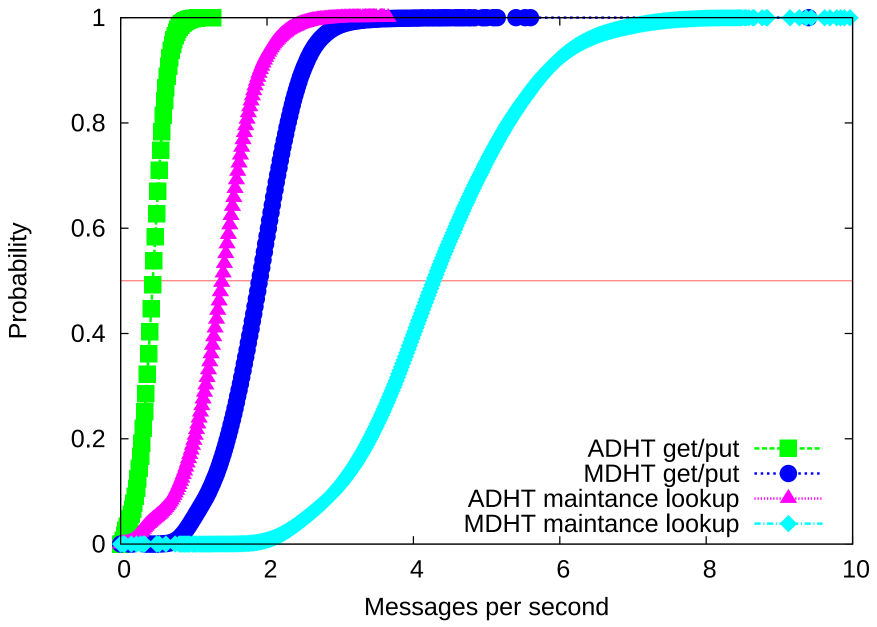

Figure 11: Incoming message rate.

minute refresh cycle. The 99.5th percentile busiest period that we observed has fewer than 4 applications messages per second plus 8 maintenance messages per second.

Both ADHT and MDHT consume very little CPU, storage, or bandwidth in their common usage. ADHT would need a swarm larger than 5000 nodes and MDHT larger than 360 nodes before the DHT application traffic would exceed 12 messages per minute. Figure 10 shows swarms this large, at least that are observable from the DHT, are very rare.

Worst case When an ADHT or MDHT peer initially joins a swarm, it first finds the replica set. To determine that the lookup is complete and no closer node to the lookup ID exists, the peer must send a find-nearest-k message to each node of a replica set and will receive a reply consisting of $30 \cdot k$ bytes of contact information (ID,IP address,port). If 100k people joined a swarm within five minutes, MDHT and ADHT would receive 300 find-nearest-k messages per second consuming 500kb/s and 1.25Mb/s sending contact information on 8 nodes and 20 nodes for MDHT and ADHT respectively in each reply. This may exceed the upload capacity of many asymmetric links and does not account for replies to DHT fetch requests as swarm members look for other swarm members.

If ADHT and MDHT nodes implemented the extended routing table, then any member of the replica set for a key *automatically* knows the other members of the replica set *and* knows that the find-nearest-k lookup is complete and can indicate so in its reply. Overall bandwidth consumption for this worst-case example would drop to 64kb/s for both ADHT and MDHT because only one member of the replica set needs to be contacted.

7.9 Estimating DHT size

We wish to estimate the size of the node population in MDHT and ADHT. Our size estimator, based on measuring nodeID densities, also makes a good test of the correctness of Kademlia's find-the-n-nearest primitive. For randomly chosen nodeIDs, the nodeID density can be expected to have a Poisson distribution with the inter-node distance exponentially distributed. Recall that an exponential distribution with rate λ has a cumulative density function (CDF) of $1 - e^{-\lambda x}$ and has mean=$1/\lambda$. If the measured inter-node distances are anomalous relative to this distribution, then the lookup algorithm is returning incorrect answers.

To estimate the density at any given lookup key, we route to it, recording the n closest nodes and their XOR distances. The differences in distances between the first and second closest node, the second and third closest node, and so forth form separate estimates of nodeID density. More formally, let L be a lookup key and N_i be the ith nearest node on a route lookup for L. One sample of adjacent-node spacing in terms of the fraction of the ID space covered is $D_i = \frac{(N_{i+1} \oplus L) - (N_i \oplus L)}{2^{160}}$, and the estimated size corresponding to that sample is $S_i = 1/D_i$.

If the number of nodes in the system is S, and the lookup returned only correct answers then all of the D_i will be an exponential distribution with mean $1/S$ and $\lambda = S$. However, if the lookup failed and there were missing nodes between N_i and N_{i+1}, then the distribution D_i will be anomolously large. It would be a mixture of exponential

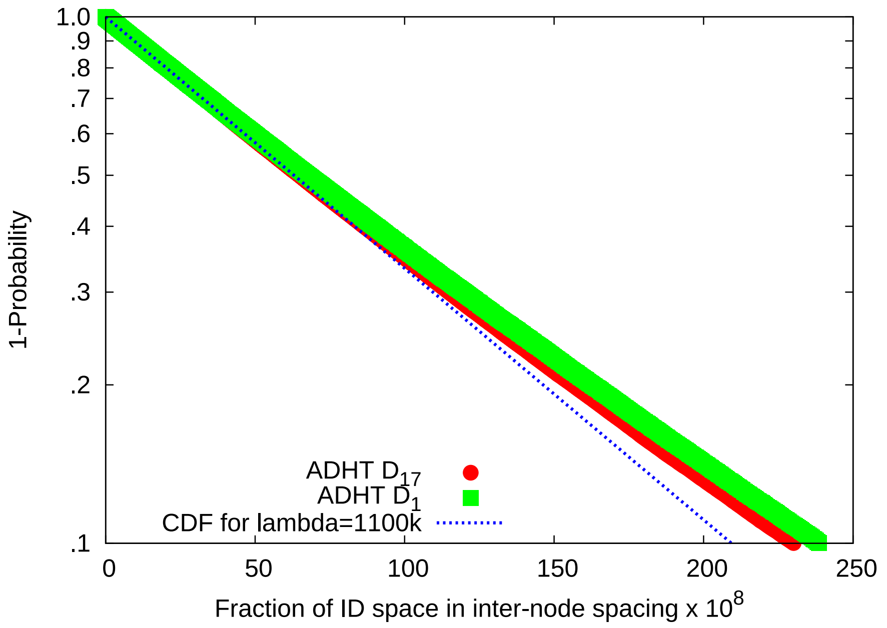

Figure 12: CCDF comparing ADHT's inter-node distance between nodeIDs D_i, D_{i+1} and an exponential distribution.

	Received requests from	Sent requests to	Sent and received request to&from	Sent and received and overlapping
Azureus	8.5M	9.3M	2.3M	404k
Fixed Mainline	15.5M	5.4M	1.3M	250k

Table 3: The breakdown of distinct hosts that we contact and that contact us.

distributions with means $1/S, 2/S, 3/S$, and so forth.

In Figure 12 we plot CCDFs of D_1 and D_{17} over 478k ADHT samples. The internode distance distributions D_i's overlap nearly completely, showing that the returned ith nearest neighbor is just as likely to be correct for small i as large i. This distribution closely matches an exponential distribution with $\lambda = 1100$k and demonstrates the strength of Azureus in properly finding the nodes closest to a given ID as well as allows us to estimate that ADHT has about 1.1M live nodes.

In contrast, Figure 13 plots D_1, D_4 and D_7 for 26k MDHT samples. We observe a different distribution and an anomalously large internode spacing for D_4, D_7, indicating missing nodes between the 4th and 5th nearest neighbor and between the 7th and 8th nearest neighbor. This demonstrates that Mainline, unlike Azureus, is not returning the correct answers to find-nearest-8 queries. We are unsure if this is caused by the smaller bucket size of MDHT or the backtracking bug in Mainline (see Section 7.1).

The lookup failures in MDHT make it more difficult to estimate the number of

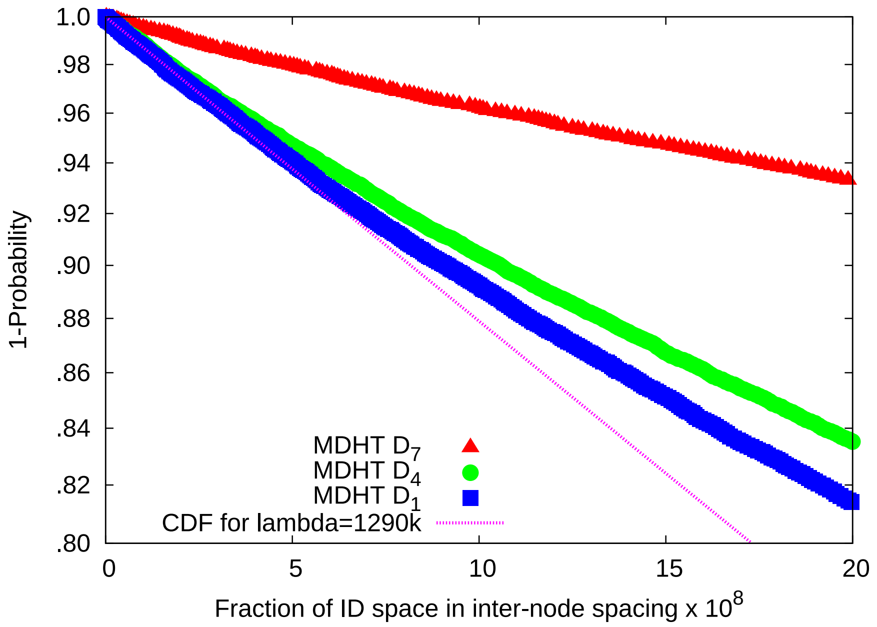

Figure 13: CCDF comparing MDHT's inter-node distance between nodeIDs D_i, D_{i+1} and an exponential distribution.

nodes. We filter the noise out of D_1 by keeping the 5k smallest samples in D_1 which closely match an exponential distribution and estimate $\lambda = 1300k$ by curve fitting. We thus estimate MDHT to have 1.3M live or recently dead nodes.[7]

7.10 Migrated and stored keys

Recall that MDHT does not do any kind of data migration, while ADHT does. Our ADHT dataset includes 1074k direct stores and 301k migrations. Each store typically is of only one key while each migration may include multiple keys. Key migration destinations are selected based on the local routing table and the destination node likewise validates the migration based on its local routing table. As a result of the large number of dead nodes the lowest layers of most routing tables, this check underestimates the amount of ID space for which a given node is responsible. We observed that about half of directly stored keys are rejected and 26.5% of migrated keys are rejected.

Not all rejected keys are rejected in error. ADHT has a million nodes and 20-way replication, so a key should share approximately $\log_2(10^6/20) \approx 15$ bits with the nodeID it is stored on. We conservatively identify a correctly rejected key if the key does not share at least 8 prefix bits with the nodeID it is being stored on. From this heuristic, at least 5% of stored and migrated keys are legitimately rejected, begging the

[7]An accurate count of only live nodes is difficult to determine because Mainline does not backtrack and remove dead nodes from its result set.

question of why at least 5% of key store requests are sent to the wrong host.

Azureus clearly needs a more accurate predictor of whether a given node is responsible for a given key. Chen and Liu propose techniques for enforcing routing consistency that may help [7]. As described in Section 7.8, an implementation of Kademlia's extended routing table would allow a node to implement a more accurate key rejection metric. Likewise, routing lookups have very good accuracy in Azureus (see Section 7.9), so lookups could also be used to determine the proper relica set.

7.11 Connectivity artifacts

Many nodes in the Internet are behind NAT or firewalls, possibly creating symptoms of poor or broken network connectivity. Some studies have estimated NAT prevalance by the network behavior of malware, with between 7% and 60% of worm infected hosts behind a NAT [38, 4]. And, of course, the Internet itself does not always guarantee connectivity between any two nodes.

Our test looks for hosts that contact us multiple times, yet we are unable to reach them, even after many attempts. This persistent unreachability may be the result of NATs or firewalls.

We define a host as k-unreachable if we fail to receive even one reply after k attempts to contact it. We define a host as j-seen if we observe at least j lookup requests from it.

Table 3 summarizes the breakdown of nodes that we sent and received lookup requests to and from. To compute the k-unreachability of various hosts from our experimental data, we need to find hosts that we have attempted to contact and that have sent requests to us at around the same time. We found 400k ADHT nodes and 250k MDHT nodes where the time interval that we attempted to contact them and the interval they contacted us is nontrivially overlapping, i.e., they contain at least one contact attempt and at least one message reception. We restrict ourselves to those nodes and to messages in the overlapping time interval.

In Figure 14 we plot the fraction of k-unreachable hosts over the total number j-seen nodes that were either reachable, or had at least k attempts to contact them. Approximately 15% of ADHT nodes and 8% of MDHT nodes that were seen alive at least $j = 15$ times were 6-unreachable and are probably unable to accept incoming connections.

Interestingly, the number of times a node is seen has virtually no correlation to the unreachability of nodes in MDHT. This likely because nodes that are contacted many times are likely to be in the routing table and Mainline always does a round-trip ping to verify reachability before adding a node to its routing table. There is only a minor correlation between the unreachability of nodes in ADHT and the number of times a node is seen. This implies that merely *seeing* a node says little about whether it is reachable.

While our results are not conclusive, our data suggest that 10-15% of BitTorrent users have significantly degraded Internet connections, likely resulting from firewalls or NATs.

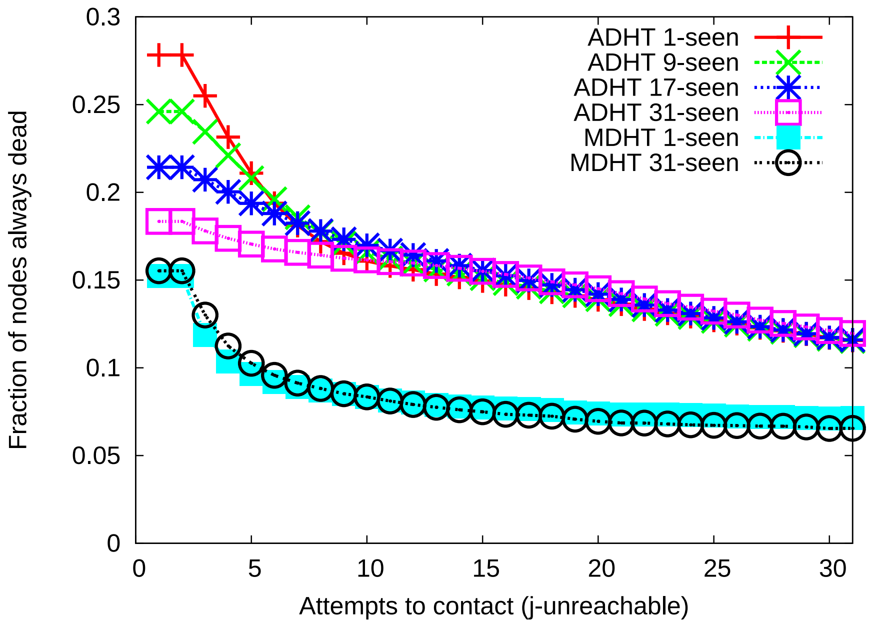

Figure 14: Unreachability ratio for Azureus and Mainline as a function of the host unreachability threshold (k) and times seen (j).

7.12 Communications locality

In general, DHT nodes tend to exchange most of their traffic with a small set of other DHT nodes. For Mainline, just 1.1% of a node's peers were sent at least 15 messages, and they account for a full third of the outgoing traffic. The 7.6% of peers that sent at least 15 messages account for 60% of the incoming traffic. Azureus is similar, with 1.6% of hosts receiving 15 messages responsible for 37% of the outgoing traffic and the 4.7% of hosts sending at least 15 messages responsible for 56% of the incoming traffic.

More interestingly, for Azureus, 54% of messages sent were sent to nodes that never replied once. For Mainline, 41% of messages were sent to nodes that never replied. For both clients, only 1% of the hosts that were never once seen alive are responsible for 28% of the messages sent to hosts that were never seen alive.

Unfortunately, it would be non-trivial to cut flakey nodes completely out of the DHT. If either Azureus or Mainline were to permanantly stop contacting a node after it failed to reply 15 times, only 10% of the contacts to dead nodes would be eliminated. Further, that same filter would cause false positives on live hosts that accounting for 2.5% of MDHT traffic and 8% of ADHT traffic.

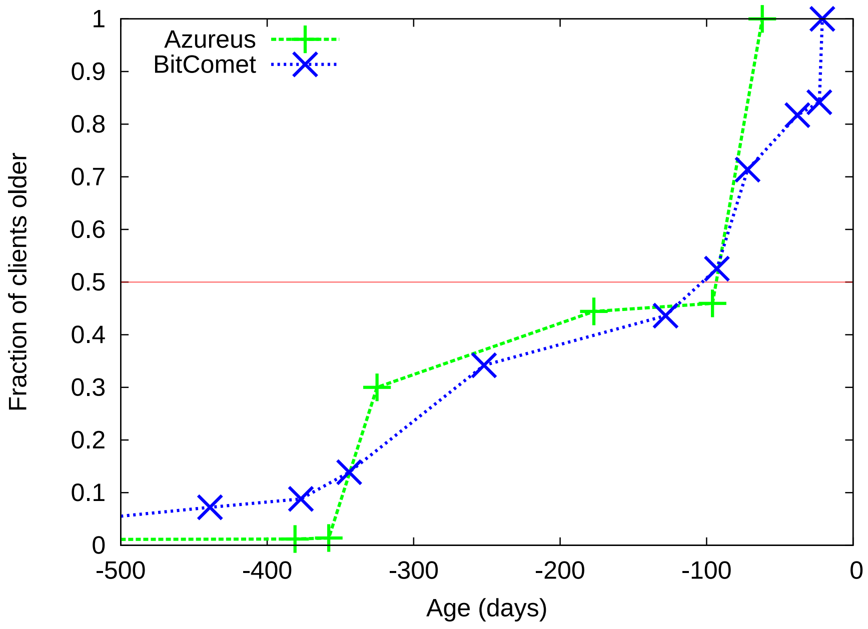

Figure 15: Client release ages for Azureus and BitComet.

7.13 BitTorrent client distribution

The distribution of BitTorrent client types can give us valuable information including the market share for each client program as well as how frequently users update their client software. If we want to roll out changes to any of the DHT protocols, this tells us about the size of our legacy issue. Helpfully, BitTorrent clients encode their client name and version number in their peer ID.

In Figure 15 we determine how often users update their clients by combining version numbers with release dates. We find that half of users are running software less than three months old. Unfortunately, a third of Azureus users run software more than 6 months old despite an auto-update mechanism. Furthermore, about 5% of BitComet users are running code over 18 months old. Clearly, no update can be expected to be universally adopted in a short timeframe.

Table 4 shows the client distributions for nodes in the MDHT, and an overall client distribution taken from directly joining the swarms for 100 large torrents. These data show that BitComet and Azureus together command $3/4$ of the market share for BitTorrent clients.

8 Conclusions

This study considered two large-scale DHTs with millions of nodes, used to help BitTorrent clients discover their peers. Although the performance issues we observed

Client name	Percent of hosts overall	Percent of hosts in MDHT
Sample count	31043	25353
Mainline	2.8%	2.8%
μtorrent	4.0%	7.3%
BitLord	15.0%	0.3%
Azureus	25.3%	n/a
BitComet	50.5%	88.8%
Other	2.5%	0.7%

Table 4: Popularity of different BitTorrent clients.

may only add a small overhead to hours-long BitTorrent downloads, they are still illustrative of the design and implementation challenges faced by large scale structured overlay networks in the real world. We found a number of problems inherent in the Kademlia-based overlays resulting from incorrect implementation of the Kademlia design, including a poor choice of DHT parameters and inherent network issues. Because the DHT largely serves as a backup for centralized trackers and there are two incompatible implementations, it is generally under-utilized, lessening the need for backward compatibility in future redesigns.

While some incremental changes could improve existing DHT use (such as increasing the number of concurrent lookups in Azureus's iterative routing), many of the problems we observe cannot be fixed while retaining compatibility. We therefore recommend an incompatible jump to a new DHT. While such a jump would make it attractive to go with a completely different DHT design, it would be perfectly acceptable for the new DHT to still use Kademlia but with many of the present problems addressed.

While it should be straightforward to repair many of the problems we discovered, security attacks are much more difficult to address. We suspect that a central authority for nodeID assignment would not be politically feasible, and NATs complicate any use of hashed IP addresses. As such, nodeID assignment remains an important open problem in p2p systems and would represent a open avenue of attack against the BitTorrent DHTs. Likewise, the BitTorrent DHTs are wide open to a variety of other security attacks. Addressing these security issues is future work, although the relatively small amount of data stored in the DHT suggest that techniques like Condie et. al.'s churning approach might be applicable [10].

Regardless, the limited use of DHTs as a backup peer discovery mechanism limits the damage that may be caused even by the most catastrophic attacks against the DHT. Once a BitTorrent peer discovers at least one legitimate peer in the swarm, whether from a central tracker or from the DHT, it may discover other peers through peer gossip (PEX). As such, the Kademlia DHT, even without aggressive security mechanisms, appears to be a good fit for the needs of BitTorrent peer discovery.

Our measurements clearly show that concurrent iterative routing can only deal with a limited number of dead nodes before incurring timeouts. To reduce these timeouts, nodes should preferentially refresh the routing entries corresponding to their local neighborhood and implement Kademlia's extended routing table. This will address the lower reliability of the lower levels of the routing, avoiding the surge in timeouts when finding all of the elements in a replica set, and providing proper support for key migration and replication. Improving Kademlia's iterative routing to give feedback on dead nodes, discovered later, to nodes earlier in the path will also increase opportunities to discover dead nodes. Opportunities for discovering live nodes may be increased by giving hints to later nodes on a search with nodes discovered earlier on the search path.

We have also identified a high variance in network RTTs, suggesting the benefits of using a dynamic timeout system, as in Bamboo, rather than the static timeout currently in use.

BitTorrent's many implementations clearly benefit from having the distributed tracker, running on a Kademlia DHT. This structure provides robustness when the central tracker is overloaded or otherwise unavailable. It also has the potential to eliminate the need for central trackers, entirely, improving the speed with which a BitTorrent client can find the nodes in a swarm and thus improving the startup speed of large downloads. Of course, DHTs have security concerns that are not faced by central trackers, and future work to address these concerns should have an impact both on BitTorrent and many other distributed systems.

References

[1] E. Adar and B. Huberman. Free riding on Gnutella. *First Monday*, 5(10), Oct. 2000.

[2] A. R. Bharambe, C. Herley, and V. N. Padmanabhan. Analyzing and improving BitTorrent's network performance mechanism. In *Proceedings of INFOCOM*, Barcelona Spain, Apr. 2006.

[3] P. Biondi and F. Desclaux. Silver needle in the Skype. In *Black Hat Europe*, Krasnapolsky, Netherlands, Feb. 2006.

[4] M. Casado, T. Garfinkel, W. Cui, V. Paxson, and S. Savage. Opportunistic measurement: Extracting insight from spurious traffic. In *Workshop on Hot Topics in Networks*, College Park, MD, Nov. 2005.

[5] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of DSN*, Philadelphia, PA, June 2004.

[6] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of OSDI*, pages 299–314, Boston, MA, Dec. 2002.

[7] W. Chen and X. Liu. Enforcing routing consistency in structured peer-to-peer overlays: Should we and could we? In *Proceedings of IPTPS*, Santa Barbara, CA, Feb. 2006.

[8] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: An overlay testbed for broad-coverage services. *ACM Computer Communications Review*, 33(3), July 2003.

[9] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.

[10] T. Condie, V. Kacholia, S. Sank, J. M. Hellerstein, and P. Maniatis. Induced churn as shelter from routing table poisoning. In *NDSS*, San Diego, CA, Feb. 2006.

[11] L. P. Cox and B. D. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proceedings of SOSP*, Bolton Landing, NY, Oct. 2003.

[12] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of SIGCOMM*, Portland, Oregon, Aug. 2004.

[13] J. R. Douceur. The Sybil attack. In *Proceedings of IPTPS*, Cambridge, MA, Mar. 2002.

[14] M. Feldman and J. Chuang. Overcoming free-riding behavior in peer-to-peer systems. *ACM SIGECOM Exchanges*, 5(4):41–50, 2005.

[15] M. Feldman, K. Lai, I. Stoica, and J. Chuang. Robust incentive techniques for peer-to-peer networks. In *Proceedings of the 5th ACM Conference on Electronic Commerce*, pages 102–111, New York, NY, May 2004.

[16] C. P. Fry and M. K. Reiter. Really truly trackerless bittorrent. Technical Report CMU-CS-06-148, Carnegie Mellon University, Aug. 2006.

[17] A. C. Fuqua, T.-W. J. Ngan, and D. S. Wallach. Economic behavior of peer-to-peer storage networks. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.

[18] S. Guha, N. Daswani, and R. Jain. An experimental study of the skype peer-to-peer VoIP system. In *Proceedings of IPTPS*, Santa Barbara, CA, Feb. 2006.

[19] 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 SIGCOMM*, pages 381–394, Karlsruhe, Germany, Aug. 2003.

[20] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of SOSP*, pages 314–329, Bolton Landing, NY, Oct. 2003.

[21] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: estimating latency between arbitrary internet end hosts. In *Internet Measurment Workshop*, pages 5–18, Marseille, France, Nov. 2002.

[22] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding, and X. Zhang. Measurements, analysis, and modeling of BitTorrent-like systems. In *Internet Measurement Conference*, Berkeley, CA, Oct. 2005.

[23] A. Haeberlen, A. Mislove, A. Post, and P. Druschel. Fallacies in evaluating decentralized systems. In *Proceedings of IPTPS*, Santa Barbara, CA, Feb. 2006.

[24] M. Izal, G. Urvoy-Keller, E. W. Biersack, P. A. Felber, A. Al Hamra, and L. Garcés-Erice. Dissecting BitTorrent: Five months in a torrent's lifetime. In *5th Annual Passive & Active Measurement Workshop*, Apr. 2004.

[25] S. Jun and M. Ahamad. Incentives in BitTorrent induce free riding. In *Workshop on Economics of Peer-to-Peer Systems*, pages 116–121, Philadelphia, PA, Aug. 2005.

[26] A. Legout, G. Urvoy-Keller, and P. Michiardi. Understanding BitTorrent: An experimental perspective. Technical Report 00000156, version 3, INRIA, Sophia Antipolis, Nov. 2005.

[27] A. Legout, G. Urvoy-Keller, and P. Michiardi. Rarest first and choke algorithms are enough. Technical Report 00001111, version 1, INRIA, Sophia Antipolis, Feb. 2006.

[28] 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 *Proceedings of INFOCOM*, Miami, FL, Mar. 2005.

[29] J. Liang, R. Kumar, and K. W. Ross. The FastTrack overlay: A measurement study. *Computer Networks*, 50(6):842–858, Apr. 2006.

[30] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free riding in BitTorrent is cheap. In *5th Workshop on Hot Topics in Networks (HotNets)*, Irvine, CA, November 2006.

[31] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of IPTPS*, Cambridge, MA, Mar. 2002.

[32] C. Ng, D. C. Parkes, and M. Seltzer. Strategyproof Computing: Systems infrastructure for self-interested parties. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.

[33] T. S. E. Ng and H. Zhang. Predicting internet network distance with coordinates-based approaches. In *Proceedings of INFOCOM*, New York, NY, June 2002.

[34] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent? In *Proceedings of NSDI*, Cambridge, MA, Apr. 2007.

[35] F. L. Piccolo, G. Neglia, and G. Bianchi. The effect of heterogeneous link capacities in BitTorrent-like file sharing systems. In *International Workshop on Hot Topics in Peer-to-Peer Systems*, pages 40–47, Volendam, The Nederlands, Oct. 2004.

[36] J. A. Pouwelse, P. Garbacki, D. H. J. Epema, and H. Sips. The BitTorrent P2P file-sharing system: Measurements and analysis. In *Proceedings of IPTPS*, Ithaca, NY, Feb. 2005.

[37] D. Qiu and R. Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. In *Proceedings of SIGCOMM*, pages 367–378, Portland, Oregon, Aug. 2004.

[38] M. A. Rajab, F. Monrose, and A. Terzis. On the impact of dynamic addressing on malware propagation. In *Workshop on Recurring malcode*, pages 51–56, Alexandria, VA, USA, Nov. 2006.

[39] S. Rhea, B.-G. Chun, J. Kubiatowicz, and S. Shenker. Fixing the embarrassing slowness of OpenDHT on PlanetLab. In *Proceedings of USENIX WORLDS*, San Francisco, CA, Dec. 2005.

[40] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2004.

[41] 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 *Proceedings of SIGCOMM*, Philadelphia, PA, Aug. 2005.

[42] J. Risson and T. Moors. Survey of research towards robust peer-to-peer networks: Search methods. IETF Internet Draft, Mar. 2006. `http://www.ietf.org/internet-drafts/draft-irtf-p2prg-survey-search-00.txt`.

[43] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM Middleware*, Heidelberg, Germany, Nov. 2001.

[44] S. Saroiu, K. P. Gummadi, and S. D. Gribble. Measuring and analyzing the characteristics of Napster and Gnutella hosts. *Multimedia Systems*, 9(2):170–184, 2003.

[45] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing system. In *Proc. Multimedia Computing and Networking*, San Jose, CA, Jan. 2002.

[46] I. S. Shelley Zhuang, Dennis Geels and R. Katz. Exploring tradeoffs in failure detection in routing overlays. Technical Report UCB/CSD-03-1285, EECS Department, University of California, Berkeley, 2003.

[47] A. Venkataramani, R. Kokku, and M. Dahlin. TCP-Nice: A mechanism for background transfers. In *Proceedings of OSDI*, pages 329–343, Boston, MA, 2002.