

Accelerating Lookups in P2P Systems using Peer Caching

Supratim Deb[†], Prakash Linga[‡], Rajeev Rastogi[†], Anand Srinivasan^{*}

[†]Bell Labs Research India, Bangalore, e-mail: {*supratim, rastogi*}@*alcatel-lucent.com*

[‡]Moka5TM, CA, USA, e-mail: *plinga@gmail.com*

^{*}Google Inc., Bangalore, India, e-mail: *anands@gmail.com*

Abstract—Many structured peer-to-peer (P2P) systems have been proposed as distributed hash tables (DHTs) for fast and efficient lookup of queries. In this paper, we propose a novel technique for improving average lookup times in P2P systems by caching additional neighbor pointers based on peer access frequencies.

In particular, we address the problem of each peer choosing the k best pointers to store (in addition to its index pointers) to minimize the average query lookup times. We focus on two popular P2P systems, namely Pastry and Chord: we exploit the inherent structure of these systems to develop efficient, scalable algorithms for optimally choosing the k additional pointers. Simulations with Chord and Pastry demonstrate that our algorithms are very effective in reducing the lookup times significantly. Our approach can be used in tandem with other techniques such as item caching and replication, and is particularly useful for applications such as name services in mobile environments or location services, where we can expect a low churn rate for peers and a relatively higher churn rate for items.

I. INTRODUCTION

Peer-to-peer (P2P) architecture and technologies have received considerable attention as an efficient way for providing large-scale distributed storage and information retrieval. P2P systems consist of Internet hosts that act as peers and form an overlay network. Significant amount of research has been directed towards designing *structured* P2P systems [20], [22], [26], [18], [14]. Many of these implement distributed hash tables (DHTs), which are highly scalable and offer efficient, low latency lookups (*i.e.*, search and retrieval of data). However, some applications such as naming services require lower latencies than can be offered by these DHTs [6]. In this paper, we focus on the problem of improving the lookup performance of such structured P2P systems.

Every node in a P2P system¹ is responsible for a certain set of items and the system has an appropriate protocol for efficient routing of queries for items. The design objective is to have every node store pointers to certain other nodes in the system such that a query for an item reaches the destination node (*i.e.*, the node with the queried item) in as few hops

as possible. There is an inherent trade-off between the worst-case number of hops required by a query, and the number of neighbor pointers maintained by a node [23].

There are two key components in the design of P2P systems: the design of the routing protocol and the choice of neighbors. In two popular DHTs, Pastry [20] and Chord [22], each node stores $O(\log n)$ neighbors in its routing table, where n is the number of nodes. These neighbors are chosen intelligently to ensure that queries take $O(\log n)$ hops in the worst case. However, the average query latency (in terms of hops) can be markedly different depending on the popularity distribution of nodes. In this work, we present efficient, scalable algorithms to minimize average query latency by caching additional neighbors based on frequencies of node accesses. Intuitively, the existing neighbors based on the underlying P2P system (henceforth, referred to as *core* neighbors) attempt to minimize the worst-case query latency, while the additional neighbors selected by our algorithms try to reduce the average query latency by accounting for the non-uniformity of query popularities. We refer to these additional neighbors as *auxiliary* neighbors since they supplement the core neighbors for accelerating query lookups.

The number of neighbors maintained by each node is directly related to the maintenance cost induced by peers joining and leaving the system. In this paper, we provide algorithms that take the size of the routing table as input, and provide the appropriately-sized set of optimal auxiliary neighbors. The size of the routing table is a design choice, and needs to be traded off with the cost of maintaining the routing table. The maintenance cost consists of the overhead messages for ensuring that all the entries point to live neighbors in the presence of churn. This cost might be incurred when a node leaves (maybe, after making an announcement), or when nodes refresh their neighbor list. For instance, at regular intervals, a P2P node may send ping messages to check the liveness of its neighbors; the neighbors found dead after pinging are replaced by new neighbors. If the churn is high, then the refresh interval needs to be smaller to ensure that stale neighbors in the routing table do not result in unanswered queries. Clearly, the maintenance cost of the routing table grows with the size of the routing table (refer to [13] for an excellent evaluation of how the maintenance cost varies with different system parameters). To keep the maintenance cost roughly of the same order

[‡]This work was done when Prakash Linga was visiting Bell Labs India.

^{*}This work was done when Anand Srinivasan was working with Bell Labs India.

¹Unless otherwise stated, throughout this paper, we use the term “P2P system” to refer to structured P2P systems, and in particular DHTs such as Pastry [20] and Chord [22].

as before, it is reasonable to keep the number of auxiliary neighbors around the same as the number of core neighbors. In [12], the authors propose a protocol that adjusts the size of the routing table based on a given bandwidth budget for maintenance. Our approach can be used to populate the routing table entries on top of such an approach.

It is important to recognize that there are several different ways to improve lookup performance in P2P systems, including replication and caching of items. In item caching, a node caches items that have been queried previously, and in a replication-based approach, the popular items are replicated at several nodes. [16] presents a replication scheme that can achieve $O(1)$ hops in lookup performance, assuming a zipfian distribution for item popularities. However, item caching and replication schemes have some inherent drawbacks, especially, in scenarios where the items are large or the items are updated frequently and peer churn is relatively low. Clearly, if items are large and storage at nodes is limited, then storage will be a bottleneck. Moreover, large items imply high replication costs. Further, even if items are small, if they are updated frequently, then item caching will result in a large number of stale copies, and replication will incur a large cost due to the necessary updates. One example of such a scenario would be a P2P implementation [6], [17], [4], [15] of the Domain Name System (DNS) with support for mobile IP. In this example system, IP addresses of domain names may change frequently but the DNS servers (which are the peers in this case) can be expected to be reasonably stable. An approach based on caching additional neighbor pointers does not suffer from the above-mentioned problems even in the face of frequent item updates or large items. Another drawback with item caching and replication is that such approaches exclusively benefit the queries to the particular items that are cached or replicated. On the other hand, *caching a pointer to a peer not only benefits queries to items in the cached peer, but also queries to items in other peers that can be reached faster via the cached peer*. Nevertheless, our approach does not preclude the use of replication and item caching and can be used in conjunction with such schemes for added benefit.

As hinted above, one of the motivations for improving query lookup times in P2P systems comes from the proposed use of P2P architectures for implementing DNS [6], [17], [4], [15]. In [15], the authors compare hierarchical and DHT-based naming services, and note that an ideal naming service needs to combine features of both: though P2P systems are more resilient to attacks, they suffer from poor query response times as compared to hierarchical DNS. Our work is motivated by the fact that today's DNS servers obtain enormous performance gains by caching IP addresses of other DNS servers [10]. We leverage this idea to accelerate query lookups in P2P systems as well. Interestingly, the dynamic nature and the structure of P2P systems not only pose some unique challenges (quite different from hierarchical DNS implementations), but also provide scope for designing intelligent algorithms that exploit the underlying structure and routing policy.

In this paper, we focus primarily on Pastry [20] and

Chord [22]. However, the basic technique is also applicable to other P2P systems that employ similar routing policies. More precisely, the techniques presented for Pastry can be directly applied to Tapestry [26] and PGrid [1], and the techniques for Chord are applicable to SkipGraphs [2].

There are three main contributions of this paper.

- 1) We propose a novel technique for optimizing average lookup times in Pastry and Chord by caching additional neighbor pointers based on the access frequencies of peers. Our algorithms are efficient and scale well with the number of nodes in the system. Any node S wishing to choose the optimal set of k auxiliary neighbors can compute the desired pointers in $O(nk)$ time in Pastry and $O(nk \log n)$ time in Chord, where n is the number of distinct nodes for which S has seen queries. We also provide an incremental version of our algorithm for Pastry that runs in $O(k)$ time and can be used to efficiently maintain the optimal set of neighbors as peer popularities change and peers join or leave the system.
- 2) We also extend our algorithms to support multiple QoS classes by handling queries that require guaranteed worst-case lookup times.
- 3) We present extensive simulation results, which clearly demonstrate that our schemes provide substantial improvements in the query lookup times as compared to an approach that does not account for the access frequencies in choosing the auxiliary neighbors. For example, with $\log n$ auxiliary neighbors in Chord, the improvement is up to 57% when the item popularities are zipfian with parameter 1.2. In other words, our algorithm for Chord results in response times less than half of what is achieved without taking access frequencies into account.

The rest of the paper is organized as follows. In Section II, we provide a quick overview of Chord and Pastry, and also discuss other related work. In Section III, we formally define the problem addressed in this paper. In Sections IV and V, we present optimal algorithms for auxiliary neighbor selection in Pastry and Chord, respectively. We present the simulation results in Section VI. Finally, we conclude in Section VII.

II. BACKGROUND AND RELATED WORK

Since we focus on Pastry and Chord, we first describe the routing protocols in these two P2P systems in more detail. We then discuss other related work on improving lookup times in these or other similar P2P systems.

A. Pastry Routing

Pastry [20] is a self-organizing P2P system where each node in the Pastry overlay is assigned a b -bit identifier. The node id is used to indicate the location of the node in a circular id space ranging from 0 to $2^b - 1$. The node id is assigned randomly when a node joins. As a result each node becomes responsible for a chunk of the id space. Each item, which can be a file for example, is also associated with a unique id by hashing the file content into the id space. We refer to the id

associated with an item as a key. The Pastry routing problem is the following: given a query at a node for an item with a given id, locate the node in the system that is responsible for the queried item. For the purpose of routing the queries, node ids and keys are viewed as a sequence of digits with base 2^d for some d . To simplify explanation, we assume $d = 1$ in the following paragraph.

To efficiently locate queried items, each node in Pastry stores a few pointers to other nodes in its routing table. The table has a maximum of b rows, where the i^{th} ($i = 0, 1, \dots, b - 1$) row contains the IP address of some node that matches with the given node in the first i bits but differs in the $(i + 1)^{\text{th}}$ bit. As noted in [20], the uniform distribution of the node ids ensures that, with n nodes in the system, on average, only $\log n$ rows of the routing table have entries. Given such a routing table, the routing works as follows. Queries are routed to the node that is numerically closest to the queried key. At each step, a node forwards the query to a node whose id shares with the key a prefix that is at least one digit longer than the prefix that the key shares with the present node's id. This procedure ensures that, in the steady state, queries are routed in $\log n$ hops. The Pastry protocol has several additional features to efficiently handle node joins and leaves. The interested reader can refer to [20] for more details.

B. Chord Routing

In the following, we describe a slight variant of Chord. The nodes and the keys (*i.e.*, the item ids) in Chord have identifiers structured in an identifier circle. Suppose the identifier lengths are b . A key k can be assigned to k 's predecessor (*i.e.*, the first node whose id is equal to k , or precedes k in the id space). Thus, a lookup for a key has to visit the key's predecessor, *i.e.*, the node whose id most closely precedes the key. A node with id x in Chord keeps $\log n$ neighbors whose ids lie at exponentially increasing fractions of the id space away from itself. The node with the numerically smallest id among the ones with id within the range $x + 2^i$ and $x + 2^{i+1}$ (modulo 2^b) is used as the i^{th} neighbor of x . Then, the Chord routing policy is the following: for a query with id v at x , the next hop is the neighbor of x that is closest to v and is between x and v in the clockwise direction on the ring. The choice of Chord pointers ensures that, with n nodes in the system, it takes a maximum of $\log n$ hops for the query to reach the destination (in the steady state). We refer the reader to [22] for the complete details, including handling of node joins and leaves.

C. Other Related Work

There have been other schemes proposed in the literature for improving the lookup times in P2P systems [24], [14], [25], [19], [21], [16], [11], [5]. We briefly discuss some of them here, and point out the similarities and differences from our work.

[16] proposes a "structured" item replication scheme called Beehive to provide $O(1)$ lookup times for items following a zipfian distribution. The basic intuition is to have more replicas

of popular items, thus driving down the average lookup time for a query to $O(1)$ hops. (The authors have also shown that significant improvements in DNS lookup times can be obtained by implementing DNS on top of Beehive [17].) One of the issues with this approach is the overhead of maintaining the replicas. Since popular items are replicated more, the overhead of keeping those replicas updated is also larger. Thus, in a scenario where items are frequently changing, replication can result in significant overhead. Our work addresses this issue by storing pointers to nodes rather than items. Further, the Beehive replication scheme requires the query pattern to be zipfian across all the nodes, whereas our scheme computes the optimal set of auxiliary neighbors locally, and hence, simply depends on the local query access pattern. Nevertheless, as we remarked in Section I, our approach can be used in conjunction with replication for additional benefit.

In [11], Leong *et al.* proposed another variant of Chord called EpiChord. There are two significant differences from Chord. First, they remove the $O(\log n)$ state-per-node restriction. In particular, instead of maintaining the regular $O(\log n)$ Chord neighbors, they maintain a cache of all the known nodes, and classify them into different slices depending on their distance. The slices are similar to the slices obtained in Chord by partitioning the node space using the neighbors. Additionally, in EpiChord, several lookups are initiated in parallel for each query. This greatly reduces the chances of a failed query and also significantly improves the lookup time. Though the idea of caching known peers is similar to our approach, there is a basic difference: EpiChord places no restriction on the number of node pointers that can be cached, and hence, does not provide an algorithm for caching a fixed number of pointers. In order to circumvent the maintenance cost of storing a large number of pointers, they assign an ad-hoc "lifetime" to each cache entry, in very much the same way as TTLs are assigned to cache entries in today's DNS.

There has also been some prior work on caching of node pointers based on peer access history [24], [21], [5]. In [24] and [21], the authors focus on *unstructured* P2P systems and present simple algorithms for caching node pointers based on access frequencies. These algorithms are primarily used to substitute flooding techniques for answering queries with more structured forwarding techniques. Ghosh *et al.* [5] focus on Chord and base their work on the assumption that nodes can be clustered into disjoint groups that primarily contact each other. They view these clusters as mini-Chord rings and select $\log W$ neighbors from these clusters (where W is the cluster size). This allows them to ensure $\log W$ latency for a query within the cluster and $W \log(n/W)$ for outside queries. However, their approach is more restrictive than ours because they assume that nodes can be clustered into disjoint groups, which is stronger than assuming a zipfian distribution for queries. In fact, if nodes can indeed be clustered in such a manner, then we believe that our approach will be significantly beneficial as well.

III. PROBLEM DEFINITION AND TERMINOLOGY

In this section, we define the exact problem of choosing auxiliary neighbors, and also the terminology that we use in the following sections.

Let s be any P2P node, and let N_s denote the set of its core neighbors. Let V ($|V| = n$) denote the set of nodes for which s has seen queries (not including s) and maintains access frequencies. Note that access frequencies can be easily maintained by s based on past history of accesses within a time window. (We address this issue in more detail later.) For each node v in V , we are given binary² ids of length b , and an access frequency f_v .

We use the term d_{uv} to denote the “distance” from node u to node v in terms of number of hops. Since the computation is being done at node s , this distance is only an estimate of the actual distance between u and v . Typically, a simple estimate can be computed easily at s based only on the ids of u and v and the underlying routing policy. We elaborate on this distance function later while discussing the routing policies of Pastry and Chord in Sections IV and V. We also use $d(u, S)$ to denote the minimum distance between the node u and a set of nodes S , i.e., $d(u, S) = \min_{v \in S} d_{uv}$. Similarly, $d(S, u) = \min_{v \in S} d_{vu}$. Given these definitions, our goal is to identify a set of auxiliary neighbors $A_s \subseteq V - N_s$ with $|A_s| = k$, such that

$$\text{Cost}(A_s) = \sum_v f_v(1 + d(v, N_s \cup A_s)) \quad (1)$$

is minimized. Note that $d(v, N_s \cup A_s)$ is an estimate of the minimum distance between v and the neighbors (core and auxiliary) of s . Thus, $(1 + d(v, N_s \cup A_s))$ is the estimated distance of v from s taking into account the one hop from s to its neighbor. For simplicity, we refer to $f_v(1 + d(v, N_s \cup A_s))$ as the weighted distance of v . Thus, the goal is to select a subset of V of size k as auxiliary neighbors such that the total weighted distance for all the nodes in V is minimized.

1) *Routing policy with auxiliary neighbors:* The choice of the k best auxiliary neighbors clearly depends on how they will be used. In this paper, we assume that there is no change in the underlying routing policy, and the auxiliary neighbors are used for routing in the same manner as the core neighbors. This assumption allows our algorithms to be incrementally deployed in a large P2P system without any issues.

2) *Implementation Considerations:* Before describing our algorithms, we discuss two important implementation issues: (1) maintenance of access frequencies, and (2) maintenance of auxiliary neighbors. Maintenance of access frequencies is crucial to our algorithms and can be done by noting the node containing the queried item for every query. If the number of accessed nodes is very large, then a node can simply store the top- n frequent nodes for a suitable n based on its storage limitation. Online computation of the top- n frequent nodes can be done using standard streaming algorithms [3]. A smaller n also reduces the computation overhead of our algorithms, which are linear in n . However, the resulting solution may be sub-optimal because some nodes are ignored.

²We assume binary ids only for ease of exposition. It is straightforward to extend our algorithms to deal with an arbitrary base.

Note that the entries in the routing table of a node may become stale due to churn in the system. For instance, as new nodes join in Chord, the first node at a particular distance may change. Further, some entries may point to nodes that have already left the system. To overcome these issues, each node pings its core neighbors at regular intervals and also periodically re-initializes all the entries (see [22] for more details). In the presence of auxiliary neighbors, the ping process can be modified to check the auxiliary neighbors as well. The stale auxiliary entries can be marked or removed from the routing table. These entries will then be fixed the next time the auxiliary neighbor selection algorithm is executed. The algorithm can be invoked either periodically or based on some criteria that determines that the system has undergone a significant change since the previous computation of the auxiliary neighbors.

IV. ALGORITHMS FOR PASTRY

We now present algorithms for selecting the k best auxiliary neighbors in Pastry. As described in Section II-A, in Pastry, a query for an item with id I is forwarded to a neighbor whose id has the longest prefix match with I . In order to compute the optimal set of k auxiliary neighbors that minimizes (1), we need to estimate the distance between any two nodes u and v in terms of number of hops. We use $b - l$ as an estimate of this distance, where l is length of the longest prefix match. For example, the distance between 4-bit ids 1011 and 1111 is 3 because l in this case is just 1. $b - l$ is a reasonable estimate in the absence of any additional information, since in the worst case, while routing and “fixing” each bit at a time, the number of hops incurred can go up to $b - l$. Thus, this would be the upper bound in the steady state. As the source node does not have knowledge of the exact position of every node, it is reasonable to use this upper bound as a measure of the actual distance. We remark that the choice of k pointers remains the same even if the distances are scaled by a constant factor.

We first present a simple $O(nk^2b)$ algorithm, based on dynamic programming, to obtain the optimal set of k auxiliary pointers. We then build upon this basic algorithm and obtain an optimal $O(nkb)$ greedy algorithm. Here, n represents the number of distinct peers for which the source node (where the algorithm is executed) has seen queries. In Section IV-C, we present an incremental version of this greedy algorithm to adaptively maintain the set of auxiliary neighbors as nodes join and leave or as node popularities change. In Section IV-D, we show how to adapt our algorithm to handle queries that must be answered within a certain number of hops.

A. A Simple $O(nk^2b)$ algorithm

Given the ids of the nodes in V , we construct a tree T that is a trie of these ids. Figure 1 illustrates such a trie. Note that each node in V corresponds to a leaf of T , such that the path from the root of T to the leaf determines the node’s id. Henceforth, we use “nodes” and “leaf vertices” interchangeably to refer to the corresponding Pastry nodes.

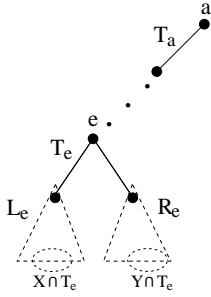


Fig. 2. A portion of the trie depicting the scenario in the proof of (P).

Proof of (P): Let $P(T_a, j)$ denote the j optimal pointers within subtree T_a , i.e., pointers for whom the optimal cost $C(T_a, j)$ is attained. Then, (P) states that $P(T_a, j-1) \subseteq P(T_a, j)$. We use the following lemma (proved in [7]) in our proof. The lemma essentially states that the reduction in cost due to adding a neighbor x cannot increase as the number of selected neighbors increases.

Lemma 4.1: Suppose $x \in A \cap L_a$ and $A \cap L_a \subset A' \cap L_a$. Then, $C(T_a, A' - \{x\}) - C(T_a, A') \leq C(T_a, A - \{x\}) - C(T_a, A)$.

Suppose $P(T_a, j-1) \subseteq P(T_a, j)$ is not true. Then, we show that we can obtain another set of $j-1$ pointers that is a subset of $P(T_a, j)$ and has the same cost as $P(T_a, j-1)$. Let $X = P(T_a, j) - P(T_a, j-1)$ and $Y = P(T_a, j-1) - P(T_a, j)$. Clearly, $Y \neq \emptyset$ and $X \neq \emptyset$. Now, it is easy to see that there exists a vertex e in T_a such that $T_e \cap X \neq \emptyset$, $T_e \cap Y \neq \emptyset$, and one of its (two) subtrees contains no vertices from X while the other contains no vertices from Y . Wlog, let $L_e \cap Y = \emptyset$ and $R_e \cap X = \emptyset$. Figure 2 illustrates this scenario. Also, let $Q_j = T_e \cap P(T_a, j)$ and $Q_{j-1} = T_e \cap P(T_a, j-1)$.

Let $x \in (L_e \cap X)$ be the leaf in L_e for which $\Delta_x = C(T_e, Q_j - \{x\}) - C(T_e, Q_j)$ is minimum. Similarly, let $y \in (R_e \cap Y)$ be the leaf in R_e for which $\Delta_y = C(T_e, Q_{j-1} - \{y\}) - C(T_e, Q_{j-1})$ is minimum. We now show that $\Delta_x = \Delta_y$ holds. Intuitively, this means that y in $P(T_a, j-1)$ can be replaced with x without any increase in cost. Repeating this procedure for all nodes in Y , we can transform $P(T_a, j-1)$ into a subset of $P(T_a, j)$ without hurting its optimality.

We prove $\Delta_x = \Delta_y$ by showing that an assumption to the contrary leads to a contradiction. Suppose $\Delta_x > \Delta_y$. Note that $C(T_e, Q_{j-1} - \{y\}) = \Delta_y + C(T_e, Q_{j-1})$. Also,

$$\begin{aligned} & C(T_e, Q_{j-1}) - C(T_e, Q_{j-1} \cup \{x\}) \\ & \geq C(T_e, Q_j - \{x\}) - C(T_e, Q_j) \\ & \geq \Delta_x \end{aligned} \quad (5)$$

The first inequality follows from Lemma 4.1 since x is in L_e and $L_e \cap Q_{j-1} \subset L_e \cap Q_j$. Again using Lemma 4.1, we have

$$\begin{aligned} & C(T_e, Q_{j-1}) - C(T_e, Q_{j-1} \cup \{x\}) \\ & \leq C(T_e, Q_{j-1} - \{y\}) - C(T_e, Q_{j-1} - \{y\} \cup \{x\}), \end{aligned}$$

Hence, $C(T_e, Q_{j-1} - \{y\} \cup \{x\})$

$$\begin{aligned} & \leq C(T_e, Q_{j-1} - \{y\}) - (C(T_e, Q_{j-1}) - C(T_e, Q_{j-1} \cup \{x\})) \\ & = \Delta_y + C(T_e, Q_{j-1} \cup \{x\}) \\ & \leq \Delta_y + C(T_e, Q_{j-1}) - \Delta_x \quad (\text{from (5)}) \\ & < C(T_e, Q_{j-1}) \quad (\text{because } \Delta_y < \Delta_x) \end{aligned}$$

Thus, we have $C(T_e, Q_{j-1} - \{y\} \cup \{x\}) < C(T_e, Q_{j-1})$. Since x and y are both in subtree T_e , the cost of nodes outside T_e do not change as a result of swapping y with x . Thus, it follows that $C(T_a, P(T_a, j-1) - \{y\} \cup \{x\}) < C(T_a, P(T_a, j-1))$. However, this contradicts the optimality of $P(T_a, j-1)$. Similarly, we can show that $\Delta_x < \Delta_y$ also leads to a contradiction.

Thus, it follows that $\Delta_x = \Delta_y$, and we can show that $C(T_a, P(T_a, j-1) - \{y\} \cup \{x\}) = C(T_a, P(T_a, j-1))$, using similar arguments as above. This implies that we can replace y in $P(T_a, j-1)$ with x without hurting its optimality. This increases the overlap between $P(T_a, j)$ and $P(T_a, j-1)$ by 1. Repeating this argument, we can replace all the nodes in Y with nodes in X without increasing the cost of $P(T_a, j-1)$. This proves that property (P) is true.

C. Incremental Algorithm for Handling Changing Popularities

Recall from Section IV-B that the optimal set of k neighbors is computed for every subtree of T . Note that as long as nothing changes in a subtree, the computed set remains optimal. In other words, if a node's popularity changes (or it is inserted/deleted), then the only vertices affected are the ones whose subtree contains this node. These are exactly the vertices that are on the path from the root of T to the affected leaf. This observation provides us with a simple algorithm for re-computing the optimal set as popularities change. In particular, the algorithm presented in Section IV-B is modified to take an additional parameter I , the id of the affected peer. Each vertex checks whether I is a leaf within its subtree; if not, it simply returns the previously-computed values of the cost and pointers, and does not invoke the algorithm on its children. Thus, new computations are performed only on the vertices along the path corresponding to I . The running time of this algorithm is $O(bk)$, because the processing cost at each vertex in the trie is $O(k)$, and the number of vertices at which the optimal set is re-computed is at most b .

D. QoS-aware Routing

It is also easy to extend our algorithm to handle different query classes. Given that QoS-sensitive applications such as VoIP, IPTV, and video on demand are gaining prominence, it is easy to envision real-time applications that require certain queries to be answered within a fixed time period and hence within a certain number of hops. It is also reasonable to expect such queries to be small in number, and hence destined for a small, specific set of nodes. The new minimization problem is now a constrained version of the original minimization problem stated in Section III: given per-node access frequencies and delay bounds, the goal is to identify a set of auxiliary neighbors that minimize the average lookup time

while ensuring that the per-node delay bounds are met. The algorithm, presented in Section IV-B, can be easily adapted to handle this case. Note that the delay bounds translate to restrictions on the vertices of the trie. In particular, if a node has a delay bound of x , then the subtree of height x that contains the corresponding leaf must have a auxiliary neighbor. This additional constraint can be easily added to the algorithm by marking such subtrees, and modifying the algorithm to select at least one pointer from such subtrees. This modification can be easily accomplished without any increase in complexity.

V. ALGORITHMS FOR CHORD

We now present the auxiliary neighbor selection algorithm for Chord. In Chord, all the nodes are placed in a clockwise fashion on a ring in increasing order of their ids. The Chord routing policy is as follows: for a query to node v at s , the next hop is the neighbor of s closest to v , and between s and v in the clockwise direction. (Here, we are using node ids to refer to the nodes.) In other words, the next hop chosen is $\arg \min_{w \in N_s} \{(v - w) \bmod 2^b\}$. The choice of the core neighbors ensures that, in the steady state, a query originating at node u and destined for a node v requires a maximum of d_{uv} hops, where d_{uv} is defined as follows.

$$d_{uv} = 1 + \lceil \log((v - u) \bmod 2^b) \rceil. \quad (6)$$

This distance is an upper bound in the steady state, and does not make any assumptions about the actual positions of the nodes. (d_{uv} is equivalent to the position of the leftmost ‘1’ in $(v - u) \bmod 2^b$, and unlike Pastry, this distance function is not symmetric.)

In the rest of this section, for simplicity, we assume that the auxiliary neighbor computation is done at the node with id zero (*i.e.*, zero in all b bits of the identifier), without loss of generality. We refer to this node as the zero-node. Further, we refer to the m^{th} immediate successor of the zero-node in the clockwise direction on the ring as node m .

We begin by presenting a simple algorithm based on dynamic programming to solve the above recurrence. This algorithm has complexity $O(n^2k)$ and hence, does not scale very well with the number of nodes. However, it provides the basis for an $O(n(b + k \log b) \log n)$ algorithm, which we present in Section V-B.

A. A Simple $O(n^2k)$ Algorithm

Before presenting the details, we introduce some notation. Let N_0 denote the set of core neighbors of the zero-node. Also, let $C_l(m)$ be the cost of optimally placing l auxiliary neighbor pointers when only the m immediate successors of the zero-node are considered. Considering the nodes in increasing order of node ids, we obtain the following.

$$C_l(m) = \min_{1 \leq j \leq m} [C_{l-1}(j-1) + s(j, m)] \quad (7)$$

$$\text{where, } s(j, m) = \sum_{l=j+1}^m f_l d(\{j\} \cup N_0, l) \quad (8)$$

The minimum cost that can be obtained using k pointers is clearly $C_k(n)$. In the above equations, $s(j, m)$ denotes the cost

of routing queries to all nodes between the j^{th} and the m^{th} successor when there is a pointer to the j^{th} successor and no auxiliary pointers between the j^{th} and the m^{th} successor. The key intuition behind recurrence (7) is simple: if j is the largest id among the auxiliary neighbors of the zero-node that have ids smaller than that of m , then all queries to a node with id between that of j and m are routed through j or an existing neighbor from the set N_0 that is closer. On the other hand, queries to nodes $1, 2, \dots, j-1$ cannot use the pointer at j for routing, and hence the cost contributed by such nodes is recursively handled by $C_{l-1}(j-1)$. Minimizing over all possible j gives the cost $C_l(m)$.

The recurrence given by (7) can be solved in two steps.

- 1) Compute $s(j, m)$ for all pairs (j, m) such that $j < m$.
- 2) Solve (7) using a simple dynamic programming algorithm consisting of two nested *for* loops.

The complexity of the first step is $O(n^2)$ and that of the second step is $O(n^2k)$, thus giving an overall complexity of $O(n^2k)$.

B. An $O(n(b + k \log b) \log n)$ Algorithm

We now present an improved algorithm to solve the recurrence given by (7) that scales with the number of nodes as $O(n \log n)$. There are two obstacles to doing so. First, tabulating all the possible $O(n^2)$ values of $s(j, m)$ for solving (7) is clearly $O(n^2)$. Secondly, given the values of $s(j, m)$, solving the recurrence given by (7) using a naive dynamic programming is again $O(n^2k)$. The second challenge can be overcome using a result from [9], which can be adapted to solve (7) using $O(kn \log n)$ operations (assuming the values of $s(j, m)$ are known or can be computed in $O(1)$ time). To overcome the first challenge, we avoid tabulating all the $O(n^2)$ values of $s(j, m)$ a priori. Rather, we construct suitable data-structures using $O(bn \log n)$ operations such that $s(j, m)$ (for any j and m) can be computed using $O(\log b)$ operations.

Overview of the algorithm: We now provide a brief sketch of the algorithm. We first present the data structures that are maintained to compute $s(j, m)$ for a given (j, m) pair. In the following paragraphs, let $F(j)$ denote the cumulative frequencies until node j , *i.e.*, $F(j) = \sum_{l \leq j} f_l$.

For simplicity, let us first consider the case in which there is no core neighbor of the zero-node between node j and node m . In this case, $s(j, m)$ can be written as follows.

$$s(j, m) = \sum_{r=1}^{d_{jm}} r \times \left(\begin{array}{l} \text{total frequency of nodes at distance } r \\ \text{and id numerically less than id of } m \end{array} \right)$$

Let $p_j(r)$ denote the node that is farthest from j among all the nodes that are at distance at most r from j . Then, the total frequency of nodes at a distance r is simply $F(p_j(r)) - F(p_j(r-1))$. Hence, the above equation can be re-written as follows.

$$s(j, m) = \sum_{r=1}^{d_{jm}-1} r(F(p_j(r)) - F(p_j(r-1))) + d_{jm}(F(m) - F(p_j(d_{jm}-1))) \quad (9)$$

The relationship given by (9) points us to a recipe for the minimal data structures that we need to maintain to compute

$s(j, m)$ for a given (j, m) pair. First, we need to maintain the points $p_j(r)$ for all $r \leq d(j, N_0)$. Second, we need to maintain the cumulative frequencies $F(j)$.

Since distance is upper-bounded by b , the space requirement for the points $p_j(r)$ ($1 \leq j < n$, $1 \leq r \leq b$) is at most $O(nb)$. Now, each $p_j(r)$ can be computed using binary search over the n ordered points. Because there are at most b such points for all j , the overall complexity for computing the points $p_j(r)$ is $O(nb \log n)$. Cumulative frequencies can be easily computed using $O(n)$ operations.

Let us now consider the case in which there is a core neighbor between the nodes j and m . Let j_1, j_2, \dots, j_r be the core neighbors of the zero-node (the node for which we are running the algorithm) that lie between j and m . Then, $s(j, m)$ can be obtained as follows.

$$s(j, m) = s(j, j_1 - 1) + s(j_1, j_2 - 1) + \dots + s(j_r, m) \quad (10)$$

This follows because a pointer placed at node j can only help a node i such that there is no core neighbor between j and i . Otherwise, the routing in Chord dictates that queries to i are routed through the closest core neighbor that lies between j and i . The relationship given by (10) gives us a two-step procedure to compute $s(j, m)$ using the $p_j(k)$'s and $F(j)$'s. First, we determine the core neighbors between which m lies using binary search, and then use (9) to compute each of the terms in (10).

Thus, the necessary data structures can be constructed using $O(nb \log n)$ operations, and as shown in [7], can be used to compute $s(j, m)$ for any (j, m) pair using $O(\log b)$ operations.

In order to improve the straightforward dynamic programming algorithm described in Section V-A, we use a key result from [9], which shows how to solve the following recurrence using $O(n \log n)$ computations assuming the function $w(i, j)$ is concave.

$$f(j) = \min_{1 \leq i \leq j-1} [f(i) + w(i, j)], \quad \forall j \leq n$$

It is easy to show that $s(j, m)$ is also concave, and though our recurrence is two dimensional, it is not difficult to extend the algorithm of [9] to obtain an $O(nk \log n)$ algorithm for solving (7). A detailed description of our algorithm and a proof for the following theorem can be found in [7].

Theorem 5.1: The recurrence (7) can be solved using $O(n(b + k \log b) \log n)$ operations and $O(n(b + k))$ space.

C. QoS-aware Routing.

Recall from Section IV-D that the QoS-aware routing problem considered is a constrained version of the original optimization problem in which some nodes have delay bounds associated with them. As described in [7], our algorithm can be easily adapted to handle these constraints.

VI. EXPERIMENTAL EVALUATION

The purpose of our experiments is two-fold: (1) to demonstrate the gains obtained by using our algorithms, and (2) to provide insight on how the benefit varies with different parameters such as number of nodes and the number of auxiliary neighbors. We first describe our experimental setup and then present the results of our experiments.

A. Experimental Setup

To validate our algorithms on Pastry, we used the open-source Pastry implementation called Freepastry [8], while we implemented our own event-driven simulator for Chord.

Given a set of nodes and items with randomly-generated identifiers, we assign popularities to items based on a zipfian distribution, and the queries are samples from this distribution. Depending on the underlying application, the popular items might be different at different peers. We have thus considered two possibilities for ranking the items based on their popularities: (i) identical at all nodes, and (ii) different at different nodes. While we have covered both of these cases in our simulations, due to space constraints, the results for the former case are demonstrated in our plots with Pastry, and the results for the latter are demonstrated in our plots with Chord. For the latter case, we used five distinct item popularity lists in our experiments, each with identical zipfian parameter, but with different rankings of the items. Each node is then assigned one of the five item popularity lists at random. This is done primarily to simulate the fact that the item popularities may vary across nodes, but can be expected to follow a zipfian distribution with a similar parameter. In addition, the experiments were run in two distinct modes: a stable mode with no peer insertions and deletions, and a churn-intensive mode where peers are inserted and deleted regularly.

We used the following default parameters for our experiments: number of nodes $n = 1024$, number of auxiliary neighbors $k = \log n$ and the item popularities follow a zipfian distribution with parameter $\alpha = 1.2$. We used 32-bit binary ids for the experiments. In addition, we show the effects of using $\alpha = 0.91$ in the Pastry plots, and high churn in the Chord plots.

Performance Metric: To illustrate the advantages of our schemes, we compared our algorithms for choosing the k auxiliary neighbors with a scheme where the k additional neighbors are chosen without taking the access frequencies into account. We refer to this scheme as the *frequency-oblivious* scheme, and have implemented it as follows. In Chord, if k is $r \log n$, then the frequency-oblivious approach selects r auxiliary neighbors at random in the range $(2^i, 2^{i+1}]$ for all i .³ Similarly, in Pastry, the r auxiliary neighbors are chosen for each prefix match of a certain length. In all our plots, the performance metric used is the *percentage reduction in the average number of hops compared to the frequency-oblivious scheme*.

B. Results with Pastry

As mentioned before, we present plots for $\alpha = 1.2$ as well as for $\alpha = 0.91$. Note that the improvement for $\alpha < 1$ can be expected to be lower because hashing results in the node popularities being almost uniform even though the item popularities are zipfian. However, we saw significant gains

³Recall that the $\log n$ core neighbors in Chord are chosen by selecting the first node in the range $(2^i, 2^{i+1}]$ for all i . The technique described here is similar to the one adopted in Kademia [14].

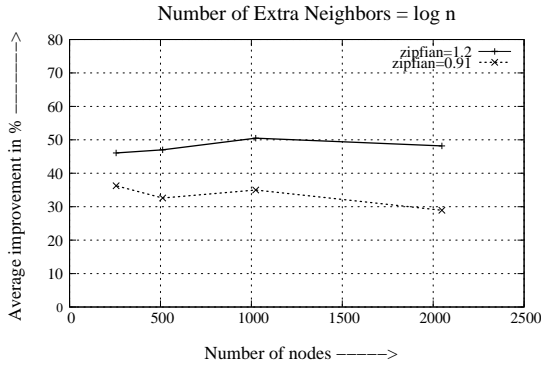


Fig. 3. Pastry: improvement in average no. of hops vs. the no. of nodes. ($n = 1024$.)

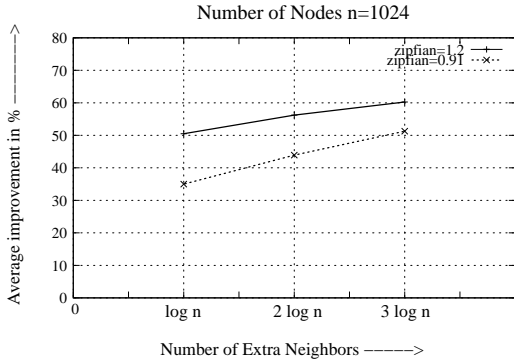


Fig. 4. Pastry: improvement in average no. of hops vs. the no. of extra neighbors. ($n = 1024$.)

even for $\alpha < 1$. The general trends for $\alpha < 1$ and $\alpha > 1$ are similar to those observed for 0.91 and 1.2, respectively.

Variation with n : We measured the percentage reduction (over a frequency-oblivious approach) in the average number of hops as n is varied from 256 to 2048, with k fixed at $\log n$ (which is same as the expected number of core neighbors). Figure 3 shows the observed improvements for α values of 1.2 and 0.91. Clearly, the auxiliary neighbors selected by our algorithm gives a significant improvement. In particular, for $n = 2048$ and $\alpha = 1.2$, we see around 49% reduction in the average query lookup times, *i.e.*, the average number of hops required for answering a query reduces to almost half. Even for $\alpha = 0.91$, we see substantial improvements of up to 29%.

Variation with k : We also studied the effect of varying the number of auxiliary neighbors. Figure 4 shows the observed improvement over a frequency-oblivious approach for $k \in \{\log n, 2 \log n, 3 \log n\}$, where n is fixed at 1024. For $\alpha = 1.2$, the observed improvement increases from 50% for $k = \log n$ to 60% for $k = 3 \log n$.

C. Results for Chord

Here, in addition to results for stable systems, we also present the improvements observed in a churn-intensive system. (Note that a system with low churn rate shows a behavior similar to a stable system.) The way we simulate churn is

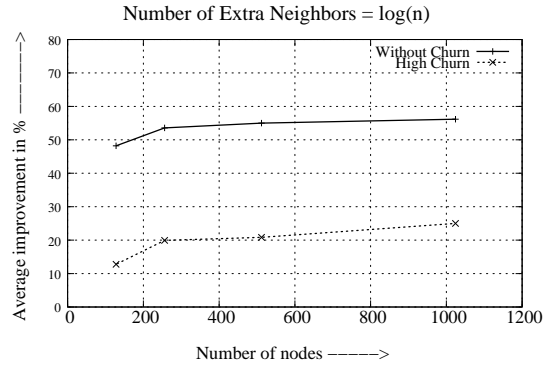


Fig. 5. Chord: improvement in average no. of hops vs. the no. of nodes. ($k = \log n$.)

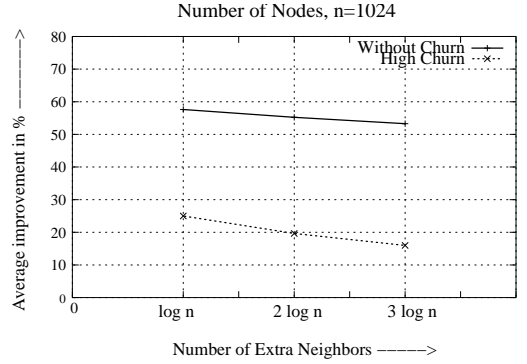


Fig. 6. Chord: improvement in average no. of hops vs. the no. of extra neighbors. ($n = 1024$.)

similar to the way it was done in [13]: essentially, the n nodes crash and re-join the system alternately. Once a node joins (or fails), it remains alive (or dead) for a mean duration of 900 seconds with the duration being sampled from an exponential distribution. On an average, we simulate 4 queries per second. The above parameters equate to a churn rate of around 2 peer-joins and peer-leaves per second, *i.e.*, one join and one leave for every two queries on an average. The stabilization interval is set to 25 seconds, and the auxiliary neighbor computation period is set to 62.5 seconds. Our results show that significant performance benefits can be obtained even under this high churn rate.

Variation with n : We vary n from 128 to 1024, and fix k as $\log n$. As shown in Figure 5, the improvements are again quite significant. In particular, we see up to 57% reduction in the average number of hops in a stable system with 1024 nodes with our algorithm. Even in the presence of very high churn, we see improvements of up to 25%.

Variation with k : We fixed n at 1024 and varied k from $\{\log n, 2 \log n, 3 \log n\}$. Figure 6 shows the observed improvements. As shown, the observed improvement reduces as the number of additional neighbors increases. For example, in the high churn case, the improvement is around 26% for $k = \log n$, whereas the improvement drops to around 17% for $k = 3 \log n$. Intuitively, when the number of additional

pointers is small, then it is more important that the optimal ones are chosen to provide the most benefit. As k increases, the benefit obtained by even a random choice of neighbors increases because the chance of an optimal neighbor being chosen randomly increases. This leads to a reduction in the relative benefit observed by using our approach to select the auxiliary neighbors.

On the other hand, the improvements observed for Pastry actually keep increasing as k increases (see Figure 4). This behavior is an artifact of the FreePastry implementation and can be explained as follows. Routing in FreePastry is locality-aware, *i.e.*, if there is more than one candidate node for the next hop, then the candidate node that is live and closest to the current node is picked. Thus, increasing the number of core neighbors does not decrease the hop count as much as increasing the number of auxiliary neighbors. In our Chord implementation, we select the candidate node in the routing table that is closest to the destination, and hence the observed behavior is different.

VII. CONCLUSIONS

In this paper, we have proposed a novel technique for improving average lookup times in P2P systems by caching auxiliary neighbors based on the access frequencies of peers. Our approach is particularly useful for applications such as name services in mobile environments or location services, where we can expect low churn rate for peers and relatively higher churn rate for items. We have presented efficient, scalable algorithms for optimally choosing k auxiliary neighbors in prefix-based P2P systems like Pastry and small-world based P2P systems like Chord. We have also shown how to adapt these algorithms to handle queries that require worst-case performance guarantees. Simulations with Chord and Pastry demonstrate that our algorithms are very effective in reducing the lookup times significantly.

Our results give rise to several questions that warrant further investigation. We opted for locally optimal algorithms, primarily to ensure that they can be easily deployed in an incremental fashion. However, since our algorithms do not take into account the auxiliary neighbors of other peers, the “globally” optimal choice of auxiliary neighbors can be different. The challenge would be to design a globally optimal decentralized algorithm that can be incrementally deployed.

REFERENCES

- [1] K. Aberer. P-grid: A self-organizing access structure for P2P information systems. In *CoopIS*, 2001.
- [2] J. Aspnes and G. Shah. Skip graphs. In *SODA*, 2003.
- [3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.
- [4] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A Layered Naming Architecture for the Internet. In *SIGCOMM*, 2004.
- [5] A. Bejan and S. Ghosh. Self-optimizing DHTs using request profiling. In *OPDIS*, 2004.
- [6] R. Cox, A. Muthitacharoen, and R. Morris. Serving DNS using chord. In *IPTPS*, 2002.
- [7] S. Deb, P. Linga, R. Rastogi, and A. Srinivasan. Peer caching for faster lookups in peer-to-peer systems (full version). *Bell Labs Technical Report*, 2006.
- [8] FreePastry. <http://freepastry.org>.
- [9] D. Hirschberg and L. Larmore. The least weight subsequence problem. *J. Computing*, 16:628–638, 1987.
- [10] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. In *SIGCOMM IMW*, 2001.
- [11] B. Leong, B. Liskov, and E. Demaine. Epichord: Parallelizing the chord lookup algorithm with reactive routing state management. In *ICON*, 2004.
- [12] J. Li, J. Stribling, R. Morris, and M. Kaashoek. Bandwidth-efficient management of DHT routing tables. In *NSDI*, 2005.
- [13] J. Li, J. Stribling, R. Morris, M. Kaashoek, and T. Gil. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *INFOCOM*, 2005.
- [14] P. Maymounkov and D. Mazieres. Kademlia: A P2P information system based on the XOR metric, 2002.
- [15] V. Pappas, D. Massey, A. Terzis, and L. Zhang. A comparative study of current DNS with DHT-based alternatives. In *INFOCOM*, 2006.
- [16] V. Ramasubramanian and E. G. Sirer. Beehive: Exploiting power law query distributions for O(1) lookup performance in P2P overlays. In *NSDI*, pages 331–342, 2004.
- [17] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the internet. In *SIGCOMM*, 2004.
- [18] S. Ratnasamy, M. Handley P. Francis, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, 2001.
- [19] M. Rousopoulos and M. Baker. Cup: Controlled update propagation in P2P networks. In *USENIX*, 2003.
- [20] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale P2P systems. In *Middleware*, 2001.
- [21] K. Sripanidkulchai, B. Maggs, and H. Zhang. Efficient content location using interest-based locality in P2P systems. In *INFOCOM*, 2003.
- [22] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable P2P lookup service for internet applications. In *SIGCOMM*, 2001.
- [23] J. Xu, A. Kumar, and X. Yu. On the fundamental tradeoffs between routing table size and network diameter in P2P networks. *IEEE JSAC*, 22(1):151–163, 2004.
- [24] B. Yang and H. Garcia-Molina. Improving search in P2P networks. In *ICDCS*, 2002.
- [25] H. Zhang, A. Goel, and R. Govindan. Incrementally improving lookup latency in distributed hash table systems. In *SIGMETRICS*, pages 114–125, 2003.
- [26] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: an infrastructure for fault-tolerant wide-area location and routing. *Technical Report, U.C.Berkeley*, 2001.