ShortCuts: Using Soft State To Improve DHT Routing

Kiran Tati and Geoffrey M. Voelker
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{ktati, voelker}@cs.ucsd.edu

Abstract

Distributed hash tables are increasingly being proposed as the core substrate for content delivery applications in the Internet, such as cooperative Web caches, Web index and search, and content delivery systems. The performance of these applications built on DHTs fundamentally depends on the effectiveness of request routing within the DHT. In this paper, we show how to use soft state to achieve routing performance that approaches the aggressive performance of one-hop schemes, but with an order of magnitude less overhead on average. We use three kinds of hint caches to improve routing latency: local hint caches, path hint caches, and global hint caches. Local hint caches use large successor lists to short cut final hops. Path hint caches store a moderate number of effective route entries gathered while performing lookups for other nodes. And global hint caches store direct routes to peers distributed across the ID space. Based upon our simulation results, we find that the combination of the hint caches significantly improves Chord routing performance: in a network of 4,096 peers, the hint caches enable Chord to route requests with average latencies only 6% more than algorithms that use complete routing tables with significantly less overhead.

1 Introduction

Peer-to-peer overlay networks provide a distributed, fault-tolerant, scalable architecture on which wide-area distributed systems and applications can be built. An increasing trend has been to propose content delivery services on peer-to-peer networks, including cooperative Web caches [9], Web indexing and searching [14,16], content delivery systems [3,12], and Usenet news [6]. Popular designs of these overlay networks implement a distributed hash table (DHT) interface to higher level software. DHTs map keys in a large, virtual ID space to associated values stored and managed by individual nodes in the overlay network. DHTs use a distributed routing protocol to implement this mapping. Each node in the overlay network maintains a routing table. When a node receives a request for a particular key, it forwards the request to another node in its routing table that brings the request closer to its destination.

A natural trade off in the design of these routing protocols is the size of the routing table and the latency of routing requests. Larger routing tables can reduce routing latency in terms of the number of hops to reach a destination, but at the cost of additional route maintenance overhead. Because the performance and overhead of DHT overlay networks fundamentally depend upon the distributed routing protocol, significant work has focused on the problem of balancing the degree of routing state and maintenance with route performance.

Initial systems like Chord [28], Pastry [26], Tapestry [31], and CAN [24] use routing tables of degree $O(\log n)$ to route requests in $O(\log n))$ hops, where n is the number of hosts in the network. Newer algorithms improve the theoretical bounds on routing state and hops. Randomized algorithms like Viceroy [17] and Symphony [18] achieve small, constant-degree routing tables to route requests on average in $O(\log n)$ and $O(\log \log n)$ hops, respectively. Koorde [10] is a tunable protocol that can route requests with a latency ranging from $O(\log n)$ to $O(\log n / \log \log n)$ hops for routing tables of constant size to $O(\log n))$ size, respectively. Other approaches, such as Kelips [8], Structured Superpeers [22], Beehive [23], and CUP [25] focus on achieving constant-time $O(1)$ hops to route requests at the expense of high degree routing tables, hierarchical routing, tailoring to traffic distributions, or aggressive update protocols to maintain consistency among the large routing tables in each peer.

In this paper, we argue that the appropriate use of cached routing state within the routing protocol can provide competitive improvements in performance while using a simple baseline routing algorithm. We describe and evaluate the use of three kinds of *hint caches* containing route hints to improve the routing performance of distributed hash tables (DHTs): *local hint caches* store direct routes to successors in the ID space; *path hint caches* store direct routes to peers accumulated during the natural processing of lookup requests; and *global hint caches* store direct routes to a set of peers roughly uniformly distributed across the ID space.

These hint caches require state similar to previous approaches that route requests in constant-time hops, but they do not require the complexity and communication overhead of a distributed update mechanism to maintain consistency among cached routes. Instead, the hint caches do not explicitly maintain consistency in response to peer arrivals and departures other than as straightforward extensions of the standard operations of the overlay network. We show that hint cache inconsistency does not degrade their performance benefits.

We evaluate the use of these hint caches by simulating the latest version of the Chord DHT [5] and extending it to use the three hint caches. We evaluate the effectiveness of the hint caches under a variety of conditions, including highly volatile peer turnover rates and relatively large network sizes. Based upon our simulation results, we find that the combination of the hint caches significantly improves Chord routing performance. In networks of 4,096 peers, the hint caches enable Chord to route requests with average latencies only 6% more than algorithms like "One-Hop" that use complete routing tables, while requiring an order of magnitude less bandwidth to maintain the caches and without the complexity of a distributed update mechanism to maintain consistency.

The remainder of the paper is organized as follows. In Section 2, we discuss related work on improving routing performance in peer-to-peer overlays. Section 3 describes how we extend Chord to use the local, path, and global hint caches. Section 5 describes our simulation methodology for evaluating the hint caches, and presents the results of our evaluations. Finally, Section 6 summarizes our results and concludes.

2 Related Work

Initial distributed routing protocols for DHT peer-to-peer overlay networks were designed to balance routing state overhead and route maintenance overhead while providing provably attractive routing performance. Chord [28], Pastry [26], Tapestry [31], and Kademlia [19] use routing tables with degree $O(\log n)$ to route requests in $O(\log n)$ hops through the networks.

Since the performance and overhead of these networks fundamentally depend upon the distributed routing protocol, significant research has focused on the problems of improving route maintenance and performance. As a result, newer algorithms have improved the theoretical bounds on routing degree and routing hops. The most efficient of these algorithms achieve either constant degree routing state or constant hop routing latency, often, however, at the price of additional system complexity.

A number of algorithms seek to minimize route maintenance overhead by using only constant-size

$O(1)$ routing tables per node, such as Viceroy [17], Symphony [18], and [10]. Several efforts have also been made to achieve constant-time $O(1)$ hops to route requests at the cost of high-degree routing tables. These approaches use gossip protocols to propagate route updates [8], hierarchical routing through structured superpeers [22], complete routing tables consistently maintained using a hierarchical update protocol [7], and reliance on traffic distributions [23].

Chord/DHash++ [5] exploits the fact that lookups for replicated values only need to reach a peer near the owner of the key associated with the value (since the peer will have a replica). Although this is appropriate for locating any one of a number of replicas, many applications require exact lookup. The "OneHop" approach uses a very aggressive update mechanism to route requests in only a single hop [7]. However, this approach requires a hierarchical update distribution tree overlayed on the DHT, and requires significant communication overhead to distribute updates to all nodes in the system. Beehive exploits the power-law property of lookup traffic distributions [23] to achieve constant-time lookups. However, a large class of applications induce different types of lookup traffic.

Perhaps the most closely related work is the Controlled Update Protocol (CUP) for managing *path caches* on peers [25]. CUP uses a query and update protocol to keep path caches consistent with peer arrivals and departures. CUP was implemented in the context of the CAN overlay network, and evaluated relative to straightforward path caches with expiration. Although the CUP path caches are analogous to the path hint caches in our work, our work differs in a number of ways with the CUP approach. Rather than providing an update mechanism to keep caches consistent, we instead combine the use of local hint caches with path and global hint caches to improve performance and tolerate inconsistency. We also evaluate hint caches with a baseline DHT routing algorithm that routes in $O(\log n)$ hops (rather than a range of coordinate dimensions).

3 Design

Distributed hash tables (DHT) increasingly serve as the foundation for a wide range of content delivery systems and applications. The DHT lookup operation is the fundamental operation on which applications base their communication. As a result, the performance of these applications directly depends on the performance of the lookup operation, and improving lookup performance improves performance for all applications layered on DHTs.

The primary goal of our work is to reduce lookup performance as close to direct routing with much less overhead than previous approaches and without relying upon specific traffic patterns. We also integrate the cache update mechanism to refresh cached route entries into the routing protocol to minimize the update complexity as well as overhead. To achieve this goal, each peer employs three *hint caches*. *Local hint caches* store direct routes to neighbors in the ID space. *Path hint caches* store direct routes to peers accumulated during the natural processing of lookup requests. Finally, *global hint caches* store direct routes to a set of peers roughly uniformly distributed across the ID space. We call them hint caches since the cached routes are hints that may potentially be stale or inconsistent. We also consider them soft-state hints since they can be reconstructed quickly at any time and they are not necessary for the correctness of the routing algorithm.

The following sections describe the behavior of each of the three hint caches. Although these caches are applicable to DHTs in general, we describe them in the context of integrating them into the Chord DHT as a concrete example. So we start with a brief overview of the Chord lookup operation and routing algorithm as background.

3.1 The Chord DHT

In Chord, all peers in the overlay form a circular linked list. Each peer has one successor and one predecessor. Each peer also maintains $O(\log n)$ successors and $O(\log n)$ additional peers called *fingers*. The owner of a key is defined as a peer for which the key is in between the peer's predecessor's ID and its ID. The lookup operation for a given key returns the owner peer by successively traversing the overlay. Peers construct their finger tables such that the lookup operation traverses progressively closer to the owner in each step. In recursive lookup, the initiator peer uses its routing table to contact the closest peer to the key. This closest peer then recursively forwards the lookup request using its routing table. Included in the request is the IP address of the initiating peer. When the request reaches the peer that owns the key, that peer responds directly to the initiator. This lookup operation contacts $O(\log n)$ application level intermediate peers to reach the owner for a given key.

We augment Chord with the three hint caches. Chord uses these hint caches as simple extensions to its original routing table. When determining the next best hop to forward a request, Chord considers the entries in its original finger table as well as all entries in the hint caches.

3.2 Local Hint Caches

Local hints are direct routes to neighbors in the ID space. They are extensions of successor lists in Chord and leaf nodes in Pastry, except that their purpose is to improve routing performance. With a cache of local hints, a peer can directly reach a small fraction of peers directly and peers can short cut the final hops of request routing.

Local hints are straightforward to implement in a system like Chord using its successor lists. Normally, each peer maintains a small list of its successors to support fault-tolerance within Chord and upper layer applications. Peers request successor lists when they join the network. As part of a process called *stabilization* in Chord, each peer also periodically pings its successor to determine liveness and to receive updates of new successors in its list. This stabilization process is fundamental for maintaining lookup routing correctness, and most DHT designs perform similar processes to maintain successor liveness.

We propose enlarging these lists significantly — on the order of a thousand entries — to become local hint caches. Growing the successor lists does not introduce any additional updates, but it does consume additional bandwidth. The additional bandwidth required is $\frac{S}{H}$ entries per second where S is the number of entries in local hint cache, and H is the half life time of peers in the system. Each peer change, either joining or leaving, requires two entries to update. Similar to [15], we define the half life as the time in seconds for half of the peers in the system to either leave or join the DHT. For perspective, a study of the Overnet peer-to-peer file sharing system measured a half life of four hours [1].

The overhead of maintaining the local hint cache is quite small. For example, when S is 1000 entries and H is four hours, then each peer will receive 0.07 extra entries per second during stabilization. Since entries are relatively small (e.g., 64 bytes), this corresponds to only a couple of bytes/sec of overhead.

Local hint caches can be inconsistent due to peer arrivals and departures. When a peer fails or a new peer joins, for example, its immediate predecessor will detect the failure or join event during stabilization. It will then update its successor list, and start propagating this update backwards along the ring during subsequent rounds of stabilization. Consequently, the further one peer is from one its successors, the longer it takes that peer to learn that the successor has failed or joined.

The average amount of stale data in the local hint cache is $\frac{R*S*(S+1)}{4*H}$, where R is the stabilization period in seconds (typically one second). On average a peer accumulates $\frac{1}{2*H}$ peers per second of stale data. Since a peer updates its x'th successor every $x*R$ seconds, it accumulates $\frac{x*R}{2*H}$ stale entries from its x'th successor. If a peer has S successors, then on average the total amount of stale data is $\sum_{i=1}^{S} \frac{i*R}{2*H}$. If the system half life time H is four hours and the local hint cache size is 1000 peers, then each peer only has 1.7% stale entries. Of course, a peer can further reduce the stale data by using additional update mechanisms, introducing additional bandwidth and complexity. Given the small impact on routing, we argue that such additions are unnecessary.

3.3 Path Hint Caches

The distributed nature of routing lookup requests requires each peer to process the lookup requests of other peers. These lookup requests are generated both by the application layered on top of the DHT as well as the DHT itself to maintain the overlay structure. In the process of handling a lookup request, a given peer can get information about other peers that contact it as well as the peer that initiated the lookup.

With Path Caching with Expiration (PCX) [25], peers cache path entries when handling lookup requests, expiring them after a time threshold. PCX caches entries to the initiator of the request as well as the result of the lookup, and the initiator caches the results of the lookup. In PCX, a peer stores routes to other peers without considering the latency between itself and these new peers. In many cases, these extra peers are far away in terms latency. Using these cached routes to peers can significantly add to the overall latency of lookups (Figure 6(a)). Hence PCX, although it reduces hops (Figure 6(b)), can also counter-intuitively increase lookup latency.

Instead, peers should be selective in terms of caching information about routes to other peers learned while handling lookups. We propose a selection criteria based on the latency to select a peer to cache it. A peer x caches a peer y if the latency to y from x is less than the latency from x to peer z, where (1) z is in x's finger table already and (2) its ID comes immediately before y's ID if x orders the IDs of its finger table peers. For example, assume y falls between a and b in x's finger table and then peer x contacts a to perform the lookup request for an ID between $(a, b]$. If we insert y, then x would contact y for the ID between $(y, b]$. Since the latency to y from x is less than the latency a from x, the lookup latency may reduce for IDs between $(y, b]$. As a result, x will cache the hop to y. We call the cache that collects such hints the path hint cache.

We would like to maintain the path hint cache without the cost of keeping entries consistent. The following cache eviction mechanism tries to achieve this goal. Since a small amount of stale data will not affect lookup performance significantly (Figure 4), a peer tries to choose a time period to evict entries in the path hint cache such that amount of stale data in its path cache is small, around 1%. The average time to accumulate d percentage of stale data in the path hint cache is $2 * d * h$ seconds, where h is the halving time [15]. Hence a peer can use this time period as the eviction time period.

Although the improvement provided by path hint caches is somewhat marginal (1–2%), we still use this information since it takes advantage of existing communication and comes free of cost.

3.4 Global Hint Caches

The goal of the global hint cache is to approximate two-hop route coverage of the entire overlay network using a set of direct routes to low-latency, or *nearby*, peers. Ideally, entries in the global hint cache provide routes to roughly equally distributed points in the ID space; for Chord, these nearby routes are to peers roughly equally distributed around the ring.

These nearby peers work particularly well in combination with the local hint caches at peers. When routing a request, a peer can forward a lookup to one of its global cache entries whose local hint cache has a direct route to the destination. With a local hint cache with 1000 entries, a global hint cache with a few thousand nodes will approximately cover an entire system of few million peers in two hops.

A peer populates its global hint cache by collecting route entries to low-latency nodes by walking the ID space. A peer x contacts a peer y from its routing table to request a peer z from y's local hint cache. The peer x can repeat this process from z until it reaches one of its local hint cache peers. We call this process *space walking*.

While choosing peer z, we have three requirements: minimizing the latency from x, minimizing x's global hint cache size, and preventing gaps in coverage due to new peer arrivals. Hence, we would like to have a large set of peers to choose from to find the closest peer, to choose the farthest peer in the y's local hint cache to minimize the global hint cache size, and to choose the closer peer in y's local hint cache to prevent gaps. To balance these three requirements, when doing a space walk to fill the global hint cache we use the second half of the successor peers in the local hint cache.

Each peer uses the following algorithm to maintain the global hint cache. Each peer maintains an index pointer into the global hint cache called the *refresh pointer*. Initially, the refresh pointer points to the first entry in the global hint cache. The peer then periodically walks through the cache and examines cache entries for staleness. The peer only refreshes a cache entry if the entry has not been used in the previous half life time period. The rate at which the peer examines entries in the global hint cache is $\frac{g}{2*d*h}$, where d is targeted percentage of stale data in the global hint cache, g is the global hint cache size, and h is the halving time. This formula is based on the formula for stale data in the path hint cache (Section 3.3).

d is a system configuration parameter, and peers can estimate h based upon peer leave events in the local hint cache. For example, if the halving time h is four hours, the global hint cache size g is 1000, and the maximum staleness d is 0.125%, then the refresh time period is 3.6 seconds. Note that if a peer uses an entry in the global hint cache to perform a lookup, it implicitly refreshes it as well and consequently reduces the overhead of maintaining the hint cache.

Scaling the system to a very large number of nodes, such as two million peers, the global hint cache would have around 4000 entries and peers would require one ping message per second to maintain 0.5% stale data in very high churn situations like one-hour halving times. Such overheads are small, even in large networks.

Peers continually maintain the local and path hint caches after they join the DHT. In contrast, a peer will only start space walking to populate its global hint cache if it receives a threshold explicit lookup requests directly from the application layer (as opposed to routing requests from other peers). The global hint cache is only useful for the lookups made by the peer itself. Hence, it is unnecessary to maintain this cache for a peer that is not making any lookup requests. Since a peer can build this cache very quickly (Figure 9), it benefits from this cache soon after it starts making application level lookups. A peer maintains the global hint cache using the above algorithm as long as it receives lookups from applications on the peer.

3.5 Discussion

Our goal is to achieve near-minimal request routing performance with significantly less overhead than previous approaches. Local hint caches require $\frac{S}{H}$ entries/sec additional stabilization bandwidth, where S is the number of entries in the local hint cache and H is the half life of the system. Path hint caches require no extra bandwidth since they incorporate information from requests sent to the peer. And, in the worst case, global hint caches require one ping message per $\frac{2*d*h}{g}$ seconds to refresh stale entries.

For comparison, in the "OneHop" approach [7] each peer periodically communicates $\frac{N}{2*H}$ entries to update its routing table, an order of magnitude more overhead. With one million peers at four hour half life time, for example, peers in "OneHop" would need to communicate at least 35 entries per second to maintain the state consistently, whereas the local hint cache requires 0.07 entries per second and one ping per 28 seconds to maintain the global hint cache.

4 Shun Pikes

The direct route provided by the Internet between two peers may not be the best route [27]. According to the latest study, 17% pairs of peers reduce 25 milliseconds in latency if we use alternate paths [13] with some intermediate hops. Our goal is to reduce the DHT lookup latency by exploiting the alternate paths that are better than the direct paths. Each peer tries to forward the lookup request to a peer that is closest to the lookup key from its routing table until the lookup request reaches the owner. We try to use these alternate paths to reach these intermediate hops if alternative paths reduce latency.

The simplest approach to find alternate shortest paths is to run the single source shortest paths algorithm (either Dijkstra or Bellman-Ford) at each peer using the local hint cache and global hint cache as the intermediate hops. The shortest paths from a single node to all other nodes in a complete graph can be approximated closely by the shortest paths that are constructed from a few random subset of nodes in the graph similar to Internet [29]. In other words, the shortest paths constructed for single node with some random nodes and edges to these random nodes are almost as good as the shortest paths constructed from the same node to all other nodes in the graph with complete graph. In our case, the local hint cache has small set of random subset of peers in the DHT and the global hint cache has the closest peers for a given peer. Hence, the shortest path constructed from a peer x to any other peer y in the DHT using just x's local and global hint cache peers as intermediate hops is within constant factor from shortest path from x to y if x uses all other peers in the DHT as intermediate hops to construct the shortest path.

Each peer constructs the shortest paths to all other peers in its caches and tries to uses this information in forwarding the lookup operation. As previously, a peer x tries to forward the lookup request to the closest peer y in the identifier space. However if there is a better path from x to y through peer z then peer x forwards the lookup to peer z, instead of directly forwarding to y, to forward the lookup request to the y.

Overall this new lookup algorithm combining with the local and global hint cache performs better than the "OneHop" approach with significantly less overhead. However, "OneHop" could be modified to take advantages of detours to improve the lookup operation and can achieve similar lookup performance.

5 Methodology and Results

In this section we describe our DHT simulator and our simulation methodology. We also define our performance metric, average space walk time, to evaluate the benefits of our hint caches on DHT routing performance.

5.1 Chord Simulator

Although the caching techniques are applicable to DHTs in general, we chose to implement and evaluate them in the context of Chord [28] due to its relative simplicity. Although the Chord group at MIT makes its simulator available for external use [11], we chose to implement our own Chord simulator together with our hint caching extensions. We implemented a Chord simulator according to the recent design in [5] that optimizes the lookup latency by choosing nearest fingers. It is an event-driven simulator that models network latencies, but assumes infinite bandwidth and no queuing in the network. Since our experiments had small bandwidth requirements, these assumptions have a negligible effect on the simulation results.

We separated the successor list and finger tables to simplify the implementation of the hint caches. During stabilization, each peer periodically pings its successor and predecessor. If it does not receive an acknowledgment to its ping, then it simply removes that peer from it tables. Each peer also periodically requests a successor list update from its immediate successor, and issues lookup requests to keep its finger table consistent. When the lookup reaches the key's owner, the initiating peer chooses as a finger the peer with the lowest latency among the peers in the owner's successor list.

For our experiments, we used a period of one second to ping the successor and predecessor and a 15 minute time period to refresh the fingers. A finger is refreshed immediately if a peer detects that the finger has left the DHT while performing the lookup operation. These time periods are same as ones used in the Chord implementation [11].

To compare different approaches, we want to evaluate the potential performance of a peer's routing table for a given approach. We do this by defining a new metric called *space walk latency*. The space walk latency for a peer is the average time it takes to perform a lookup to any other peer on the DHT at a given point of time. We define a similar metric, *space walk hops*, in terms of hops rather than latency. The space walk time is a more complete measurement than a few thousands of random sample lookups because space walk time represent lookup times to all peers in the network.

We simulate experiments in three phases: an initial phase, a stabilization phase, and an experiment phase. The initial phase builds the Chord ring of a specified number of nodes, where nodes join the ring at the rate of 50 nodes per second. The stabilization phase settles the Chord ring over 15 minutes and establishes a stable baseline for the Chord routing data structures. The experimental phase simulates the peer request workload and peer arrival and departure patterns for a specified duration. The simulator collects results to evaluate the hint caching techniques only during the experimental phase.

Because membership churn is an important aspect of overlay networks, we study the performance of the hint caches using three different churn scenarios: twenty-four-hour, four-hour, and one-hour half life times. The twenty-four-hour half life time represent the churn in a distributed file system with many stable corporate/university peers [2]. The four-hour half life time represent the churn in a file sharing peer-to-peer network with many home users [20]. And the one-hour half life time represent extremely aggressive worst-case churn scenarios [7].

For the simulations in this paper, we use an overlay network of 8,192 peers with latency characteristics derived from real measurements. We start with the latencies measured as part of the Vivaldi [4] evaluation using the King [13] measurement technique. This data set has approximately 1,700 DNS servers, but only has complete all-pair latency information for 468 of the DNS servers. To simulate a larger network, for each one of these 468 DNS servers we create roughly 16 additional peers to represent peers in the same stub networks as the DNS servers. We create these additional peers to form a network of 8,192 peers. We model the latency among hosts within the group as zero to correspond to the minimal latency among hosts in the same network. We model the latency among hosts between groups according to the measured latencies from the Vivaldi data set and we refer this data set as a "King". The minimum, average, and maximum latencies among groups are 2, 165, and 795 milliseconds, respectively. As a timeout value for detecting failed peers, we use a single round trip time to that peer (according to the optimizations in [5]).

Using measurements to create the network model adds realism to the evaluation. At the same time, though, the evaluation only scales to the limits of the measurements. To study the hint caches on systems of much larger scale, we also performed experiments using another network model. We have two different data sets in this model and both of them are significantly larger than above data set. First, we created a matrix of network latency among 8,192 groups by randomly assigning a latency between two groups from the range of 10 to 500 milliseconds and a latency within a group from the range of 1 to 5 milliseconds. We then created an overlay network of 262,144 peers by randomly assigning each peer to one group, keeping the groups balanced. We also created another data set with same latencies among groups and with in a group for 65536 peers that are randomly distributed into 2,048 groups. We refer these data sets as a "Random".

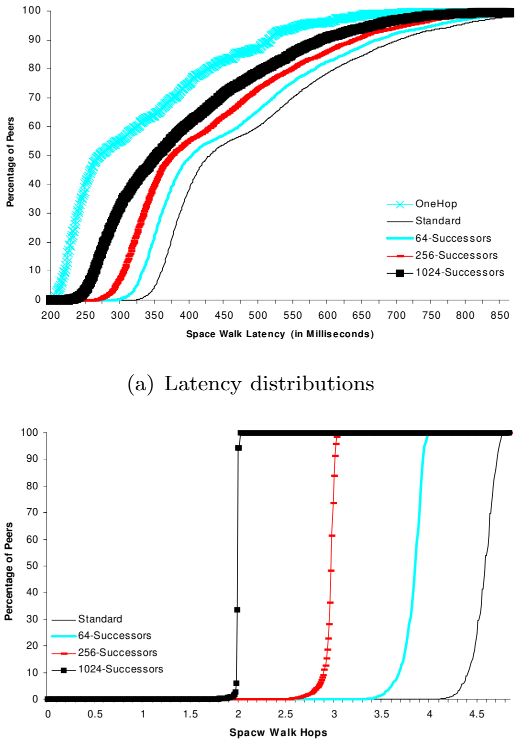
The goal of "Random" data set is to show the performance of various caches at large scale networks that we can simulate on the resources that are available to us. We used two different clusters [21, 30] to scale the computational resources. However, in some of the experiments we couldn't parallelize our simulation easily hence we used smaller "Random" data set with 65,536 peers.

5.2 Local Hint Caches

In this experiment we evaluate the performance of the local hint cache compared with two baseline routing algorithms, "Standard" and "OneHop." "Standard" is the default routing algorithm in Chord++ [5] that optimized for lookup latency by choosing nearest fingers. "OneHop" maintains complete routing tables on all peers [7].

Figures 1(a) and 1(b) show the cumulative distributions for the space walk latencies and hops, respectively, across all peers in the system. Since there is no churn in this experiment, we calculate the latencies and hops after the network stabilizes when reaching the experimental phase; we address churn in the next experiment. Figure 1(a) shows results for "Standard" and "OneHop" and local hint cache sizes ranging from 64–1024 successors; Figure 1(b) omits "OneHop" since it only requires one hop for all lookups with stable routing tables.

Figure 1(a) shows that the local hint caches improve routing performance over the Chord baseline, and that doubling the cache size roughly improves space walk latency by a linear amount. The median



(a) Latency distributions

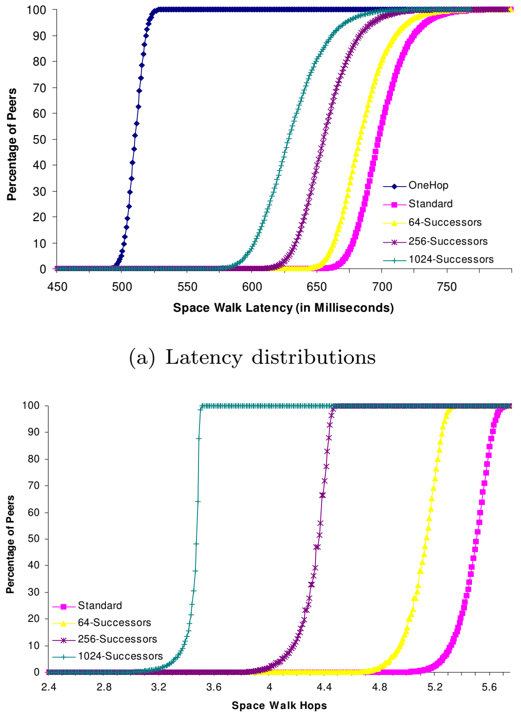
(b) Hop count distributions

Figure 1: Sensitivity of local hint cache size on lookup performance for "King" data set.

space walk latency drops from 432 ms in Chord to 355 ms with 1024 successors in the local hint cache (a decrease of 18%). Although an improvement, the local hint cache alone is still substantially slower than "OneHop", which has a median space walk latency of 273 ms (a decrease of 37% is needed).

Figure 1(b) shows similar behavior for local hint caches in terms of hops. A local hint cache with 1024 successors decreases median space walk latency by 2.5 hops, although using such a cache still requires one more hop than "OneHop".

The performance local hint cache for "Random" data set in Figure 2(a) and Figure 2(b) is similar to the performance of local hint cache for "King" data set in Figure 1(a) and Figure 1(b) respectively. We used 65,536 peers for the "Random" data set to be consistent with other Figures in this section.



(a) Latency distributions

(b) Hop count distributions

Figure 2: Sensitivity of local hint cache size on lookup performance for "Random" data set.

From the graph, we see that doubling the local hint cache size improves the number of hops by at most 0.5. Doubling the local hint cache size reduces hop count by one for half of the peers, and the remaining half does not benefit from the increase. For example, consider a network of 100 peers where each peer maintains 50 other peers in its local hint cache. For each peer, 50 peers are one hop away and the other 50 peers are two hops away. As a result, the space walk hop distance is 1.5 hops. If we increase the local hint cache to 100 peers, then each peer reduces the hop distance for only the 50 peers that were two hops away in the original scenario. In this case, the space walk hop distance is 1.

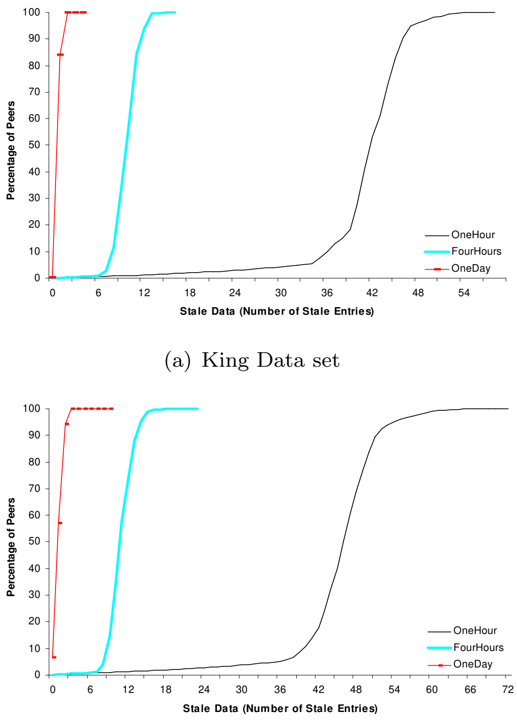
When there is no churn in the system, the lookup performance when measured in terms of hops either remains the same or improves when we double the local hint cache size. The results are more complicated when we measure lookup performance in terms of latency. Most peers improves their lookup latencies to other peers and, on average, increasing local hint cache improves the space walk latency. However, lookup latency to individual peers can increase when we double the local hint cache size in some cases. This is because Internet routing does not necessarily follow the triangular inequality: routing through multiple hops may have lower latency than a direct route between two peers. Since we derive our network latency model from Internet measurements, our latency results reflect this characteristic of Internet routing.

5.3 Staleness in Local Hint Caches

The previous experiment measured the benefit of using the local hint cache in a stable network, and we now measure the staleness in terms of stale entries in the local hint cache and the effect of staleness on lookup performance.

In this experiment, we use a local hint cache size of 1024 successors. To calculate stale data in local hint caches, we ran the simulator with King data set and Random data set with 65,536 peers for an experimental phase of 30 minutes. During the experimental phase, the network experiences churn in terms of peer joins and leaves. We vary peer churn in the network by varying the half life of peers in the system from one hour to one day; we select nodes to join or leave from a uniform distribution.

Figure 3 shows the fraction of stale entries in local hint caches for various system half life times as a cumulative distribution across all peers. We calculated the fraction of stale entries by sampling each peer's local hint cache every second and determining the number of stale entries. We then averaged the samples across the entire simulation run. Each point (x, y) on a curve indicates that y percentage of peers have at most x% stale data in their local hint caches. As expected, the amount of stale data increases as the churn increases. Note that the amount of stale data is always less than the amount calculated from our analysis in Section 3.2 since the analysis conservatively assumes worst case update synchronization.

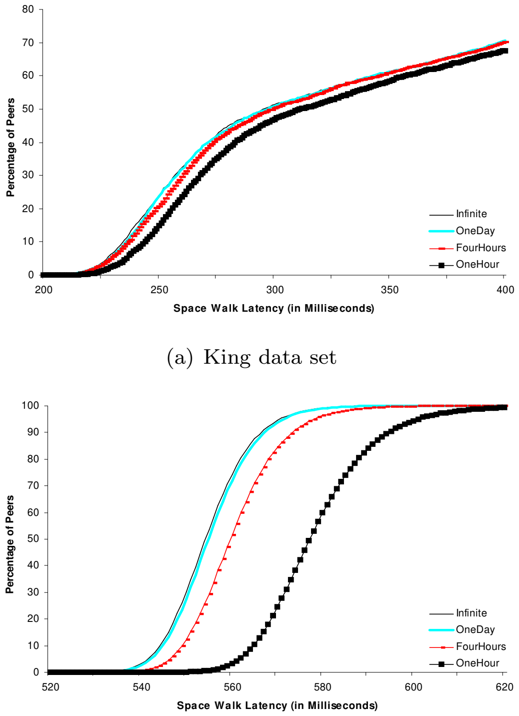


(a) King Data set

(b) Random Data set

Figure 3: Stale data distribution under various churn situations for King and Random data sets.

Figure 4 shows the effect of stale local hint cache entries on lookup performance across all peers for King data set and Random data set with 32768 peers. It shows results for the same system half life times as Figure 3 and adds results for an "Infinite" half life time. An "Infinite" half life means that there is no churn, no stale entries in the hint cache, and therefore represents the best-case latency. At the end of the experiment phase in the simulation, we used the state of each peer's routing table to calculate the distribution of space walk latencies across all peers. Each point (x, y) in the figure indicates that y percentage of peers have at most x space walk latency. We cut off the y-axis at 75% of peers to highlight the difference



(a) King data set

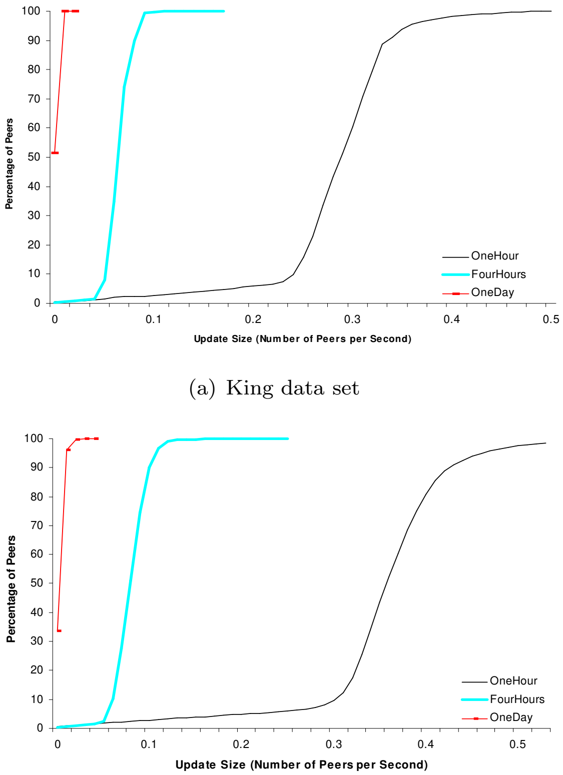
(b) Random data set

Figure 4: Space walk latency distributions under various churn situations.

between the various curves.

The space walk latencies for a four hour half life time are similar to the latencies from the ideal case with no churn (medians differ by only 1.35%). From these results we conclude that the small amount of stale data (1–2%) does not significantly degrade lookup performance, and that the local hint cache update mechanism maintains fresh entries well. As the churn rate increases, stale data increases and lookup performance also suffers. At an one hour half life, lookup performance increases moderately.

Note that the "Infinite" half life time curves in Figure 4(a) and Figure 4(b) performs better than the 1024 successors curve in Figure 1(a) and Figure 2(a) even though one would expect them to be the same. The reason they differ is that the finger table entries in these two cases are different. When we evaluated the local hint cache, we used a routing table with 13 successors and added the remaining successors to create the local hint cache without changing the finger table. When we evaluated the stale data effects in the local hint cache we have 1024 successors from which to choose "nearest" fingers. As a result, the performance is better.



(a) King data set

(b) Random data set

Figure 5: Update traffic distribution under various churn situations.

5.4 Update Traffic

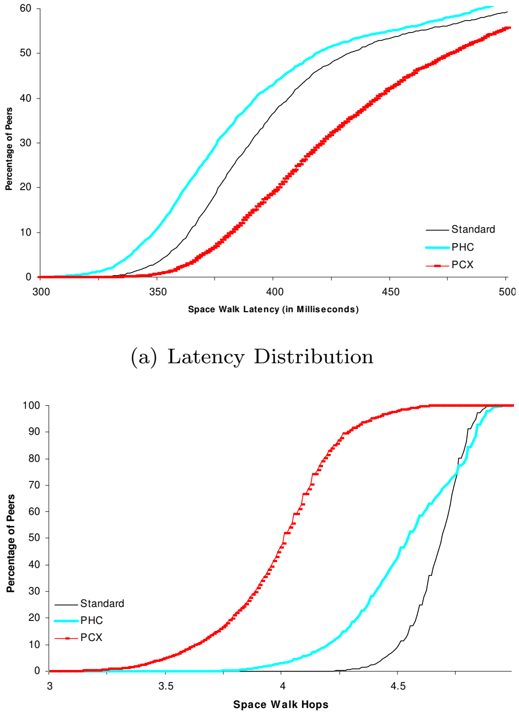
In the previous section we evaluated the effect of stale data on lookup performance under various churn scenarios. In this section we evaluate the update traffic load under various churn scenarios to evaluate the update traffic bandwidth required by large local hint caches.

We performed a similar experiment as in Section 5.3 for King data set and Random data set with 65536 peers. However, instead of measuring stale data entries we measured the update traffic size. We calculated the average of all update samples per second for each peer over its lifetime in terms of the number of entries communicated. Figure 5 presents this average for all peers as cumulative distributions. Each curve in Figure 5 corresponds to a different churn scenario. A point (x, y) on each curve represent the y percentage of peers that have at most x entries of average update traffic. The average update traffic closely matches the estimate from our analysis in Section3.2. The average update traffic (0.4 entries/second) is extremely low even under worst case conditions. Hence, this traffic does not impose a burden on the system.

5.5 Path Hint Caches

Next we evaluate the performance of the path hint cache (PHC) described in Section 3.3 compared to path caching with expiration (PCX) as well as Chord. PCX is the technique of caching path entries described in [25]. When handling lookup requests on behalf of other nodes, PCX caches route entries to the initiator of the request as well as the result of the lookup.

In this experiment, we simulate a network of "King" data set with a 30-minute experimental phase. We study the lower-bound effect of the path hint caches in that we do not perform any application level lookups. Instead, the path hint caches are only populated by traffic resulting from network stabilization. We did not simulate application level lookup traffic to separate its effects on cache performance; with applications performing lookups, the path hint caches may provide more benefit, although it will likely be application-dependent. Since there is no churn, cache entries never expire. To focus on the effect of path caches only, we used a local hint cache size of 13, the size of the standard Chord successor list, and no global hint cache. We collected the routing tables for all peers at the end of the simulations and calculated the space walk latencies and hops.



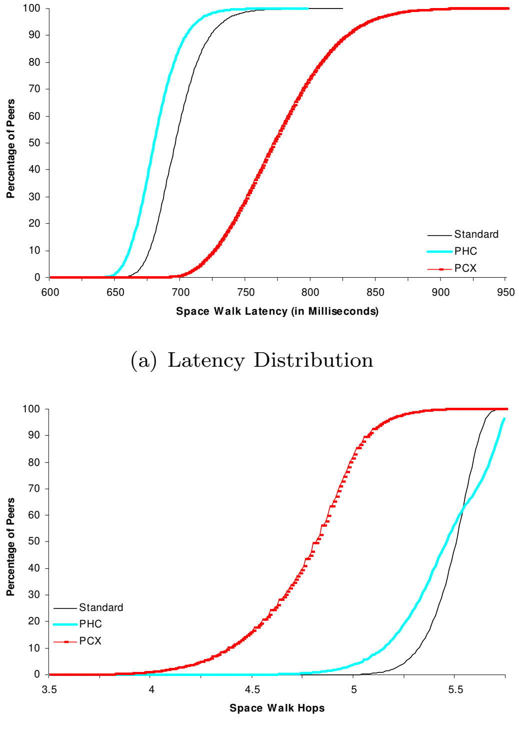
(a) Latency Distribution

(b) Hop count distributions

Figure 6: Lookup performance of path caching with expiration (PCX), path hint cache (PHC), and standard Chord for "King" data set.

Figure 6(a) shows the cumulative distribution of space walk latencies across all peers at the end of the simulation for the various path caches and standard Chord. Each point (x, y) in this figure indicates that y percent peers have at most x space walk latency. From these results we see that, as expected, the path hint cache improves latency only marginally. However, the path hint cache is essentially free, requiring no communication overhead and a small amount of memory to maintain.

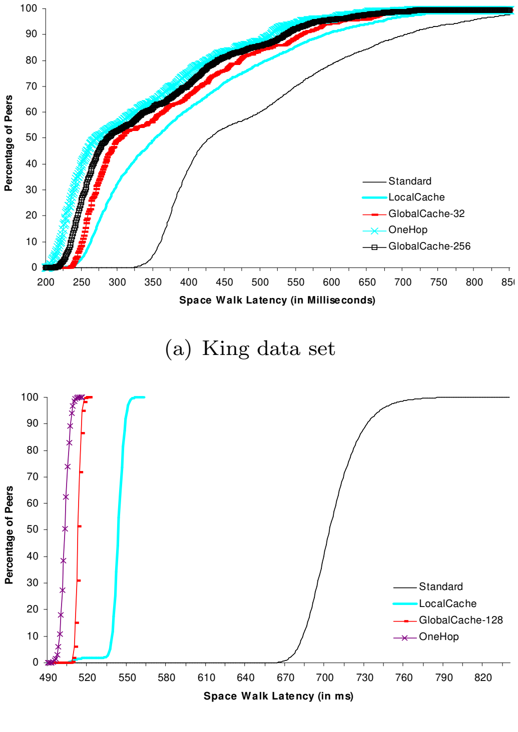
We also see that PCX performs worse even than Chord. The reason for this is that PCX optimizes for hops and caches routing information independent of the latency between the caching peer and the peer being cached. The latest version of Chord and our



(a) Latency Distribution

(b) Hop count distributions

Figure 7: Lookup performance of path caching with expiration (PCX), path hint cache (PHC), and standard Chord for “Random” data set.



(a) King data set

(b) Random data set

Figure 8: Global hint cache performance in different network models.

path hint caches use latency to determine what entries to place and use in the caches and in the routing tables. For peers with high latency, it is often better to use additional hops through low-latency peers than fewer hops through high-latency peers.

Figure 6(b) shows this effect as well by presenting the cumulative distribution of space walk hops across all peers for the various algorithms. Each point (x, y) in this figure indicates that y percent peers have at most x space walk hops. Using the metric of hops, PCX performs better than both Chord and PHC. Similar to results in previous work incorporating latency into the analysis, these results again demonstrate that improving hop count does not necessarily improve latency. Choosing routing table and cache entries in terms of latency is important for improving performance.

We did the same experiment replacing the ”King” data set with the “Random” data set with 65,536 peers and Figures 7 shows the results of this experiment. These results are qualitatively similar to the results of above “King” data set.

The path hint cache are small and each peer aggressively evicts the cache entries to minimize the stale data. Hence the effects of stale data on lookup request performance is marginal.

5.6 Global Hint Caches

Finally, we evaluate the performance of using the global hint cache together with the local and path hint caches. We compare its performance with “Standard" Chord and "OneHop". In this experiment, we simulated 8,192 peers with both 32 and 256 entries in their local hint caches. We have two different configurations of local hint caches to compare the extent to which the global hint cache benefits from having more candidate peers from which to select nearby peers to place in the global hint cache. (Note that the global hint cache uses only the second half of the nodes in the local hint cache to select nearest nodes; hence, the global hint cache uses only 16 and 128 entries to choose nearest peers in the above two cases.) Each peer constructs its global cache when it joined the network as described in Section 3.4. We collected the peer's routing tables once the network reached a stable state during the experimentation phase, and calculated the space walk latencies for each peer from the tables.

Figure 8(a) shows the cumulative distributions of space walk latencies across all peers for the various algorithms. The "Standard", "OneHop", "LocalCache", "GlobalCache-32", and "GlobalCache-256" curves represent Chord, the "OneHop" approach, a 1024-entry local hint cache, a 32-entry global hint cache with a 256-entry local hint cache, and a 256-entry global hint cache with a 32-entry local hint cache. Comparing the size of local hint caches used to populate the global hint cache, we find that the median space walk latency of "GlobalCache-256"is 287 ms and "GlobalCache-32" is 305 ms; the performance of the global hint cache improved only 6% when it has more peers in the local hint cache to choose the nearest peer.

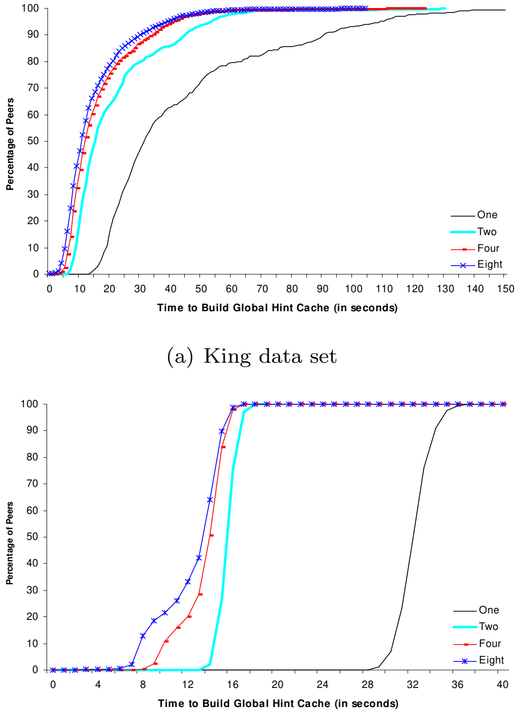
We also did same experiment with Random data set for 262,144 peers. We set the local hint cache size to 128 instead of 1024 to grow the global hint caches to few thousand peers which is the expected global hint cache size when the network has few million peers. This is the maximum network size we could able to simulate on our cluster machines. At this point, the main memory becomes the bottle-neck for our simulation. The results are presented in Figure 8(b) and the global hint cache performed close to the "OneHop" approach (the median space walk latency of global hint cache is 2% more than the median space walk latency of "OneHop" approach).

Comparing algorithms, we find that the median latency of the global hint cache comes within 6% of the "OneHop" approach when the global hint cache uses 128 of 256 entries in the local hint cache to choose nearby peers. Although these results are from a stable system without churn, the amount of stale data in the global hint cache under churn is similar to the local hint cache under churn because both of them use a similar update mechanism. Hence, the global hint cache performance under churn is same or a little better than the local hint cache performance under churn because global hint cache is filled with the nearest peers as opposed to the random peers in local hint cache. As a result, the effect of stale data in the global hint cache is negligible for a one-day system half life time and four-hour system half life time. Overall, our soft-state approach approaches the lookup latency of algorithms like "OneHop" that use significantly more communication overhead to maintain complete routing tables.

5.6.1 Global Hint Cache Build Time

Since we contact closer peers while constructing the global hint cache, one can build this cache within a few minutes. To demonstrate this, we calculated the time to build the global hint cache for King data set with 8,192 peers and Random data set with 262,144 peers. Figure 9 presents the results of this experiment as a distribution of cache build times. Each point (x, y) on a curve indicates that y percentage of peers needs at most x seconds to build the cache. Each peer has around 500 peers in its the global hint caches and 16 peers in its local hint cache for King data set and in the Random data set each peer has around 1500 peers in the global hint cache and 128 peers in the local hint cache.

In the King data set, on the average it took 45 seconds to build the global hint cache. A peer can speed up this process by initiating walks from multiple peers from its routing table in parallel. The curves labeled "Two", "Four", and "Eight" represent the cache build times with two, four, and eight parallel walks, respectively. As expected, cache build time reduces as we increase the number of parallel walks. The median reduces from 32 seconds for single walk to 12 seconds for four parallel walks. We see only



(a) King data set

(b) Random data set

Figure 9: Global hint cache build time

a small benefit of increasing the parallel walks after four parallel walks.

The global hint cache build times for Random data set has similar trends as the global hint cache build times for King data set. Even though global hint cache sizes are bigger in the Random data set the build time is less comparing with the King data set. The Random data set has more peers in the local hint cache which improve the chances of finding a nearest peer for a given peer which in turn improves the build time.

5.6.2 Network Coordinates

So far we have assumed that, when populating the global hint caches, peers are aware of the latencies among all other peers in the system. As a result, the results represent upper bounds. In practice, peers will likely only track the latencies of other peers they communicate with, and not have detailed knowledge of latencies among arbitrary peers. One way to solve this problem is to use a distributed network coordinate system such as Vivaldi [4]. Of course, network coordinate systems introduce some error in the latency prediction. To study the effect of coordinate systems for populating global hint caches, we next use Vivaldi to estimate peer latencies.

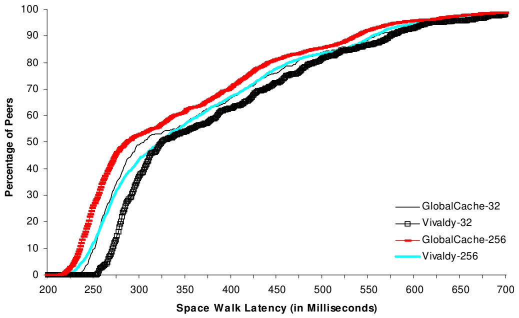


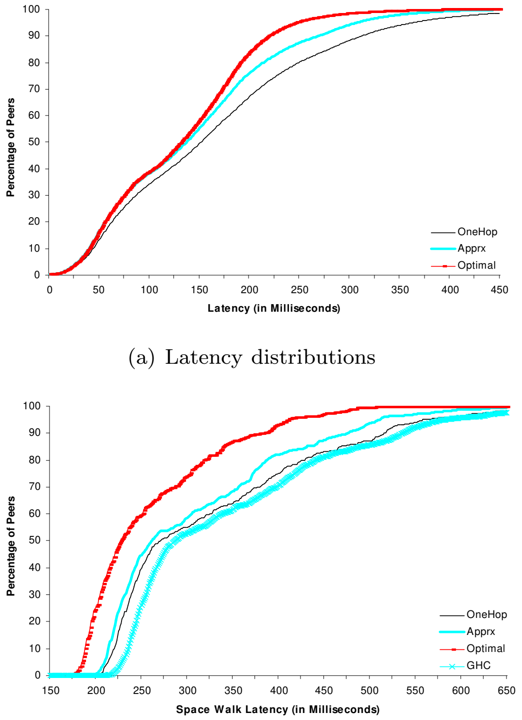
Figure 10: Effects of Network Coordinates

In our simulation we selected the nearest node according to network latency estimated according to the Vivaldi network coordinate system, but calculated the space walk time using actual network latency. We did this for the "GlobalCache-32" and "GlobalCache-256" curves in Figure 8. Figure 10 shows these curves and the results using Vivaldi coordinates as "Vivaldi-32" and "Vivaldi-256". The performance using the coordinate system decreases 6% on average in both cases, showing that the coordinate system performs well in practice.

5.7 Shun Pikes

In section 4 we described ways to exploit the detours to improve the lookup performance even further than the "OneHop" approach. In this section we evaluated the shun pikes for the King data set. The latency distributions between all pairs are presented in Figure 11(a) and Figure 11(b) shows the performance results of our new lookup algorithm.

First we calculated the shortest path between all



(a) Latency distributions

(b) Space walk latency distributions

Figure 11: Effects of shun pikes

pair of peers in the data set assuming that each peer has full knowledge of other peers. We just simply ran the shortest path algorithm on network latency matrix. The result of this is presented in Figure 11(a) as the "Optimal" labeled curve. We then calculated the shortest paths from each peer to all other peers assuming each peer has 64 peers in its local hint cache and approximately 64 peers in the global hint cache. The resultant latency distributed is represented as "Apprx" curve in Figure 11(a). The curve "OneHop" shows the the network latency distribution. Overall for 40% of pairs shortest path latency improved over the one hop latency. The "Apprx" and "Optimal" performed similarly for smaller latencies and differ a bit at higher latencies. Though theoratically the "Apprx" should be constant factor away from the optimal case in general Internet like graphs. However, in this case we got similar performace as the optimal case when each node has partial infomrationabout the enitre graph.

The Figure 11(b) shows the space walk latency for one hop, shortest path latencies, our new lookup algorithm and our global and local hint caches . The "OneHop" and "Optimal" curves represent the one hop and shortest path latency approaches. The "GHC" curve represents the space walk latency of our global hint cache along with the local hint cache that is same as the "GlobalCache-256" curve in Figure 8(a). The "Apprx" curve represents the space walk latency of our approach where each peer has only partial information about the entire network and each peer tries to optimize the lookup as described in section 4. The median space latency for our caches, one hop, our new lookup algorithm and shortest paths is 287, 271, 262 and 228 milliseconds. Our new lookup algorithm improved approximately 10% over our global and local hint caches and it improved around 3% over one hop approach. As expected our lookup algorithm performed approximately 15% slower than the optimal because we need one extra hop to complete the lookup. Overall our new lookup algorithm performed a little better than the one hop approach.

6 Conclusions

In this paper, we describe and evaluate the use of three kinds of *hint caches* containing route hints to improve the routing performance of distributed hash tables (DHTs): *local hint caches* store direct routes to neighbors in the ID space; *path hint caches* store direct routes to peers accumulated during the natural processing of lookup requests; and *global hint caches* store direct routes to a set of peers roughly uniformly distributed across the ID space.

We simulate the effectiveness of these hint caches as extensions to the Chord DHT. Based upon our simulation results, we find that the combination of hint caches significantly improves Chord routing performance with little overhead. For example, in networks of 4,096 peers, the hint caches enable Chord to route requests with average latencies only 6% more than algorithms like "OneHop" that use complete routing tables, while requiring an order of magnitude less bandwidth to maintain the caches and without the complexity of a distributed update mechanism to maintain consistency.

7 Acknowledgments

We would like thank the anonymous reviewers of our WCW submission for their valuable feedback for improving this paper. We would also like to express our gratitude to Marvin McNett for system support for performing our simulation experiments, and to Frank Dabek for providing us with the network latency information used in all simulations. Support for this work was provided in part by AFOSR MURI Contract F49620-02-1-0233 and DARPA FTN Contract N66001-01-1-8933.

References

[1] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *2nd International Workshop on Peer-to-Peer Systems*, Feb. 2003.

[2] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proceedings of SIGMETRICS*, June 2000.

[3] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *19th ACM Symposium on Operating Systems Principles*, Oct. 2003.

[4] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris. Practical, distributed network coordinates. In *proceedings of Second Workshop on Hot Topics in Networks*, Nov. 2003.

[5] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a dht for low latency and high throughput. In *ACM/USENIX Symposium on Networked Systems Design and Implementation*, Mar. 2004.

[6] F. D. Emil Sit and J. Robertson. Usenetdht: A low overhead usenet server. In *3rd International Workshop on Peer-to-Peer Systems*, Feb. 2004.

[7] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In *ACM/USENIX Symposium on Networked Systems Design and Implementation*, Mar. 2004.

[8] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable p2p dht through increased memory and background overhead. In *2nd International Workshop on Peer-to-Peer Systems*, Feb. 2003.

[9] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized and peer-to-peer web cache. In *21st ACM Symposium on Principles of Distributed Computing*, July 2002.

[10] F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal hash table. In *2nd International Workshop on Peer-to-Peer Systems*, Feb. 2003.

[11] M. F. Kaashoek and R. Morris. http://www.pdos.lcs.mit.edu/chord/.

[12] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *19th ACM Symposium on Operating Systems Principles*, Oct. 2003.

[13] S. S. Krishna P. Gummadi and S. D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *2nd Internet Measurement Workshop*, Nov. 2002.

[14] J. Li, B. T. Loo, J. M. Hellerstein, M. F. Kaashoek, D. R. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *2nd International Workshop on Peer-to-Peer Systems*, Feb. 2003.

[15] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Observations on the dynamic evolution of peer-to-peer networks. In *First International Workshop on Peer-to-Peer Systems*, Mar. 2002.

[16] B. T. Loo, S. Krishnamurthy, and O. Cooper. Distributed web crawling over dhts. Technical Report UCB/CSD-4-1305, UC Berkeley, 2004.

[17] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *21st ACM Symposium on Principles of Distributed Computing*, July 2002.

[18] G. S. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed hashing in a small world. In *4th USENIX Symposium on Internet Technologies and Systems*, Mar. 2003.

[19] P. Maymounkov and D. Mazires. Kademlia: A peer-to-peer information system based on the xor metric. In *1st International Workshop on Peer-to-Peer Systems*, Mar. 2002.

[20] J. McCaleb. http://www.overnet.com/.

[21] M. McNett. https://ramp.ucsd.edu/group.

[22] A. Mizrak, Y. Cheng, V. Kumar, and S. Savage. Structured superpeers: Leveraging heterogeneity to provide constant-time lookup. In *IEEE Workshop on Internet Applications*, June 2003.

[23] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *ACM/USENIX Symposium on Networked Systems Design and Implementation*, Mar. 2004.

[24] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *proceedings of ACM SIGCOMM*, Aug. 2001.

[25] M. Roussopoulos and M. Baker. Cup: Controlled update propagation in peer-to-peer networks. In *USENIX Annual Technical Conference*, June 2003.

[26] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Nov. 2001.

[27] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: A case for informed internet routing and transport. *IEEE Micro*, 19, 1999.

[28] 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 ACM SIGCOMM*, Aug. 2001.

[29] M. Thorup and U. Zwick. Compact routing schemes. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, 2001.

[30] C. UCSD. http://activeweb.ucsd.edu/.

[31] B. Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.