Proceedings of the Second Workshop on Real, Large Distributed Systems (WORLDS '05)

Fixing the Embarrassing Slowness of OpenDHT on PlanetLab

Sean Rhea, Byung-Gon Chun, John Kubiatowicz, and Scott Shenker
University of California, Berkeley
opendht@opendht.org

1 Introduction

The distributed hash table, or DHT, is a distributed system that provides a traditional hash table's simple put/get interface using a peer-to-peer overlay network. To echo the prevailing hype, DHTs deliver incremental scalability in the number of nodes, extremely high availability of data, low latency, and high throughput.

Over the past 16 months, we have run a public DHT service called OpenDHT [14] on PlanetLab [2], allowing any networked host to perform puts and gets over an RPC interface. We built OpenDHT on Bamboo [13] and shamelessly adopted other techniques from the literature—including recursive routing, proximity neighbor selection, and server selection—in attempt to deliver good performance. Still, our most persistent complaint from actual and potential users remained, "It's just not fast enough!"

Specifically, while the long-term median latency of gets in OpenDHT was just under 200 ms—matching the best performance reported for DHASH [5] on PlanetLab—the 99th percentile was measured in seconds, and even the median rose above half a second for short periods.

Unsurprisingly, the long tail of this distribution was caused by a few arbitrarily slow nodes. We have observed disk reads that take tens of seconds, computations that take hundreds of times longer to perform at some times than others, and internode ping times well over a second. We were thus tempted to blame our performance woes on PlanetLab (a popular pastime in distributed systems these days), but this excuse was problematic for two reasons.

First, peer-to-peer systems are supposed to capitalize on existing resources not necessarily dedicated to the system, and do so without extensive management by trained operators. In contrast to managed, cluster-based services supported by extensive advertising revenue, peer-to-peer systems were supposed to bring power to the people, even those with flaky machines.

Second, it is not clear that the problem of slow nodes is limited to PlanetLab. For example, the best DHASH performance on the RON testbed, which is smaller and less loaded than PlanetLab, still shows a 99th percentile get latency of over a second [5]. Furthermore, it is well known that even in a managed cluster the distribution of individual machines' performance is long-tailed. The performance of Google's MapReduce system, for example, was improved by 31% when it was modified to account for a few slow machines its designers called "stragglers" [6]. While PlanetLab's performance is clearly worsened by the fact that it is heavily shared, the current trend towards utility computing indicates that such sharing may be common in future service infrastructures.

It also seems unlikely that one could "cherry pick" a set of well-performing hosts for OpenDHT. The MapReduce designers, for example, found that a machine could suddenly become a straggler for a number of reasons, including cluster scheduling conflicts, a partially failed hard disk, or a botched automatic software upgrade. Also, as we show in Section 2, the set of slow nodes isn't constant on PlanetLab or RON. For example, while the 90% of the time it takes under 10 ms to read a random 1 kB disk block on PlanetLab, over a period only 50 hours, 235 of 259 hosts will take over 500 ms to do so at least once. While one can find a set of fast nodes for a short experiment, it is nearly impossible to find such a set on which to host a long-running service.

We thus adopt the position that the best solution to the problem of slow nodes is to modify our algorithms to account for them automatically. Using a combination of delay-aware routing and a moderate amount of redundancy, our best technique reduces the median latency of get operations to 51 ms and the 99th percentile to 387 ms, a tremendous improvement over our original algorithm.

In the next section we quantify the problem of slow nodes on both PlanetLab and RON. Then, in Sections 3 and 4, we describe several algorithms for mitigating the effects of slow nodes on end-to-end get latency and show their effectiveness in an OpenDHT deployment of approximately 300 PlanetLab nodes. We conclude in Section 5.

2 The Problem of Slow Nodes

In this section, we study the problem of slow nodes in PlanetLab as compared to a cluster of machines in our lab. Our PlanetLab experiments ran on all the nodes we were able to log into at any given time, using a slice dedicated to the experiment.

Our cluster consists of 38 IBM xSeries 330 1U rackmount PCs, each with two 1.0 GHz Pentium III CPUs, 1.5 GB ECC PC133 SDRAM, and two 36 GB IBM UltraStar 36LZX hard drives. The machines use a single Intel PRO/1000 XF gigabit Ethernet adaptor to connect to a Packet Engines PowerRail gigabit switch. The operating system on each node is Debian GNU/Linux 3.0 (woody),

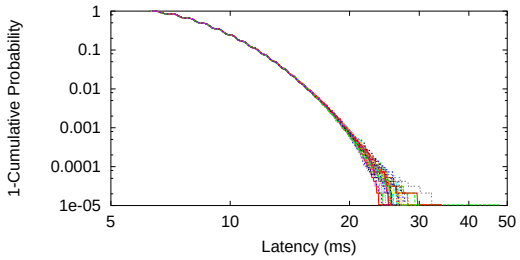

Figure 1: Time to compute a 128-bit RSA key pair on our cluster.

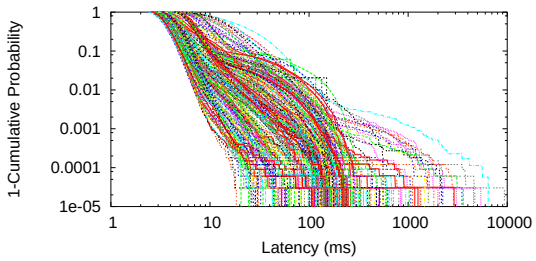

Figure 2: Time to compute a 128-bit RSA key pair on PlanetLab.

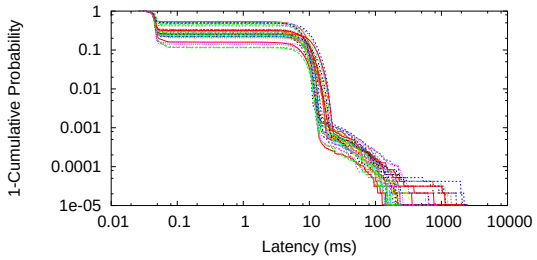

Figure 3: Time to read a random 1 kB disk block on our cluster.

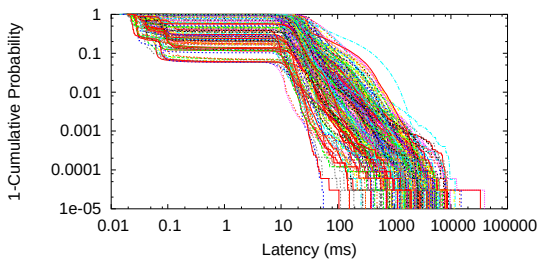

Figure 4: Time to read a random 1 kB disk block on PlanetLab.

running the Linux 2.4.18 SMP kernel. The two disks run in software RAID 0 (striping) using md raidtools-0.90. During our experiments the cluster is otherwise unused.

We study slowness with respect to computation time, network round-trip latency, and disk read latency. For each test, we wrote a simple C program to measure the latency of the resource under test. Our computation benchmark measures the latency to compute a new 128-bit RSA key pair, our network benchmark measures the round-trip time (RTT) of sending a 1 kB request and receiving a 1 kB response, and our disk benchmark measures the latency of reading a random 1 kB block out of a 1 GB file.

Figures 1–4 present the results of the computation and disk read tests; each line in these figures represents over 100,000 data points taken on a single machine. Figures 5 and 6 show the results of the network test; here each line represents over 10,000 data points taken between a single pair of machines.

Looking first at the cluster results, we note that operations we expect to be quick are occasionally quite slow. For example, the maximum ping time is 4.8 ms and the maximum disk read time is 2.5 seconds, a factor of 26 or 54,300 larger than the median time in each case.

Furthermore, there is significant variance between machines or pairs of machines (in the case of network RTTs). For example, the fraction of disk reads served in under 1 ms (presumably out of the cache) varies between 47% and 89% across machines. Also, one pair of machines never sees an RTT longer than 0.28 ms, while another pair sees a maximum RTT of 4.8 ms.

Based on these data, we expect that even an isolated cluster will benefit from algorithms that take into account performance variations between machines and the time-varying performance of individual machines. The MapReduce experience seems to confirm this expectation.

Turning to the PlanetLab numbers, the main difference is that the scheduling latencies inherent in a shared testbed increase the unpredictability of individual machines' performance by several orders of magnitude. This trend is most evident in the computation latencies. On the cluster, most machines showed the same, reasonably tight distribution of computation times; on PlanetLab, in contrast, a computation that never takes more than 18 ms on one machine takes as long as 9.3 *seconds* on another.

Unfortunately, very few nodes in PlanetLab are always fast, as shown in Figure 7. To produce this figure, we ran the disk read test on 259 PlanetLab nodes for 50 hours, pausing five seconds between reads. The figure shows the number of nodes that took over 100 ms, over 500 ms, over 1 s, or over 10 s to read a block since the start of measurement. In only 6 hours, 184 nodes take over 500 ms at least once; in 50 hours, 235 do so.

Furthermore, this property does not seem to be unique to PlanetLab. Figure 8 shows a similar graph produced from a trace of round-trip times between 15 nodes on RON [1], another shared testbed. We compute for each node the median RTT to each of the other fourteen, and rank nodes by these values. The lower lines show the values for the eighth largest and second largest values over time, and the upper line shows the size of the set of nodes that have ever had the largest or second largest value. In only 90 hours, 10 of 15 nodes have been in this set. This graph shows that while the aggregate performance of the 15 nodes is relatively stable, the ordering (in terms of per-

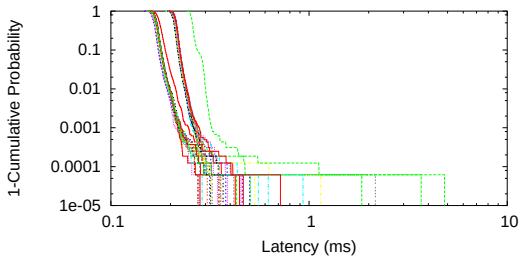

Figure 5: Time to send and receive a 1 kB message on our cluster.

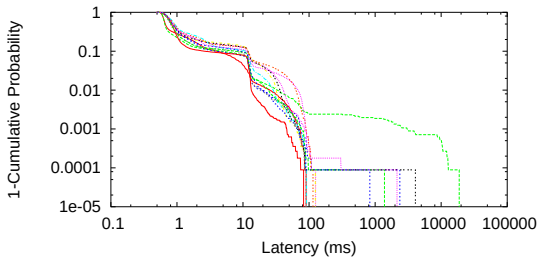

Figure 6: Time to send and receive a 1 kB message on PlanetLab.

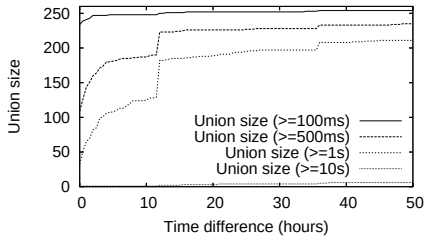

Figure 7: Slow disk reads on PlanetLab over time.

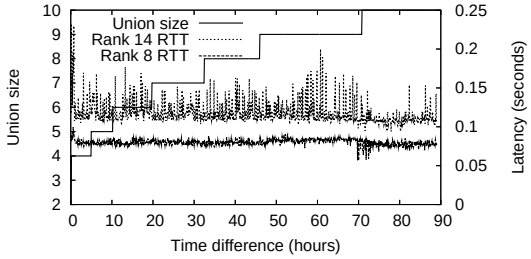

Figure 8: Slow round-trip times on RON over time.

formance) among them changes greatly.

In summary, with respect to network RTT and disk reads, both the relative performance of individual machines and the performance of a single machine over time can show significant variation even on an isolated cluster. On a shared testbed like PlanetLab or RON, this variation is even more pronounced, and the performance of computational tasks shows significant variation as well.

3 Algorithmic Solutions

Before presenting the techniques we have used to improve get latency in OpenDHT, we give a brief overview of how gets were performed before.

3.1 The Basic Algorithm

The key space in Bamboo is the integers modulo 2^{160}. Each node in the system is assigned an identifier from this space uniformly at random. For fault-tolerance and availability, each key-value pair (k,v) is stored on the four nodes that immediately precede and follow k; we call these eight nodes the *replica set* for k, denoted $R(k)$. The node numerically closest to k is called its *root*.

Each node in the system knows the eight nodes that immediately precede and follow it in the key space. Also, for each (base 2) prefix of a node's identifier, it has one neighbor that shares that prefix but differs in the next bit. This latter group is chosen for network proximity; of those nodes that differ from it in the first bit, for example, a node chooses the closest from roughly half the network.

Messages between OpenDHT nodes are sent over UDP and individually acknowledged by their recipients. A congestion-control layer provides TCP-friendliness and retries dropped messages, which are detected by a failure to receive an acknowledgment within an expected time. This layer also exports to higher layers an exponentially weighted average round-trip time to each neighbor.

To put a key-value pair (k,v), a client sends a put RPC to an OpenDHT node of its choice; we call this node the *gateway* for this request. The gateway then routes a put message greedily through the network until it reaches the root for k, which forwards it to the rest of $R(k)$. When six members of this set have acknowledged it, the root sends an acknowledgment back to the gateway, and the RPC completes. Waiting for only 6 of 8 acknowledgments prevents a put from being delayed by one or two slow nodes in the replica set. These delays, churn, and Internet routing inconsistencies may all cause some replicas in the set to have values that others do not. To reconcile these differences, the nodes in each replica set periodically synchronize with each other [12].

As shown in Figure 9, to perform a get for key k, the gateway G routes a get request message greedily through the key space until it reaches some node $R \in R(k)$. R replies with any values it has with key k, the set $R(k)$, and the set of nodes $S(k)$ with which it has synchronized on k recently. G pretends it has received responses from R and the nodes in $S(k)$; if these total five or more, it sends a response to the client. Otherwise, it sends the request directly to the remaining nodes in $R(k)$ one at a time until it has at least five responses (direct or assumed due to synchronization). Finally, G compiles a combined response

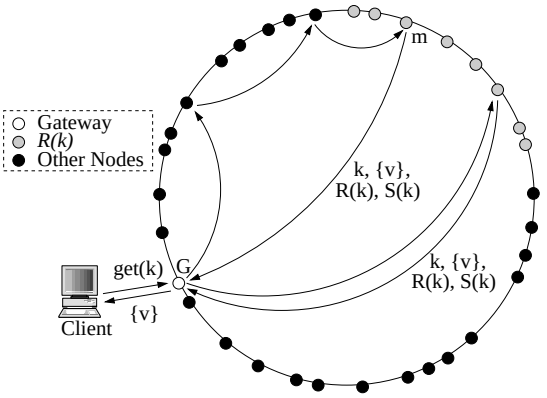

Figure 9: *A basic get request.*

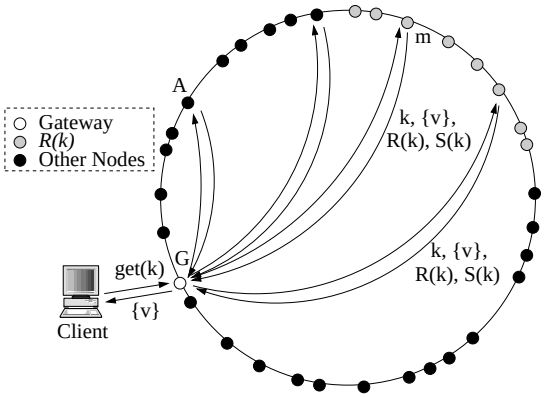

Figure 10: *An iterative get request.*

and returns it to the client.

By combining responses from at least five replicas, we ensure that even after the failure of two nodes, there is at least one node in common between the nodes that receive a put and those whose responses are used for a get.

3.2 Enhancements

We have explored three techniques to improve the latency of gets: delay-aware routing, parallelization of lookups, and the use of multiple gateways for each get.

3.2.1 Delay-Aware Routing

In the basic algorithm, we route greedily through the key space. Because each node selects its neighbors according to their response times to application-level pings, most hops are to nearby, responsive nodes. Nonetheless, a burst in load may render a once-responsive neighbor suddenly slow. Bamboo's neighbor maintenance algorithms are designed for stability of the network, and so adapt to such changes gradually. The round-trip times exported by the congestion-control layer are updated after each message acknowledgment, however, and we can use them to select among neighbors more adaptively.

The literature contains several variations on using such delay-aware routing to improve get latency. Gummadi et al. demonstrated that routing along the lowest-latency hop that makes progress in the key space can reduce end-to-end latency, although their results were based on simulations where the per-hop processing cost was ignored [7]. DHASH, in contrast, uses a hybrid algorithm, choosing each hop to minimize the expected overall latency of a get, using the expected latency to a neighbor and the expected number of hops remaining in the query to scale the progress each neighbor makes in the key space [5].

We have explored several variations on this theme. For each neighbor n, we compute ℓ_n, the expected round-trip time to the neighbor, and d_n, the progress made in the key space by hopping to n, and we modified OpenDHT to choose the neighbor n with maximum $h(\ell_n, d_n)$ at each hop, where h is as follows:

Purely greedy:	$h(\ell_n, d_n) = d_n$
Purely delay-based:	$h(\ell_n, d_n) = 1/\ell_n$
Linearly scaled:	$h(\ell_n, d_n) = d_n/\ell_n$
Nonlinearly scaled:	$h(\ell_n, d_n) = d_n/f(\ell_n)$

where $f(\ell_n) = 1 + e^{(\ell_n - 100)/17.232}$. This function makes a smooth transition for ℓ_n around 100 ms, the approximate median round-trip time in the network. For round-trip times below 100 ms, the nonlinear mode thus routes greedily through the key space, and above this value it routes to minimize the per-hop delay.

3.2.2 Iterative Routing

Our basic algorithm performs get requests *recursively*; routing each request through the network to the appropriate replica set. In contrast, gets can also be performed *iteratively*, where the gateway contacts each node along the route path directly, as shown in Figure 10. While iterative requests involve more one-way network messages than recursive ones, they remain attractive because they are easy to parallelize. As first proposed in Kademlia [9], a gateway can maintain several outstanding RPCs concurrently, reducing the harm done by a single slow peer.

To perform a get on key k iteratively, the gateway node maintains up to p outstanding requests at any time, and all requests are timed out after five seconds. Each request contains k and the Vivaldi [4] network coordinates of the gateway. When a node $m \notin R(k)$ receives a get request, it uses Vivaldi to compute ℓ_n relative to the gateway for each of its neighbors n, and returns the three with the largest values of $h(d_n, \ell_n)$ to the gateway.

When a node $m \in R(k)$ receives a get request, it returns the same response as in recursive gets: the set of values stored under k and the sets $R(k)$ and $S(k)$. Once a gateway has received a response of this form, it proceeds as in recursive routing, collecting at least five responses before compiling a combined result to send to the client.

3.2.3 Multiple Gateways

Unlike iterative gets, recursive gets are not easy to parallelize. Also, in both iterative and recursive gets, the gateway itself is sometimes the slowest node involved in a request. For these reasons we have also experimented with issuing each get request simultaneously to multiple gateways. This technique adds parallelism to both types of get, although the paths of the get requests may overlap as they near the replica set. It also hides the effects of slow gateways.

4 Experimental Results

It is well known that as a shared testbed, PlanetLab cannot be used to gather exactly reproducible results. In fact, the performance of OpenDHT varies on a hourly basis.

Despite this limitation, we were able to perform a meaningful quantitative comparison between our various techniques as follows. We modified the OpenDHT implementation such that each of the modes can be selected on a per-get basis, and we ran a private deployment of OpenDHT on a separate PlanetLab slice from the public one. Using a client machine that was not a PlanetLab node (and hence does not suffer from the CPU and network delays shown in Figures 2 and 6), we put into OpenDHT five 20-byte values under each of 3,000 random keys, reputting them periodically so they would not expire. On this same client machine, we ran a script that picks a one of the 3,000 keys at random and performs one get for each possible mode in a random order. The script starts each get right after the previous one completes, or after a timeout of 120 seconds. After trying each mode, the script picks a new key, a new random ordering of the modes, and repeats. So that we could also measure the cost of each technique, we further modified the OpenDHT code to record the how many messages and bytes it sends on behalf of each type of get. We ran this script from July 29, 2005 until August 3, 2005, collecting 27,046 samples per mode to ensure that our results cover a significant range of conditions on PlanetLab.

Table 1 summarizes the results of our experiments.

The first row of the table shows that our original algorithm, which always routes all the way to the root, takes 186 ms on median and over 8 s at the 99th percentile.

Rows 2–5 show the performance of the basic recursive algorithm of Section 3.1, using only one gateway and each of the four routing modes described in Section 3.2.1. We note that while routing with respect to delay alone improves get latency some at the lower percentiles, the linear and nonlinear scaling modes greatly improve latency at the higher percentiles as well. The message counts show that routing only by delay takes the most hops, and with each hop comes the possibility of landing on a newly slow node; the scaled modes, in contrast, pay enough attention to delays to avoid the slowest nodes, but still make quick progress in the key space.

	Parameters				Latency (ms)				Cost per Get	
Row	GW	I/R	*p*	Mode	Avg	50th	90th	99th	Msgs	Bytes
1	1	Orig. Alg.			434	186	490	8113	not measured	
2	1	R	1	Greedy	282	149	407	4409	5.5	1833
3	1	R	1	Prox.	298	101	343	5192	8.7	2625
4	1	R	1	Linear	201	99	275	3219	6.8	2210
5	1	R	1	Nonlin.	185	104	263	1830	6.0	1987
6	1	I	3	Greedy	157	116	315	788	14.6	3834
7	1	I	3	Prox.	477	335	1016	2377	33.1	6971
8	1	I	3	Linear	210	175	422	802	18.8	4560
9	1	I	3	Nonlin.	230	175	455	1103	18.3	4458
10	1	R	1	Nonlin.	185	104	263	1830	6.0	1987
11	2	R	1	Nonlin.	174	99	267	1609	6.0	1987
12	1–2	R	1	Nonlin.	107	71	171	609	11.9	3973
13	1	I	3	Greedy	157	116	315	788	14.6	3834
14	2	I	3	Greedy	147	110	294	731	14.6	3834
15	1–2	I	3	Greedy	88	70	195	321	29.3	7668
16	1–2	I	1	Greedy	141	96	289	638	13.9	4194
17	1–2	I	2	Greedy	97	78	217	375	22.5	6181
18	1–3	R	1	Nonlin.	90	57	157	440	16.8	5332
19	1–4	R	1	Nonlin.	81	51	142	387	22.4	7110
20	1–2	I	2	Greedy	105	84	232	409	20.2	5352
21	1–2	I	3	Greedy	95	76	206	358	26.5	6674
22	1–3	I	2	Greedy	86	62	196	332	30.3	8028

Table 1: *Performance on PlanetLab. GW* is the gateway, 1–4 for planetlab(14|15|16|13).millennium.berkeley.edu. *I/R* is for iterative or recursive. The costs of the single gateway modes are estimated as half the costs of using both.

We note that the median latencies achieved by all modes other than greedy routing are lower than the median network RTT between OpenDHT nodes, which is approximately 137 ms. This seemingly surprising result is actually expected; with eight replicas per value, the DHT has the opportunity to find the closest of eight nodes on each get. Using the distribution of RTTs between nodes in OpenDHT, we computed that an optimal DHT that magically chose the closest replica and retrieved it in a single RTT would have a median get latency of 31 ms, a 90th percentile of 76 ms, and a 99th percentile of 130 ms.

Rows 6–9 show the same four modes, but using iterative routing with a parallelism factor, *p*, of 3. Note that the non-greedy modes are not as effective here as for recursive routing. We believe there are three reasons for this effect. First, the per-hop cost in iterative routing is higher than in recursive, as each hop involves a full round-trip, and on average the non-greedy modes take more hops for each get. Second, recursive routing uses fresh, direct measurements of each neighbor's latency, but the Vivaldi algorithm used in iterative routing cannot adapt as quickly to short bursts in latency due to load. Third, our Vivaldi implementation does not yet include the kinds of filtering used by Pietzuch, Ledlie, and Seltzer to produce more accurate coordinates on PlanetLab [10]; it is possible that their implementation would produce better coordinates with which to guide iterative routing decisions.

Despite their inability to capitalize on delay-awareness, the extra parallelism of iterative gets provides enough resilience to far outperform recursive ones at the 99th percentile. This speedup comes at the price of a factor of two in bandwidth used, however.

Rows 10–12 show the benefits of using two gateways with recursive gets. We note that while both gateways are equally slow individually, waiting for only the quickest of them to return for any particular get greatly reduces latency. In fact, for the same cost in bandwidth, they far outperform iterative gets at all percentiles.

Rows 13–15 show that using two gateways also improves the performance of iterative gets, reducing the 99th percentile to an amazing 321 ms, but this performance comes at a cost of roughly four times that of recursive gets with a single gateway.

Rows 16–17 show that we can reduce this cost by reducing the parallelism factor, p, while still using two gateways. Using $p = 1$ gives longer latencies than recursive gets with the same cost, but using $p = 2$ provides close to the performance of $p = 3$ at only three times the cost of recursive gets with a single gateway.

Since iterative gets with two gateways and $p = 3$ use more bandwidth than any of the recursive modes, we ran a second experiment using up to four gateways per get request. Rows 18–22 show the results. For the same cost, recursive gets are faster than iterative ones at both the median and 90th percentile, but slower at the 99th.

These differences make sense as follows. As the gateways are co-located, we expect the paths of recursive gets to converge to the same replica much of the time. In the common case, that replica is both fast and synchronized with its peers, and recursive gets are faster, as they have more accurate information than iterative gets about which neighbor is fastest at each hop. In contrast, iterative gets with $p > 1$ actively explore several replicas in parallel and are thus faster when one discovered replica is slow or when the first replica is not synchronized with its peers, necessitating that the gateway contact multiple replicas.

5 Conclusions

In this work we highlighted the problem of slow nodes in OpenDHT, and we demonstrated that their effect on overall system performance can be mitigated through a combination of delay-aware algorithms and a moderate amount of redundancy. Using only delay-awareness, we reduced the 99th percentile get latency from over 8 s to under 2 s. Using a factor of four more bandwidth, we can further reduce the 99th percentile to under 400 ms and cut the median by a factor of three.

Since performing this study, we have modified the public OpenDHT deployment to perform all gets using delay-aware routing with nonlinear scaling, and we have encouraged users of the system to use multiple gateways for latency-sensitive gets. The response from the OpenDHT user base has been very positive.

Looking beyond our specific results, we note that there has been a lot of collective hand-wringing recently about the value of PlanetLab as an experimental platform. The load is so high, it is said, that one can neither get high performance from an experimental service nor learn interesting systems lessons applicable elsewhere.

We have certainly cast some doubt on the first of these two claims. The latencies shown in Table 1 are low enough to enable many applications that were once thought to be outside the capabilities of a "vanilla" DHT. For example, Cox et al. [3] worried that Chord could not be used to replace DNS, and others argued that aggressive caching was required for DHTs to do so [11]. On the contrary, even our least expensive modes are as fast as DNS, which has a median latency of around 100 ms and a 90th percentile latency of around 500 ms [8].

As to the second claim, there is no doubt that PlanetLab is a trying environment on which to test distributed systems. That said, we suspect that the MapReduce designers might say the same about their managed cluster. Their work with stragglers certainly bears some resemblance to the problems we have dealt with. While the question is by no means settled, we suspect that PlanetLab may differ from their environment mainly by degree, forcing us to solve problems at a scale of 300 nodes that we would eventually have to solve at a scale of tens of thousands of nodes. If this suspicion is correct, perhaps PlanetLab's slowness is not a bug, but a feature.

References

[1] `http://nms.csail.mit.edu/ron/data/`.

[2] A. Bavier et al. Operating system support for planetary-scale network services. In *NSDI*, Mar. 2004.

[3] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS using a peer-to-peer lookup service. In *IPTPS*, 2002.

[4] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *SIGCOMM*, 2004.

[5] F. Dabek et al. Designing a DHT for low latency and high throughput. In *NSDI*, 2004.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

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

[8] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. In *SIGCOMM IMW*, 2001.

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

[10] P. Pietzuch, J. Ledlie, and M. Seltzer. Supporting network coordinates on PlanetLab. In *WORLDS*, 2005.

[11] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the Internet. In *SIGCOMM*, 2004.

[12] S. Rhea. *OpenDHT: A public DHT service.* PhD thesis, U.C. Berkeley, Aug. 2005.

[13] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *USENIX Annual Tech. Conf.*, 2004.

[14] 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 *SIGCOMM*, 2005.