Reliability- and Capacity-based Selection in Distributed Hash Tables

Jonathan Ledlie, Jeff Shneidman, Matt Amis, Margo Seltzer

Abstract

Most current algorithms for distributed hash tables (DHTs) implicitly assume that all nodes participating in the DHT are homogeneous. However, empirical evidence indicates that participants are not: widely used traces show that the participants' network bandwidth varies by several orders of magnitude and that the average session length varies by as much as an order of magnitude. The mismatch between model and reality leads to inefficiencies and limits to scalability.

This paper makes two primary contributions in the pursuit of improving the scalability of DHTs. First, we show that selecting entries in routing tables according to their available uptime can decrease the number of maintenance messages by 42% when compared to the default proximity metric using a trace-driven simulation. Second, we demonstrate that when entering participants select their logical identifier so that the fraction of the ID space for which they are responsible more closely matches their fraction of the total system bandwidth, low-bandwidth nodes obtain a 35% improvement in productivity when nodes perform random block downloads on a 256-node network.

1 Introduction

Original distributed hash table (DHT) designs generally assumed that all nodes were created equal, in terms of their capacity, reliability, and network position. Systems built with this underlying assumption exhibited poor *stretch*, the ratio of overlay distance to Internet distance. To address the issue of stretch, researchers added proximity to the basic algorithms, either explicitly, as in Pastry [6, 30], or optionally, as in Chord [18, 32, 37]. Proximity-based lookup and routing address the heterogeneity that exists in network position, but do nothing to address the heterogeneity that exists in node *reliability*, the amount of time a node is participating in the system, or *capacity*, the amount of network bandwidth available to the node.

Because some nodes have significantly shorter session times than others, they are responsible for a proportionally greater number of maintenance messages. When all nodes' uptimes are assumed to be uniform, the entire system becomes limited by the session lengths of the least reliable nodes and scalability suffers. If, instead, we favor long-lived nodes in routing tables, we reduce the total number of maintenance messages and improve scalability. In a similar manner, when some nodes participate over low-bandwidth links, the performance of these nodes limits system efficiency when other nodes wait on these overly-busy, low-bandwidth nodes. We address this problem by relaxing logical identifier selection, allowing lower bandwidth nodes to service a smaller fraction of the total ID space.

Our research divides into two main sections, each of which addresses one of the limitations described above. First, we show that current uptime is an effective predictor of future uptime. We then incorporate the advertised uptime of a node into our routing table selection algorithm by having nodes use the node most likely to remain up instead of the most proximate. Because routing table entries stay in the system longer, maintenance messages decrease by 42% when compared with the default proximity metric in our experiments.

Next, we examine an alternative to proximity-based logical identifier selection [26] that strives to distribute load in a manner proportional to each participant's bandwidth. Each entering node estimates the fraction of the system bandwidth that it contributes and then selects its ID from among a small number to position itself so that it is responsible for a similar percentage of the system's resources. In our experiments, this results in a 35% increase in the amount of work low-bandwidth nodes can accomplish and a 20% increase in system throughput.

Table 1 summarizes how our reliability- and capacity-based techniques fit into the current set of DHT design criteria.

The rest of this paper is organized as follows. In the remainder of the introduction, we present the DHT model used throughout this study. Section 2 examines selecting routing table entries by incorporating an uptime metric. Section 3 examines selecting logical identifiers to better match a node's responsibility to its capacity. We then present results from a FreePastry implementation [13] that has been modified to perform this ID selection running on a 256-node NetBed network [35]. Section 4 places these results in context, discussing related work and Section 5 concludes.

DHT construction	Description
Proximity Neighbor Selection	Choose entries in routing table based on proximity (*e.g.,* latency). Has largest effect on reducing stretch. Abbreviated PNS.
Proximity Route Selection	Select route "on the fly" for a given key and static routing tables. Requires that multiple routes exist to a target. Has minimal stretch improvement beyond PNS.
Proximity Identifier Selection	Select logical identifier based on physical location, attempting to make physical neighbors logical neighbors. Problems: complicates load balancing and placement of redundant copies of blocks because neighbors have correlated failures.
Reliability Neighbor Selection	Choose entries in routing table based on reliability (*e.g.,* uptime). Can provide proximate and reliable neighbors when used with PNS.
Capacity Identifier Selection	Select logical identifiers based on bandwidth capacity, giving nodes domains (*i.e.,* workloads) commensurate with capacity.

Table 1: Previous work has shown how to do routing, neighbor selection, and ID selection on the basis of *proximity*. We propose using other metrics, like reliability and capacity, which we analyze in this paper. Other metrics could include trust, anti-proximity (for ID selection, making correlated failures of redundancy sets less likely), and topical similarity. The default Chord algorithm does not use Proximity Neighbor Selection, but it has been applied to Chord and evaluated by Gummadi [18] and Zhang [37]. Hildrum evaluated several proximity metrics for Tapestry [19]. Ratnasamy explored proximity-based ID selection in CAN [26].

1.1 Model

We assume a generic ring- or tree-based DHT model, a generalization of Pastry [30], Tapestry [38], and Chord [32]. We expect that the reader is familiar with the basic properties of DHTs and common implementations; please refer to the references just mentioned for an overview. We limit ourselves here to a discussion of the characteristics of DHTs that are most important to this work.

- Several, usually many, nodes are available for most entries in each node's routing table, *i.e.,* neighbor selection is possible. Note that CAN's hypercube design does not permit neighbor selection [25].
- Routing tables are not symmetric; that is, if node B is in node A's routing table, node A is not necessarily in B's.
- Data objects are divided into equal-sized blocks (*e.g.,* using erasure codes [34]) and blocks are the unit of storage and transfer: no node is storing unusually large files.
- Node IDs and block keys are generated uniformly at random on a continuous integer namespace (*e.g.,* $(0 \ldots 2^{160}]$) using a hash function.
- Each block has a redundancy set of identical copies that are stored on l logical neighbors of a block's *authority node* or *root*.
- The set of blocks for which a given node is *root* is that node's *domain*. We are concerned with two domain types:

  counterclockwise Domain ranges counterclockwise from the node's ID to its predecessor's ID (Chord-like).

  nearest node Nodes are the root for all blocks where theirs is the closest ID, extending both clockwise and counterclockwise (Pastry-like).
- Each node knows its own bandwidth and has an approximate value for the total system bandwidth.
- We do not consider caching of blocks within the DHT, path convergence, or higher, application-level caching.

2 Reliability Neighbor Selection

Neighbor Selection refers to the process by which nodes are entered into routing tables. Pastry, Tapestry, and Chord can use neighbor selection since most routing table entries can be occupied by any one of a number of possible entries. For example, the furthest finger in Chord can select from about half of the nodes, the penultimate from one quarter, and so on. Selecting any node in the range provides the same hop count guarantees. In a large system, these ranges will include many nodes, so, generally, a subset of the nodes is selected before a particular node is singled out to be placed in the routing table. Previous work examined what happens if the most proximate node is chosen from the sample [6, 18, 37]. In this work, we examine selecting a node based on a combination of its proximity and the likelihood that the node will remain responsive for a long period of time.

For the purposes of this paper, we equate *uptime* with *reliability*. *Reliability* is a general term that could incorporate uptime, link quality (drop rate), number of peers that already linked to that node (in-degree), gross uplink bandwidth, and other measures on a utility curve. While we are examining the contribution of these other components as part of our ongoing work, in this section, we limit ourselves to the analysis of uptime. Before evaluating Reliability Neighbor Selection in trace driven simulations in Section 2.1, we make several observations about Internet node behavior.

1. Choosing routing nodes based solely on proximity can actually lead to meaningfully slower routing, depending on the application and node lifetime characteristics.

In networks where a significant portion of nodes have short lifetimes, the insertion of these nodes into routing tables leads to increased message delays and increased maintenance message overhead (especially in a self-tuning system [20]). The effect of increased message delays occurs when an application attempts to route to a dead node; the message will be lost and after a timeout, typically on the order of a few seconds, retransmitted. Increased maintenance overhead is due to detection and removal of dead routing nodes. If the message overhead is significant, the capacity of the system is reduced.

Selecting slightly less proximate, but more stable, nodes for entry into a routing table may be appropriate for systems with high churn or interactive user applications on networks where the time saved avoiding timeouts and re-transmissions dominates the additional stretch. The intuition behind this statement is that although a packet may travel a longer distance, it does so on more reliable nodes, so the longer stretch is dominated by savings in timeouts and re-transmissions.

2. It is possible to predict node stability based on previous performance.

In order to take advantage of uptime, we need to be able to predict which nodes are likely to remain responsive for a long time in the future. It has been shown that periods of inactivity are good predictors of future inactivity [2, 17] and that a file's current lifetime is a good predictor of its total lifetime [14]. We hypothesize that current uptime should be a good predictor of future uptime. In order to validate this hypothesis, we examined a Gnutella trace[1] collected by Saroiu *et al.* in May 2001

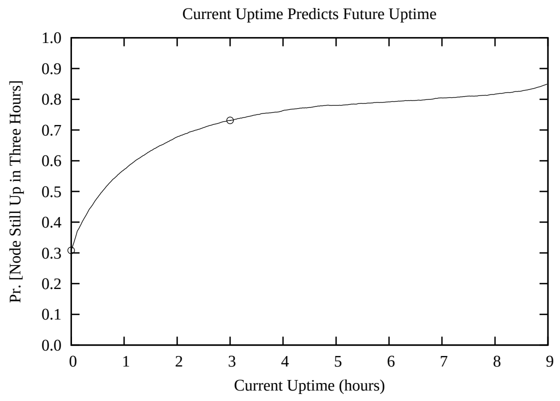


Figure 1: Example of how current uptime is a good predictor for future uptime. For all sessions that had been up for at least *current uptime* hours, we computed the percentage that continued to be up for another three hours. A node just entering the system at time 0 has only a 30% chance of being up for three more hours, whereas a node that has been in the system for three hours tends to remain up for three more hours 73% of the time. To avoid measurement bias, nine hours is the most we can show on the x-axis and remain accurate.

[31]. We confirmed that current uptime is, in fact, a good predictor of future uptime and demonstrate one example of this correlation in Figure 1 drawn from the data. Assuming that these data are representative, the way to select long-lived nodes for inclusion in a routing table is to select those nodes that have already been up for a long time. Also note that this correlation means an exponential, memoryless distribution (a common model in existing research) is not appropriate to model expected lifetime in a p2p system, at least one that exhibits behavior similar to that seen in Gnutella.

3. Long-lived nodes tend to have higher bandwidth capacity to absorb higher in-degree.

Favoring long-lived nodes for inclusion in routing tables will lead to such nodes having a higher in-degree than other nodes. In order to determine if this phenomenon might be detrimental to the system, we examined the Gnutella traces for a possible correlation between long uptimes and high bandwidth. We sorted nodes by average uptime and then divided this list into quartiles. Then, with each quartile, we plotted a cumulative distribution function (CDF) of the bandwidth; this plot is shown in Figure 2. The figure shows that long-lived nodes consistently have more bandwidth available than short-lived nodes. When we split the average uptime into deciles, the spread was more evident (the resulting graph, however, was too cluttered for inclusion here). We performed the same analysis on the down-

[1]A discussion of how we analyzed the Gnutella trace and an oddity in the data is in Appendix A.

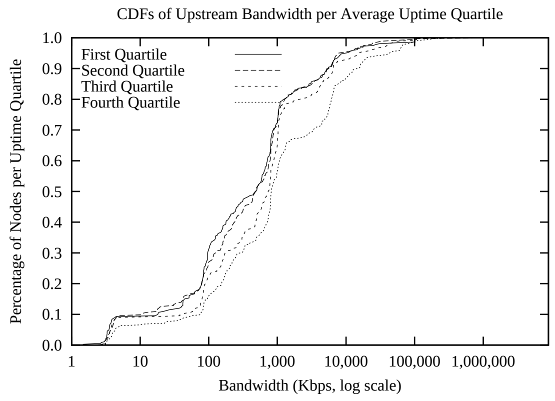


Figure 2: Correlation between average node uptime and upstream bandwidth. Nodes have been divided into quartiles based on their average uptime. The CDF of each quartile's upstream bandwidths have been plotted in kilobits per second.

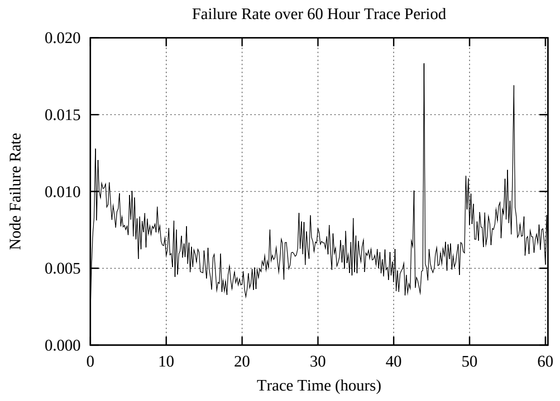


Figure 3: Gnutella trace failure rate. Node failure rate is the number of deaths per the number of alive nodes measured in ten minute intervals.

stream data and found the same correlation. This implies that these long-lived nodes should be capable of supporting the additional message forwarding burden created by a high in-degree.

2.1 Neighbor Selection Experiments

In order to quantify the potential benefit from selecting routing table entries based on uptime, we conducted a simulation study using a modified SimPastry incorporating self-tuning routing table probes as described by Mahajan [20]. Our version of the simulator uses node uptime as the routing table entry selection metric. When considering a node for insertion, it is queried with a *load probe* to determine its current in-degree. If the node reports its current in-degree to be too high, the node is added as a stopgap in the routing table until a suitable replacement can be found via a passive replacement process. Load snapshots are cached by each node and expire after 30 minutes. One drawback of the SimPastry simulator is that message losses due to network overload are not modeled; only losses due to node failure are captured in our work and that of previous research whose results we use as a comparison [7, 20]. Part of our on-going work is to add message loss to our neighbor selection experiments.

We selected the Gnutella trace workload [31] for our experiments because it allows for a comparison with previous research, and because it is a real workload. The trace characteristics that make the uptime metric perform favorably are those where many nodes enter and leave rapidly, while some set of long-lived nodes also exists. The uptime metric is less useful when nodes are generally long-lived and stable. The node failure rate of the Gnutella trace[2] is shown in Figure 3.

The Gnutella workload does not include topology information. We used a topology of 600 routers created by the Georgia Tech transit-stub topology generator [36], as has been used in previous research [7, 20]. Each node in our simulation was assigned to a router at random. The simulator sends 140 messages per minute from random nodes to random keys. As in Mahajan *et al.*, the simulator self-tuned to achieve a loss rate of 1%.

2.1.1 The Benefit

In Figure 4, we show the performance through the maintenance cost necessary to retain approximately a 1% loss rate. We plot messages/second/node at ten minute intervals for each routing table determinant: proximity and uptime. For this workload, the uptime-based selection yields a 42% improvement. Note that at the start of the trace, the uptime metric is uninformed and is temporarily bested since uptime data is meaningless at the beginning of the simulation.

2.1.2 The Cost

There are two possible "costs" to achieving this improvement in message overhead: a high in-degree for long-lived nodes, and a longer per-message routing time since we no longer select the most proximate neighbor.

[2]For simulation tractability reasons, we culled the starting set of the Gnutella trace at random so that the number of nodes in the system ranged from $400 - 1800$, instead of $1300 - 2700$. We are currently running our experiments on the whole trace for consistency with previous research though our initial results indicate our graphs will remain largely the same. All graphs in the section are generated with the smaller dataset.

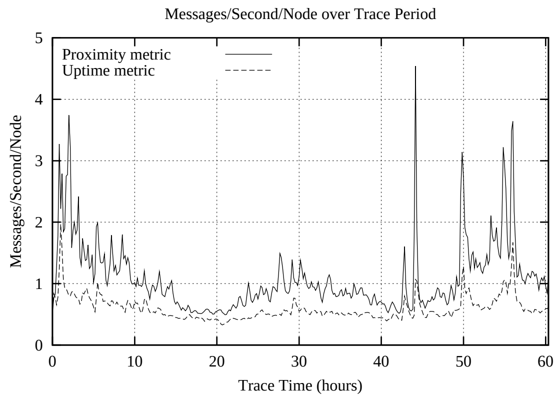


Figure 4: Rate of maintanence messages/second/node over the trace period (lower is better). The uptime metric exhibits an average improvement of 42% over proximity.

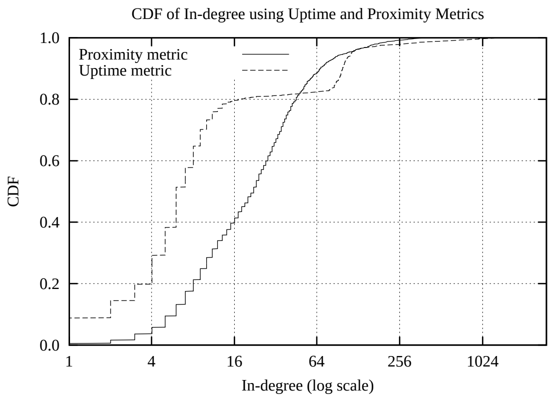


Figure 5: CDF of node in-degrees recorded over the trace period. Both distibutions show that some nodes have very high in-degrees. The spike in uptime metric in-degrees after 100 is from new nodes that have not found a replacement for the node they are pointing to yet.

Metric	Stretch
Uptime	2.74
Distance	1.83

Table 2: Mean stretch, the ratio between overlay distance traveled and network distance, for the two experiments shown.

Note that if every participant could select the most stable node for entry in their routing table, the in-degree on stable nodes could be very high. As described above, we have implemented a back-off mechanism using *load probes*, similar to the idea proposed by Castro *et al.* [7]. This additional overhead, included in the data of Figure 4, is negligible. For the experiments in this section, we set the maximum attempted in-degree to 100. As the system runs, nodes will detect overloaded nodes and back off. This behavior is visible in Figure 5. Figure 5 also shows that this uptime metric with capped in-degree produces similar results to the in-degree produced by the existing Pastry algorithm. As the back-off technique can be made more aggressive, we conclude that imbalance of in-degree does not appear to be a significant problem.

Longer per-message routing time is also a concern. However, Table 2 shows that the mean distance ratio for the uptime metric is probably acceptable for most applications, and indeed, cannot be any worse than Pastry without proximity information. In general, we might equate distance with transmission time. However, Table 2 and SimPastry both fail to capture the time lost in retransmitting lost packets. Instead of trying to save maintenance message overhead as shown earlier in this section, we could have kept the same level of maintenance and reduced the message loss rate, thus leading to even better results for the uptime metric. As mentioned earlier, including delays due to message loss is ongoing work.

3 Capacity Identifier Selection

"From each according to his ability, to each according to his needs." [21]

As in most systems consisting of unequal parts, overallocating work to under-provisioned nodes in DHTs yields unfairness and low throughput. We begin by showing what happens in today's systems when there is no attempt made to limit a node's work to its capacity. We then show the benefit of using capacity-based ID selection as a mechanism to reduce a system's ratio of workload to ability, increasing both fairness and throughput.

3.1 Capacity Ratio

We define *capacity ratio* as the ratio between a node's assignment of work (its logical domain size) and its ability to do work (its bandwidth). Note that while we use bandwidth as the one-dimensional metric for ability to do work, this analysis can be extended to include other node attributes that affect its ability to do work, such as its processing power, available memory, etc. A node's capacity ratio ρ is:

$$\rho = \frac{\%\, namespace}{\%\, system\, bandwidth} = \frac{d}{D} \Big/ \frac{b}{B}$$

where d is the size of the node's logical domain, D is the size of the entire logical namespace, b is the node's bandwidth and B is the bandwidth of the entire system. Blake has shown that the primary bottleneck in DHTs is

Node	A	B	C	D	E
BW (MB/s)	5	20	20	20	20
Example $\#1$: Max. Throughput 25MB/s					
Domain Size	.2	.2	.2	.2	.2
Cap. ratio	3.40	.85	.85	.85	.85
Usage (MB/s)	5	5	5	5	5
Usage (%)	100%	25%	25%	25%	25%
Example $\#2$: Max. Throughput 53MB/s					
Domain Size	.094	.226	.226	.226	.226
Cap. ratio	1.60	0.96	0.96	0.96	0.96
Usage (MB/s)	5	12	12	12	12
Usage (%)	100%	60%	60%	60%	60%

Table 3: Example $\#1$ shows the maximum throughput with uniform IDs. Example $\#2$ shows how the throughput doubles if node A's capacity ratio is cut by $3/4$ using ID selection. This is the change in capacity ratio we see in the simulations in Section 3.3. Both examples assume a random workload.

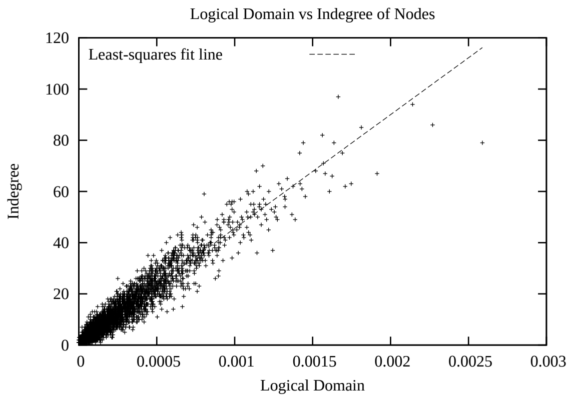


Figure 6: Correlation between logical domain in-degree in the counterclockwise domain type (Chord-like). The simulation consisted of 4k nodes; nodes used 12 fingers each. Intuitively, large domains result in large in-degrees because nodes tend to point somewhat uniformly around the ring, and a node with a large domain size is absorbing more of these pointers.

likely to be bandwidth, so eliminating other factors like CPU or disk size is a reasonable simplification [3]. As ρ approaches one, the node's work assignment approaches the fraction of the domain size for which it is responsible. When $\rho \gg 1$, the node is most likely a bottleneck, and if $\rho \ll 1$, the node is being under-utilized. Not only will overburdened nodes respond sluggishly: due to timeouts, they may also appear to flit in and out of existence to other nodes, necessitating changes in their routing tables, redundancy sets, and logical domains. Giving nodes work proportional to their capacity makes the system more stable.

As a metric, capacity ratio helps to isolate which nodes will be the first to overload as system utilization increases. Conversely, decreasing the capacity ratio of these nodes allows greater system utilization. Table 3 contains an example.

The key question to be addressed is whether workload is directly proportional to domain size. A node's resources are devoted to four tasks: (1) servicing lookup requests when the node appears in some nodes' routing tables, (2) servicing requests for data for which it is the root, (3) servicing requests for data when it is a replica and the data's authority is unavailable, and (4) performing non-DHT related work. The first three of these quantities correspond directly to domain size in the general case, when random key lookups originate from random nodes. For Chord, a strong correlation exists between a node's logical domain size and its in-degree. We created a simple Chord simulator that assigned nodes ids $(0 \ldots 1]$ at random, and then for each node found its domain size and computed how many nodes' fingers refer to it, that is, computed its in-degree. Figure 6 shows the results for a 4096 node topology. If, in addition, we assume a random proximity distribution, the same high degree of correlation exists between domain size and in-degree in Pastry: the exact distribution depends on the actual proximity distribution and on what neighbor selection metric is used. For both domain types, if a node is the lookup's root, it will be at least somewhat involved in the download of the data item: if the item is cached, it must originally be fetched from the root; if a load balancing scheme like Roussopoulos and Baker's is used, communication with the root is still required to find the redundancy set [29]. The number of blocks copied as part of updating nodes in the redundancy set is similarly proportional to domain size. The larger the domain size for which a node is responsible, the more messages the node will be expected to handle, whether as an intermediate or final destination.

It might appear that the distribution of capacity ratios for a system with randomly chosen IDs should correspond fairly closely to the distribution of bandwidths. Because the average domain size is $1/N$, one might expect that most nodes would have a "$\%\ namespace$" numerator close to $1/N$. However, this is not the case. Figure 7 shows a CDF of the counterclockwise distance between nodes when their IDs are chosen at random. This illustrates the distribution of counterclockwise (Chord-like) domains; nearest-node (Pastry-like) domains follow the same pattern. This is one of the reasons that the concept of virtual servers was introduced in earlier work on Chord [10, 33].

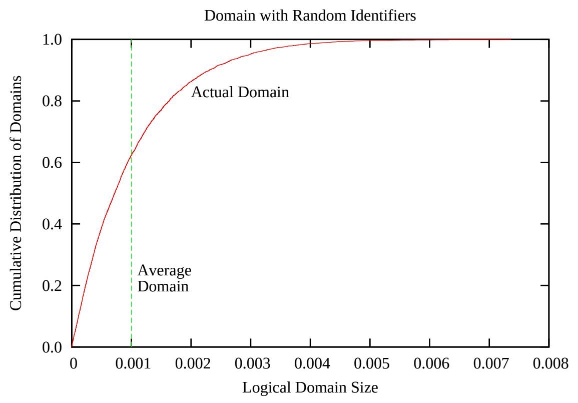


Figure 7: Domain Size generated by random identifiers when 1000 identifiers between 0 and 1 are chosen. The figure aggregates five trials for clarity. The largest value is 0.0074, *i.e.,* $7.4 \times 1/N$.

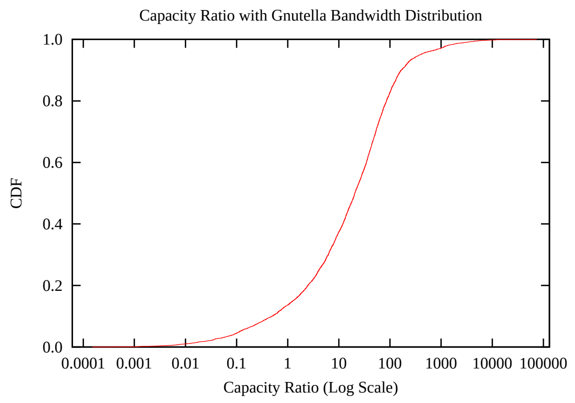


Figure 8: Capacity ratios if nodes following a Gnutella bandwidth distribution were to join a DHT. Due to the non-uniformity of domain sizes, capacity ratios exhibit an even broader spread than the base bandwidth distribution.

In this simple case we can derive some basic theoretical results. For a Chord-like domain, where IDs are chosen uniformly at random on a circle of circumference 1, the maximum domain size is at least lnN/N with constant probability, and the minimum domain size is at most $2/N^2$ with constant probability. More generally, the expected number of domains of size a/N is $N(1 - a/N)^N \approx N\mathrm{e}^{-a}$, and standard techniques (such as martingales) can be used to show that the number of domains of size a/N is sharply concentrated around its expectation. For more related results, see [4]. These results demonstrate the large variety in domain extent when IDs are chosen at random.

This distribution of domain sizes affects capacity ratio. It implies that low bandwidth nodes that fall on the steep section of the curve in Figure 7 (above 80%) will have enormous ratios: they will be extremely under-provisioned. To illustrate the spread of capacity ratios, we took the uplink bandwidth data from the Gnutella trace and assigned each node an ID, as if it were participating in a DHT. We then computed its capacity ratio. The results are shown in Figure 8. While the specific distribution depends on the run, this figure portrays a representative set of results. The largest capacity ratio in the figure is more than 70,000.

3.2 ID Selection Mechanism

We use logical ID selection to achieve good capacity ratios. The identifier selection mechanism allows each node to choose its logical ID, k, from a set, K, of possible choices, where $|K|$ is a well-known, constant. If x is the original, node-specific input into a collision-resistant hash function used to create a node's ID, we simply add the integers $0, \ldots, |K| - 1$ to x to produce $|K|$ unique IDs:

$$K = \{k_0 = h(x + 0), \ldots, k_{|K|-1} = h(x + |K| - 1)\} \quad (1)$$

where K is the logical ID set for a given node, k_i is its ith ID possibility, and h is a collision-resistant hash function such as SHA-1. Each node therefore has a pseudo-random, easily-verifiable set, K, of possible IDs. We assume that h works like a random number generator and is capable of producing pseudo-random numbers in a large space, *e.g.,* 2^{160}, and hence is responsible for the pseudo-randomness of K. To verify that a node is using a valid ID, k, another node simply has to check that there exists some $i < |K|$ such that $k = h(x + i)$, since x is well-known.

After generating the set of IDs, a node must choose one from among them. Our mechanism for deciding is a cost function that depends on the domain type. For counterclockwise, Chord-like domains:

$$c \;=\; |b_s - d_{sf}| + |b_a - d_a| - |b_s - d_{sn}| \quad (2)$$

For nearest-node, Pastry-like domains:

$$\begin{aligned} c \;=\; & |b_p - d_{pf}| + |b_a - d_a| + |b_s - d_{sf}| + \\ & |b_p - d_{pn}| - |b_s - d_{sn}| \end{aligned} \quad (3)$$

where c is the cost, a is the node joining, p and s are its potential successor and predecessor, respectively, b_x is the percentage of node x's bandwidth compared to the total system bandwidth, d_{yf} is the potential fraction of the domain node y would have in the future if a performed the join, and d_{yn} is its fraction now. Each cost is an evaluation of the potential future minus the current situation. To discover these values, a joining node a contacts its potential predecessor and successor at each ID

$k \in K$. It is assumed that an approximation for the system bandwidth is a well-known quantity, one that could be aggregated and transmitted using an epidemic protocol [23]. Once these values needed for the cost function are discovered, the node computes the cost of each ID and joins at the one with minimal cost.

3.3 Analysis and Simulations

Before evaluating ID selection experimentally, we simulate how it would alter the domain distribution when applied to the Gnutella trace. Without loss of generality, we assume a counterclockwise domain. Nodes joined serially and online, that is, in a decentralized manner without sharing information about each of their sets of potential IDs. The first node joined at position 0 and the remaining nodes selected $|K|$ IDs at random, evaluated their cost using Equation 2, and joined at the location with the lowest cost. The results of are plotted in Figure 9. The figure shows a decrease in the 95th percentile from ≈ 750 to ≈ 200 by $|K| = 16$ and then more gradual decreases for larger values for $|K|$. This behavior is expected because the Gnutella bandwidth distribution spans six orders-of-magnitude (set Figure 2) and ID selection cannot produce a perfect alignment of IDs in the online case. However, as shown in Table 3, a decrease of $3/4$ in capacity ratio can lead to a significant increase in potential system throughput.

We are in the process of analyzing the general theoretical case as part of our on-going work. In the general case, the free parameters are the distribution of bandwidths, or weights $w \in W$, and the number of IDs, or hash functions, $|K|$ allowed. The result is the distribution of w/d, where d is the logical domain. With an infinite number of hash functions, it can be easily proven that all capacity ratios become one in the offline, non-distributed case. The on-going analysis is examining how close the distribution can be with an infinite number of IDs when decisions are made online, how domain size changes with fewer and fewer IDs, and what effect churn has on the distribution.

3.4 Experiments

To examine how identifier selection performed in an actual networked scenario, we created a version of Pastry that was capable of selecting its ID during the join process and ran it on the NetBed experimentation environment [35]. Our primary goal was to measure changes in fairness and throughput with several values of $|K|$ using a large topology.

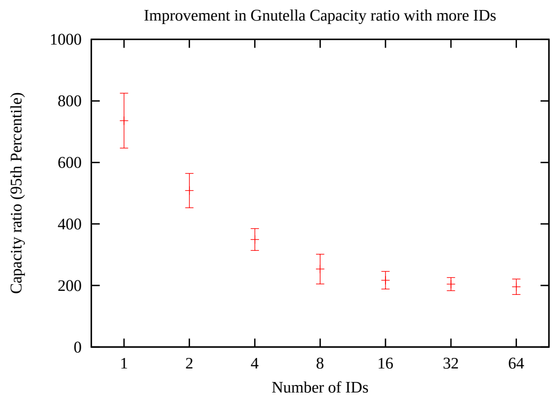


Figure 9: Mean 95^{th} percentile of distribution of capacity ratios as nodes are allowed to sample more IDs. When the number of IDs,$|K|$, equals 1, this is the default ID selection mechanism: there is only one choice. Each result is an average of 50 trials. Error bars denote the standard deviation.

3.4.1 NetBed and FreePastry

To create an implementation of Pastry that used ID selection, we started with the FreePastry implementation from Rice University [13]. This implementation is written in Java. It runs in one of three ways: by having all nodes exist on one machine and communicate over a virtual network, by having Pastry instances exist on different physical nodes and communicate via RMI, and by having distinct instances run on different physical nodes and communicate via Java's Nonblocking I/O interface. We exclusively used and modified this third method of communication; thus, all traffic between nodes is via the network. We added the ID selection process to node join: each node chose $|K|$ IDs at random, used its bootstrap node to locate the successor and predecessor of each ID, and computed the cost for each ID, as in Equation 3. The lowest cost ID was then selected and the node join progressed normally from this point on. When $|K| = 1$, nodes did not go through these steps at all and joined exactly as in the original FreePastry implementation.

We created a simple application using the Common API that FreePastry exports [11]. This application joined the network using a bootstrap node and then waited at a barrier until all other nodes had joined the system. This barrier was necessary because without it there was often too much cross traffic for low bandwidth nodes to successfully join. After passing the barrier, each node began generating lookups for random keys. The barrier functionality is part of the NetBed framework and does not cause any network activity on the interfaces we monitored for the experiment. Queries were marked successful after the destination had finished sending the source

In	Packets	Octets
Virtual nodes	4571 (1382)	3485k (1075k)
Real nodes	4550 (1478)	3493k (1135k)
Out		
Virtual nodes	4674 (1385)	3546k (1081k)
Real nodes	4779 (1549)	3623k (1184k)

Table 4: Virtual node validation. Final packet and octet counts per node over three trials. Averages are given and standard deviation is in paretheses

an 8KB block, representative of a file block created by an erasure code.

Networking in FreePastry sends messages via UDP if they are less than 64KB and contains queues for outgoing and incoming UDP packets and TCP streams. With the default settings, which we used, it queues up to six UDP packets and up to 256 TCP messages. If messages cannot be sent using UDP, they are moved to the TCP queue. If both queues are full, the messages are dropped.

NetBed is an extremely flexible experimental environment. We selected it over PlanetLab [24] because it allowed us to specify the capacity of each link explicitly.

3.4.2 Virtual Node Validation

NetBed typically works by having real PCs configure themselves according to an NS-like input file. Each node is its own PC and traffic shaping (*e.g.*, bandwidth limiting, packet loss) is also performed with one PC per link. The NetBed staff has recently developed virtual node (vnode) traffic shaping for the purpose of running large-scale experiments. Virtual nodes (vnodes) run in FreeBSD jails [16]. Users are still limited to a maximum 4×100 MB/s per node because all inter-machine communication is over real links, which currently have this physical limitation. Our experiments typically used 288 vnodes running on 98 PCs.

Because the technology is new, we cooperated with NetBed operations to validate the traffic shaping. We ran duplicate experiments on topologies that differed only in the respect that one ran entirely on real nodes and the other entirely on vnodes. The experiment shown was run on a 24 node topology where 16 nodes had 10Mb/s bandwidth and 8 had 1MB/s. Each node ran FreePastry and joined the network serially. We set $|K| = 1$ in both cases because we were not evaluating ID selection behavior. After every node joined, each node performed lookups for random keys for 10 minutes, initiating each lookup immediately after the previous one finished. Using *netstat*, we recorded the network activity on each node. Table 4 shows the average and standard deviation seen for all of the nodes over three runs of each topology. Because of the similarity of these two sets of numbers, we believe that the topology in our experiments would have functioned the same if it had been run entirely on real nodes. Real nodes have been validated previously by the NetBed staff [35].

3.4.3 Results

We discuss two representative sets of experiments. The first set, run on a smaller topology, allows us to illustrate change in domain sizes on a node-by-node basis. The second shows the change in fairness and throughput when ID selection is applied to a large topology.

The first set of experiments exhibit how a small number of low bandwidth nodes can produce drag on the entire system. The topology consists of 64 nodes total, 56 with bandwidths of 40 MB/s and 8 with 0.4 MB/s. There was no loss built in to the topology, but nodes did drop packets due to buffer overflows. As noted above, nodes joined serially, reached a barrier, and then began queries once every node has a routing table. Each node initiated a new query immediately after completing the previous one. If a query timed out, nodes would initiate a new query. We set this timeout to be 60 seconds. Because of buffer overflows, it would frequently appear to low bandwidth nodes that members of their routing table were dead, initiating even more maintenance traffic that they could not handle. The cummulative effect of low bandwidth nodes being extremely busy is shown in Figure 10. It shows how using ID selection improves capacity ratio by one order of magnitude, resulting in a dramatic increase in completed lookups for both the low bandwidth nodes (increased fairness) and for all nodes (increased throughput). We do not quantify these changes because it is a small topology and a short workload period.

While nodes did not crash in the 64 node experiment, when we scaled the experiment up to 256 nodes, they did. We found that with query rates in the range of .5 second per node, which the high bandwidth nodes could achieve, the low bandwidth nodes would occasionally deadlock ($2 - 4$ per trial). Instead of setting an arbitrary threshold on query rate, we wanted to have the nodes perform lookups as fast as they could without extreme overload occurring. To do so, we added a lookup throttle:

- Each node had its own query rate and a timeout = query rate $+10$ seconds.
- If a query was completed before the timeout, the query rate would decrease by one second.
- If a query did time out, the rate would increase by one.

	Completed lookups	
BW (MB/s)	$\lvert K\rvert = 1$	$\lvert K\rvert = 16$
.4	4341	5879 (+35%)
1	16672	20217 (+21%)
4	24025	29537 (+23%)
40	23331	26224 (+12%)
All	68370	81858 (+20%)

Table 5: Change in fairness (+35%) and throughput (+20%) on a heterogenenous 256 node topology. The data show the total number of completed lookups for the 64 nodes of that bandwidth in a one hour period averaged between two trials. There are 64 nodes of each bandwidth type. Lookups intervals are throttled to avoid crashes due to extreme overloading.

We placed a lower bound on the rate at one second because we knew the low bandwidth nodes could not handle a faster rate. Query rates began at five seconds and increased to $35 - 70$ seconds by the end of the trial. We instituted the throttle for the larger second experiment. Because we did not observe a clustering of when nodes initiated queries, we did not further model query rates around a Poisson distribution.

The large topology experiment consisted of 256 nodes, 64 nodes each of four bandwidths: 40Mb/s, 4Mb/s, 1Mb/s, 0.4Mb/s. The purpose was to see how ID selection performed in a heterogeneous topology: if it could both increase the number of blocks low bandwidth nodes could download and the total number of the system. Again nodes joined serially and began lookups after all had a routing table. Nodes used the throttle mechanism to limit the number of queries they originated to the level they could successfully process. Table 5 shows the total number of lookups completed by bandwidth type averaged over two trials that consisted of one hour of lookups after all of the nodes had joined. All nodes used one of the 40MB/s nodes as their bootstrap. As a result, they were frequently in other node's routing tables and had a higher message routing workload. This is why their completed lookups are fewer than the 4MB/s nodes. As expected, the average number of hops was a just less than 2, with minimal variance. The main experimental result, however, is that a 20% improvement in throughput suggests that ID selection is a worthwhile mechanism to add to future DHT designs.

3.5 Security Issues

Though we have shown logical identifier selection to be useful in improving fairness and throughput, it does introduce the possibility of a potential reduction in system security. As Castro *et al.* have shown, a large class of attacks on peer-to-peer systems involve the abuse of logical identifiers [8]. Given a sufficient number of identifier choices, a collection of conspiring nodes can cut target nodes off from the rest of the system, prevent access to certain data, or even partition the system.

Such attacks fall into the category of Sybil attacks [12], a general class that relies on obtaining a large enough sample of identifiers to allow conspiring nodes to probabilistically join in a certain security-threatening arrangement. Using logical identifier selection gives conspiring nodes $|K|$ times as many identifier choices to work with. Castro *et al.* suggest using a certificate authority to prevent faulty nodes from obtaining too many ID choices. This technique could be used with logical identifier selection to make it difficult for conspiring nodes to obtain too many ID sets, K.

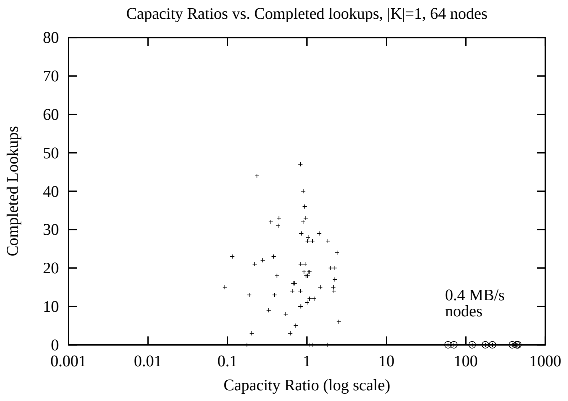
It is also feasible that identifier selection could be used to combat faulty nodes. By giving each normal node in the system $|K|$ possible IDs, it is harder for conspiring nodes to "target" a certain node, since that node can move somewhere else if it senses it is being cut off.

A number of our cost functions rely on nodes calculating and reporting certain values regarding their own behavior, such as their bandwidth, their physical location, or their average uptime. This poses a trust issue in a system containing faulty nodes. While left for future work, non-malicious, rational nodes could be motivated to truthfully reveal these characteristics through distributed algorithmic mechanism design [15].

4 Related Work

Previous work closely related to this paper falls into two general categories: (1) proximity-based improvements to DHTs and (2) load balancing in DHTs. Although others have suggested considering bandwidth and other node attributes in identifying desirable neighbors [6], to the best of our knowledge, this is the first work to explicitly consider metrics weighted more heavily towards reliability than proximity.

Gummadi provides an elegant taxonomy of proximity-based improvements to DHTs, discussing which geometries can support Proximity Neighbor Selection, Proximity Route Selection, and Proximity ID Selection [18]. In particular, they show how a tree design like Pastry and a ring design like Chord are equivalent when the tree's branching factor is one (normally it is 2 or 4, leading to fewer hops). Because Proximity Neithbor Selection results in the largest stretch improvement, it has been applied to Pastry, Tapestry, and Chord. Castro *et al.* show that stretch in Pastry can reach as low as 1.4 in certain topologies [6]. The techniques introduced here achieve a different goal: Instead of attempting to reduce stretch, our goal is to improve scalability by ensuring that low-



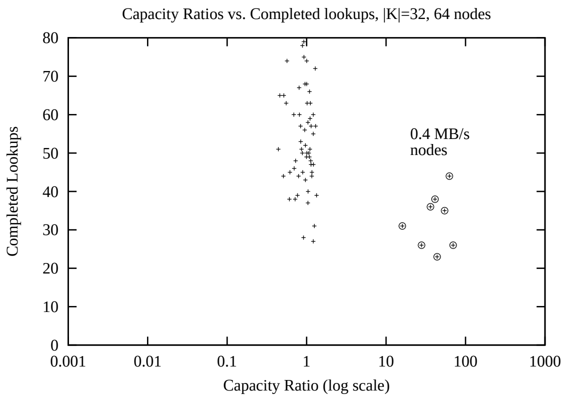


Figure 10: Scatterplots illustrating how using ID selection causes all nodes' capacity ratios to become closer to one, allowing them to download more blocks. This experiment contained 64 nodes, 48 with a bandwidth of 40MB/s and 8 with 0.4MB/s. Low bandwidth nodes are circled in the plots. Each node initiates a query immediately following the completion of the previous one with a timeout fo 60 seconds. The experiments ran for ten minutes. When $|K| = 1$, many low bandwidth nodes are not able to download one block, frequently throwing the Java Socket exception "No buffer space available" and initiating extra maintenance traffic, spuriously finding other nodes to be dead.

bandwidth nodes do not become system bottlenecks.

Zhang *et al.* portray a piggyback method to find neighbors to choose from in Chord [37]; this same mechanism could be used to provide input into our neighbor selection technique. Topologically-sensitive CAN examines using landmarks to generate logical IDs that closely match the physical topology [26]. The drawbacks of this approach are correlated leafset failures, if redundant copies are held on logically adjacent nodes, and a non-random logical density distribution, as other work has discussed [6, 9, 18]. None of these earlier works examined metrics other than proximity.

Several techniques have been developed to improve load balance in DHTs. Dabek *et al.* describe "virtual servers" that allow multiple independent DHT instances to run on a single machine. The system defines a lowest common denominator node, and machines that are idle can start up more independent copies until they are busy. Each instance uses a "virtual tag" to generate its node id, but there is no selection process. Byers *et al.* describe a load balancing technique that hashes data to be stored using two distinct hash functions, providing two potential locations [4, 5]. The less loaded of the two possibilities is chosen. During data lookup, the query must contact both possible storage locations, or appropriate forwarding pointers must be used. The method of Byers *et al.* is an example of " the power of two choices," described more fully in [22]. Our ID selection process is similar in spirit, in that we also use multiple hash functions, although here we do so to place the servers in a more appropriate fashion.

ID selection bears an interesting relation to work of Roussopoulos *et al.* on load balancing in p2p networks [29]. They develop a cooperative request scheme where nodes direct requests toward the highest capacity replica. They assume that the source of each lookup is aware both of the capacity of each possible replica holder. Sources of requests learn the replicas by first contacting the *authority node* (*i.e.,* a key's primary storage node). Storage nodes can periodically update their capacity through Controlled Update Propagation [28]. Our two schemes for load balancing are complementary: ID selection reduces an overburdened node's domain, preventing it from being contacted in the first place, and their prevents it from being contacted frequently after the replica set is known.

5 Conclusion

We examined two new ways to incorporate heterogeneity into DHT design. In our experiments, each method, reliability-based neighbor selection and capacity-based ID selection, showed promise and appear worthwhile additions to DHTs in general. As applications for large scale distributed storage and computation grow, we foresee a whole design space worth exploring. For example, banks would use *trust* as a metric to direct some traffic over trusted nodes and the rest over the Internet. Just as RONs promoted application-driven metrics on small-scale networks [1], DHTs can use proximity, reliability, and capacity, among other metrics on a massive scale.

References

[1] D. Andersen, H. Balakrishnan, M. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proceedings of the 18th ACM SOSP*, Banff, Canada, October 2001.

[2] T. Blackwell, J. Harris, and M. Seltzer. Heuristic Cleaning Algorithms for Log-Structured File Systems. In *Proceedings of the 1995 Winter USENIX Technical Conference*, New Orleans, LA, January 1995.

[3] C. Blake and R. Rodrigues. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. In *Proceedings of HotOS IX*, Lihue, HI, May 2003.

[4] J. Byers, J. Considine, and M. Mitzenmacher. Geometric Generalizations of the Power of Two Choices. Technical Report BUCS 2003-002, Boston University, 2003.

[5] J. Byers, J. Considine, and M. Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, February 2003.

[6] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. Research report, Microsoft Research, September 2002.

[7] M. Castro, P. Druschel, Y. Charlie Hu, and A. Rowstron. Exploiting network proximity in distributed hash tables. In *International Workshop on Future Directions in Distributed Computing (FuDiCo)*, June 2002.

[8] M. Castro, P. Drushel, A. Ganesh, A. Rowstron, and D. Wallach. Secure Routing for Structured Peer-to-Peer Overlay Networks. In *OSDI '02*, Boston, MA, 2002.

[9] M. Castro, M. Jones, A.-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An Evaluation of Scalable Application-Level Multicast Using Peer-to-Peer Overlays. In *IEEE INFOCOM 2003*, San Francisco, CA, June 2003.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.

[11] Frank Dabek, Ben Zhao, Peter Druschel, and Ion Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *IPTPS '03*, Berkeley, CA, February 2003.

[12] J. Douceur. The Sybil Attack. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, March 2002.

[13] P. Druschel, R. Gil, Y.C. Hu, S. Iyer, A. Ladd, A. Mislove, A. Nandi, A. Post, C. Reis, A. Singh, and R. Zhang. Rice FreePastry implementation. `http://www.cs.rice.edu/CS/Systems/Pastry/FreePastry`.

[14] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of Email Workloads. In *Proceedings of the 2003 USENIX Conference on File and Storage Technology*, San Francisco, CA, March 2003.

[15] J. Feigenbaum and S. Shenker. Distributed Algorithmic Mechanism Design: Recent Results and Future Directions. In *Sixth International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, September 2002.

[16] FreeBSD Jails. `http://www.freebsd.org`.

[17] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. In *Proceedings of the 1995 Winter USENIX Technical Conference*, New Orleans, LA, January 1995.

[18] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The Impact of DHT Routing Geometry on Resilence and Proximity. In *SIGCOMM 2003*, Karlsruhe, Germany, August 2003.

[19] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed object location in a dynamic network. In *ACM Symposium on Parallel Algorithms and Architectures*, Winnipeg, Canada, August 2002.

[20] R. Mahajan, M. Castro, and A. Rowstron. Controlling the Cost of Reliability in Peer-to-Peer Overlays. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, February 2003.

[21] K. Marx. Critique of the Gotha Program, 1875.

[22] M. Mitzenmacher, A. Richa, and R. Sitaraman. *The Power of Two Choices: A Survey of Techniques and Results*. Kluwer Academic Publishers, Norwell, MA, 2001.

[23] K. Petersen, M. Spreitzer, D.B. Terry, M. Theimer, and A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, St. Malo, France, October 1997.

[24] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *HotNets-I Workshop*, October 2002.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of the ACM SIGCOMM '01 Conference*, August 2001.

[26] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-Aware Overlay Construction and Server Selection. In *IEEE INFOCOM 2002*, New York, NY, June 2002.

[27] Drew Roselli, Jacob Lorch, and Thomas Anderson. A Comparison of File System Workloads. In *USENIX 2000 Technical Conference*, pages 41–54, San Diego, CA, 2000.

[28] M. Roussopoulos and M. Baker. CUP: Controlled Update Propagation in Peer-to-Peer Networks. In *Proceedings of USENIX Annual Technical Conference*, San Antonio, TX, June 2003.

[29] M. Roussopoulos and M. Baker. Practical Load Balancing for Content Requests in Peer-to-Peer Networks. Research Report cs.NI/0209023, Stanford University, January 2003.

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

[31] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the Multimedia Computing and Networking (MMCN)*, San Jose, CA, January 2002.

[32] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, August 2001.

[33] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Research report, MIT, January 2002.

[34] H. Weatherspoon and J. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, March 2002.

[35] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, , and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, December 2002.

[36] E. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *IEEE INFOCOM*, March 1996.

[37] H. Zhang, A. Goel, and R. Govindan. Incrementally Improving Lookup Latency in Distributed Hash Table Systems. In *IEEE INFOCOM 2003*, San Francisco, CA, June 2003.

[38] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Research Report UCB/CSD-01-1141, U.C. Berkeley, April 2001.

APPENDIX

A Funerals are only held on Sundays

We used the Gnutella traces to motivate the idea that, for a given p2p system, one could generate a function that would predict the future lifetime of a node given its current uptime. We found an interesting artifact that was a result of the tracing methodology. It is an important result because it contradicts the original finding that the average Gnutella node has a lifetime of 60 minutes [31]: we find that the average interval of all sessions is at least 80 minutes and that the average interval of a given node is on the order of at least several hours (a more precise value cannot be given with a trace of this length).

The tracing methodology began with a set of 17142 nodes that ran the Gnutella protocol. During a 60 hour session, these nodes were tested once every seven minutes to see if they (a) responded to an IP ping and (b) were listing on the conventional Gnutella port. If a node had not been responding and then did, a new interval began, and if an interval was on-going and the node did not respond, the interval would end. The timeout was 20 seconds. The output of the trace is an anonymized IP address, the number of intervals, followed by the start time and end time of each interval. The IP and Gnutella responses were recorded separately.

We wanted to find the expectations about per node behavior and not per session behavior because we were developing a function describing what a node would see: other, currently up nodes. As a first step, we averaged each node's sessions to find the average session length per node. The result is plotted in Figure 11 as "0 missed pings." The steps occur at 30, 20, 15, 12, . . . hours, at intervals that divide 60. This is because *all* nodes that were (most likely) up for the entire 60 hour session but only missed one ping averaged to 30 hours, no matter when their missed ping was. Two missed pings yields 20 hours, and so on. This begs the question: is it more likely that a node was up for $60 - x$ hours, off for seven minutes, and then again up for x hours, or did it just miss one ping? Of course, we will never know, but the most likely case is the latter. We found that 72% of the trace exhibited at least one instance where the beginning of one trace followed the end of the previous by about seven minutes.

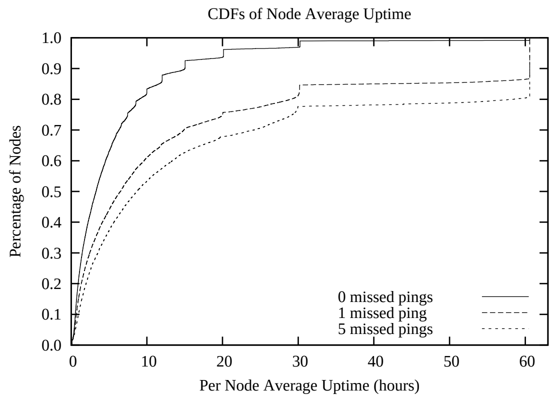


Figure 11: CDF of per node average "Internet" uptime. 0 missed pings is the original interpretation of the data. The plot shows how the average node lifetime dramatically increases as a few missed pings out of the 60 hour trace are permitted.

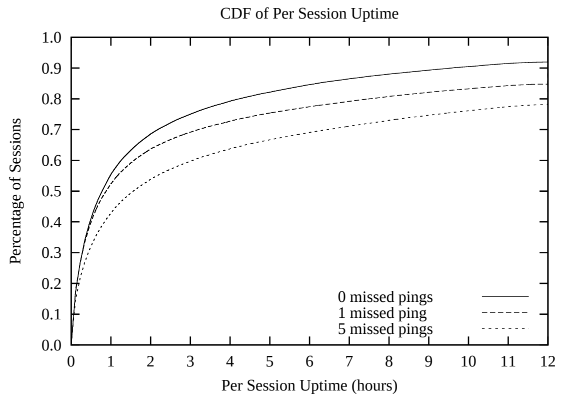


Figure 12: CDF of per session uptime when nodes are allowed to missed 0, 1, or 5 pings in a row. To eliminate deaths only on seven minute intervals, we assigned a node a death time randomly between its last live response and first non-response.

The next question, then, is: how many pings can you miss before you are definitively considered down? There is no right answer: even though there is still a step when 5 gaps $\times 7$ min $= 35$ min gaps are allowed, more than half an hour seems too much, but others are debatable. Because allowing for just seven minute gaps had a large effect and needs the least defense, that is what we chose for generating the expected uptime function. After performing these merges, we used the create-based method [27] to sweep through the trace to find actual lifetimes that were less than our 12 hour window and the residue, the number that lived beyond this window. The results are shown in Figure 12.

Because we wanted to compare our results directly with Mahajan *et al.* [20] in Section 2.1, we did not use these revised intervals in our simulation.