EpiChord: Parallelizing the Chord Lookup Algorithm with Reactive Routing State Management

Technical Report MIT-LCS-TR-963

Ben Leong
benleong@mit.edu

Barbara Liskov
liskov@csail.mit.edu

Erik D. Demaine
edemaine@mit.edu

***Abstract*— EpiChord is a DHT lookup algorithm that demonstrates that we can remove the $O(\log n)$-state-per-node restriction on existing DHT topologies to achieve significantly better lookup performance and resilience using a novel reactive routing state maintenance strategy that amortizes network maintenance costs into existing lookups and by issuing parallel queries. Our technique allows us to design a new class of unlimited-state-per-node DHTs that is able to adapt naturally to a wide range of lookup workloads. EpiChord is able to achieve $O(1)$-hop lookup performance under lookup-intensive workloads, and at least $O(\log n)$-hop lookup performance under churn-intensive workloads even in the worst case (though it is expected to perform better on average).**

Our reactive routing state maintenance strategy allows us to maintain large amounts of routing state with only a modest amount of bandwidth, while parallel queries serve to reduce lookup latency and allow us to avoid costly lookup timeouts. In general, EpiChord exploits the information gleaned from observing lookup traffic to improve lookup performance, and only sends network probes when necessary. Nodes populate their caches mainly from observing network traffic, and cache entries are flushed from the cache after a fixed lifetime.

Our simulations show that with our approach can reduce both lookup latencies and path lengths by a factor of 3 by issuing only 3 queries asynchronously in parallel per lookup. Furthermore, we show that we are able to achieve this result with minimal additional communication overhead and the number of messages generated per lookup is no more than that for the corresponding sequential Chord lookup algorithm over a range of lookup workloads. We also present a novel token-passing stabilization scheme that automatically detects and repairs global routing inconsistencies.

I. Introduction

In recent years, more than a dozen DHT lookup algorithms and routing topologies have been proposed [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. DHTs are important to distributed systems research because they offer a scalable and efficient routing and object location platform for self-organizing peer-to-peer overlay networks. DHTs are expected to become a fundamental building block of future large-scale distributed systems. While most of the initial DHT research was directed towards minimizing the amount of routing state per node, more recent research has demonstrated that it is reasonable to attempt to store a global lookup table at every node to achieve one-hop lookup, when network churn is relatively low or if enough bandwidth is available, since local storage is relatively cheap [13].

The DHT designs and the various DHT-related techniques that have been proposed, e.g., proximity neighbor selection [14], synthetic coordinates [15], [16], erasure coding [17] and integrated P2P transport protocol [18], essentially allow us to trade off different amounts of storage and background maintenance bandwidth for better or worse lookup performance in a variety of ways. In this paper, we describe EpiChord, a DHT that demonstrates that we can remove the state storage restriction on $O(\log n)$-state DHTs[1] to achieve better lookup performance using a novel reactive routing state maintenance strategy and by issuing multiple queries asynchronously in parallel. Our technique allows us to design a new class of unlimited-state-per-node DHTs that is able to adapt naturally to a wide range of lookup workloads. EpiChord is able to achieve $O(1)$-hop lookup performance under lookup-intensive workloads, and at least $O(\log n)$-hop lookup performance under churn-intensive workloads even in the worst case, though it is expected to perform better on average.

While existing DHTs tend to decouple the lookup process from routing state maintenance and adopt a proactive routing state management strategy where nodes probe all (or at least most of) their routing entries periodically to ensure that they are alive, EpiChord employs a *reactive* routing state management strategy where routing state maintenance costs are amortized into the lookup costs. Nodes rely mainly on observing lookup traffic and on piggyback-

[1] It is known that limiting the amount of state stored per node to $O(\log n)$ limits the average lookup path length to no better than $O(\log n / \log\log n)$ hops per lookup. Koorde [10] achieves this $O(\log n / \log\log n)$-hop lower bound.

ing additional network information on query replies to keep their routing state up-to-date under reasonable traffic conditions. EpiChord only sends probes as a backup mechanism if lookup traffic levels are too low to support the desired level of performance.

Our reactive routing state maintenance strategy does not keep routing state quite as up-to-date as a proactive strategy, and therefore we use parallel lookups to ameliorate the costs of keeping outdated routing state. In particular, there is a synergistic relationship between large ($> O(\log n)$) state and parallel lookups in our approach: while parallel queries allow us to avoid lookup timeouts due to stale routing entries, we can afford to issue parallel queries without generating excessive amounts of lookup traffic only because our large routing state reduces the number of hops per lookup and thereby the number of lookup messages.

Although one might expect a parallel lookup algorithm to generate significantly more lookup traffic and thereby consume significantly more network bandwidth, we show that we are able in practice to achieve significantly better lookup performance on average (both in terms of lookup path length and latency) than that for the corresponding sequential Chord lookup algorithm with comparable amounts of lookup traffic.

Our goal in this work is not to design the perfect DHT. Rather, our main objective is to explore and quantify the performance-cost trade-offs in moving from an $O(\log n)$-state-per-node DHT topology to an unlimited-state-per-node architecture, by adopting a reactive routing state management strategy and using parallel queries. Consequently, we compare EpiChord to the optimal[2] sequential Chord lookup algorithm. Our parallel lookup algorithm is simple and effective, and our reactive approach to routing state maintenance allows our DHT to adapt naturally to a range of lookup workloads.

II. Overview

Like Chord [2], EpiChord is organized as a one-dimensional circular address space where each node is assigned a unique node identifier (id). As shown in Figure 1, the node responsible for a key is the node whose id most closely follows the key, which we also call the *successor*[3]. We use the cryptographic hash function SHA-1 [19] to determine the node id of a new node. SHA-1 ensures that with high probability, the node ids do not collide (when the address space is sufficiently large, i.e. 128 bits) and are uniformly distributed over the entire circular id address space.

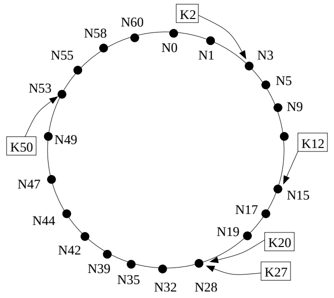

Fig. 1. Circular identifier address space with twenty nodes and five keys.

A. Basic Lookup Algorithm

To look up a given id, node x initiates p queries in parallel to the node immediately succeeding id and to the $p-1$ nodes preceding id, within the set of nodes known to it (see Figure 2). Probing the succeeding node gives us a chance of locating the destination node in one hop.

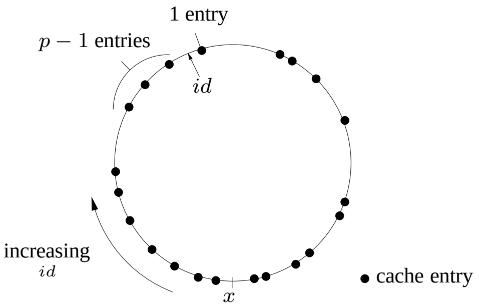

Fig. 2. Initial cache entries returned from cache for node x for a lookup of id.

We adopt two simple policies to learn new routing entries. (i) When a node first joins the network, it obtains a full cache transfer from one of its two immediate neighbors. (ii) Nodes gather information by observing lookup traffic: a node updates its cache based on information returned by queries and adds an entry to the cache each time it is queried by a node not already in the cache.

When contacted, a probed node will respond to x as follows:

- If it owns id, it will simply say so and respond with the value associated with id (if one exists) and information about its current immediate predecessor.

[2] By optimal, we mean that we ignore Chord maintenance costs and assume that the finger tables of the Chord nodes have perfectly accurate finger entries at all times regardless of node failures. The competing sequential lookup algorithm is thus a reasonably strong adversary and not just a straw man.

[3] The choice of which node to be responsible for a key is somewhat arbitrary. We could have decided to map a key to the node whose id most closely precedes the key, i.e. the *predecessor*, or the node that has the id closest to the key [4], [6], and our algorithm can still be applied with minor modifications.

- If it is a predecessor of id relative to x, it will provide information about its immediate successor and the l "best" next hops to the destination id from its cache[4].
- If it is a successor of id relative to x, it will provide information about its immediate predecessor and the l "best" next hops from its cache.

Here, l, like p, is a system parameter. We call an EpiChord network where there are at most p concurrent queries per lookup a p-*way* EpiChord.

When these replies are received, further queries will be dispatched asynchronously in parallel if x learns about nodes that are closer to the target id than the best successor and predecessor nodes that have already responded. An example of a lookup for the network shown in Figure 1 is given in Figure 3. In this example, $p = 3$, $l = 3$ and node $N32$ makes a lookup for the key $K2$. Note that when the lookup terminates, $N32$ would have learned about all the consecutive nodes in the range from $N60$ to $N9$. The simplified pseudocode for the lookup algorithm (which is implemented with callbacks and continuations) is given in Appendix A.

There are several reasons why queried nodes respond with information about their successors or predecessors. Firstly, this allows us to check for termination[5]. Secondly, since successors and predecessors are probed relatively more frequently than other cache entries, they are likely to be alive and hence with high probability, the querying node will make at least one step of progress towards the target id with each query. Lastly, even if nodes have an outdated view of the segment of the id space that they are responsible for, the querying node will be able to detect such a situation and resolve a lookup correctly. For example, an inconsistency can arise if the predecessor of a given node y is responsible for a queried id and it fails without informing y. Node y would not know that it is now responsible for id.

Our lookup algorithm is intrinsically iterative. The main reason for this is that an iterative approach allows us to avoid sending redundant queries. If we employ parallel queries in a recursive lookup, nodes at the subsequent hops would not know when other nodes respond to the original node that issued the lookup, and hence which new nodes *not* to query. In general, such an approach is likely to require $2p \times h$ messages (including both queries and responses) per lookup, where p is the number of parallel queries per hop and h is the number of hops. With an iterative approach, we usually require only about $2(p + h)$ messages per lookup.

B. *Reactive Cache Management*

Each cache entry has an associated time. When a node receives a query or reply, it adds an entry for the sender if it is not already in the cache and sets (or resets) the time of the entry associated with the sender to that of its local clock. Query responses contain a *lifetime* for each entry, equal to the sender's clock at the time of the send minus the node entry's time in the sender's cache, and this information is used to set or reset the time in the receiver's cache for that node. Node entries are flushed if their associated nodes do not respond to some number of queries or when their lifetime exceeds some limit, τ.

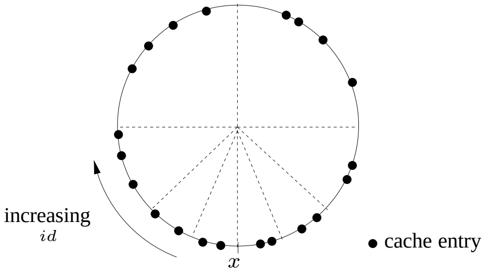

Fig. 4. Division of address space into exponentially smaller slices with respect to node x.

Like Chord, the correctness of the lookup algorithm is guaranteed because a query can always reach the destination id by moving sequentially down the successor lists. In general, $O(\log n)$-hop DHT routing schemes have a predefined set of $O(\log n)$ fingers and provide guarantees on lookup performance by ensuring that a node knows about some nodes in the vicinity of each finger. EpiChord divides the address space into two symmetric[6] sets of exponentially smaller slices as shown in Figure 4. For performance guarantees, a node enforces the following invariant:

Cache Invariant: *Every slice contains at least $\frac{j}{1-\hat{\gamma}}$ cache entries at all times.*

where $\hat{\gamma}$ is a local estimate of the probability that a cache entry is out-of-date (i.e., that the associated node had

[4] Correspondingly, the l "best" next hops are the node immediately succeeding id and to the $l - 1$ nodes preceding id.

[5] In general, we can terminate a `get()` lookup operation and return when the target id falls between a responding node and its successor or predecessor or whenever a node returns the requested object. However, if the node failure rate is high, we may choose to terminate a `put()` lookup operation only after both the best predecessor and best successor respond and we check that they are consistent (i.e., that think that they are adjacent to each other in the address space).

[6] In contrast to the asymmetric Chord finger table, the division of the address space into slices is symmetric by design. The key idea is that when node x responds to node y, they will each know that each other is alive, and if the node entry for y helps x to satisfies its cache invariant for a particular slice, we want the node entry for x to also be useful in satisfying the invariant for a corresponding slice in y's cache.

Node	**Initial cache contents**
$N32$	$\cdots, N55, N60, N0, N9, N17, \cdots$
$N60$	$\cdots, N0, N1, N3, N9, N15, \cdots$
$N0$	$\cdots, N60, N1, N3, N8, N15 \cdots$
$N1$	$\cdots, N0, N3, N8, N9, N15 \cdots$
$N3$	$\cdots, N0, N1, N8, N9, N17 \cdots$
$N8$	$\cdots, N0, N1, N3, N9, N15 \cdots$
$N9$	$\cdots, N55, N1, N3, N8, N15 \cdots$

Time	**Action by** $N32$	**Pending Queries**	**Best Predecessor**	**Best Successor**	**Comment**
$t = 0$	Send queries to $N60, N0, N9$	$N60, N0, N9$	$N32$	$N32$	send p queries
$t = 1$	Reply from $N60$ – $\{N0, N1, N3\}$	$N0, N9$	$N60$	$N32$	
$t = 2$	Send query to $N1$	$N0, N1, N9$	$N60$	$N32$	
$t = 3$	Reply from $N9$ – $\{N8, N55, N1, N3\}$	$N0, N1$	$N60$	$N9$	$N55$ ignored because $N60$ responded
$t = 4$	Send query to $N3$	$N0, N1, N3$	$N60$	$N9$	
$t = 5$	Reply from $N0$ – $\{N60, N1, N3\}$	$N1, N3$	$N0$	$N9$	
$t = 6$	Send query to $N8$	$N1, N3, N8$	$N60$	$N9$	
$t = 7$	Reply from $N8$ – $\{N0, N1, N3\}$	$N1, N3$	$N0$	$N8$	
$t = 8$	Reply from $N3$ – found key $K2$!	$N1$	$N0$	$N3$	lookup returns
$t = 9$	Reply from $N1$ – $\{N0, N1, N3\}$	-	$N1$	$N3$	lookup terminates

Fig. 3. Example of a lookup for the network shown in Figure 1. In this example, $p = 3$, $l = 2$ and node $N32$ makes a lookup for the key $K2$.

failed). A node checks its cache slices periodically and ensures that there are sufficient unexpired cache entries in each slice. Should a slice be found not to have sufficient unexpired cache entries, a node makes a lookup to the midpoint of that slice. Since j is small (e.g. 2), one lookup is usually all it takes to satisfy the cache invariant.

The key idea is that to provide an $O(\log n)$-hop guarantee on the lookup path length, the density of entries per slice must increase exponentially as we get nearer to the node's id. EpiChord estimates the number of slices from its k successors and k predecessors: it requires that the successor and predecessor lists fall into the two adjacent slices closest to the reference node. This implies that we need to choose j and k such that $k \geq 2j$.

To estimate γ, the probability that a given cache entry is stale, each node tracks two variables:

- n_p, the number of nodes probed
- n_t, the number of probed nodes that timed out

We estimate γ with:

$$\hat{\gamma} = \frac{n_t}{n_p} \tag{1}$$

In addition, we multiply n_p and n_t by δ_γ periodically (i.e., when the cache is flushed) to obtain exponentially weighted moving averages for both estimates. We weight the raw values instead of periodically computed ratios because huge errors can be introduced in the estimates when the frequency of computation is high and insufficient samples are accumulated between computations. In our implementation, we set $\delta_\gamma = 0.5$ and we observe experimentally that we can obtain relatively good estimates (to within 25% of the true value) in the steady state with our experimental parameters.

C. *Stabilization*

When multiple nodes attempt to join the Chord ring at approximately the same location, temporary inconsistencies may arise in the address space. Also, as nodes fail and leave the network unannounced, segments of the address space may become orphaned (i.e., none of the nodes know that they are responsible for them). We run a weak stabilization protocol periodically to fix local inconsistencies in the address space and a strong stabilization protocol to detect and fix global inconsistencies.

> **Definition 1:** We say that the network is (i) *weakly stable* if, for all nodes u, we have $predecessor(successor(u)) = u$; (ii) *strongly stable* if, in addition, for each node u, there is no node v such that $u < v < successor(u)$; and (iii) loopy if it is weakly but not strongly stable (see [20]).

1) Weak Stabilization Protocol: All messages contain the IP address, port number and id of the sender. So unlike Chord, there is no longer a need for a node to explicitly notify its successor that it is the new predecessor after it joins the network. When it contacts the successor to initiate a cache transfer, the successor would realize that the new node has joined the network and update its predecessor pointer accordingly.

In addition, nodes periodically probe their immediate neighbors to check if they are still alive. When probed, a node will either (i) send a short reply message with its current predecessor and successor or (ii) send a complete list of its immediate neighborhood (k predecessors and k successors) if a change was detected within k hops of the probing node.

Each node is responsible for finding and maintaining its own successor and predecessor. When a node hears from another node whose id is closer than its current predecessor and successor, the new node is automatically set as the predecessor or successor accordingly. If a node learns about a node that could possibly be its new predecessor or successor indirectly from another node (or by observing lookup traffic), the node will probe this new node and set it as the predecessor or successor only if it receives a positive response on the probe. Periodically, each node will probe its perceived successor and predecessor (which may not be correct) to learn about the nodes' neighborhoods. In this way, a node is eventually guaranteed to discover a better predecessor or successor in the vicinity of its id, if one exists.

> **Theorem 1:** The weak stabilization protocol will eventually cause an EpiChord network to converge to a *weakly stable* state.

To prove this theorem, we observe that each node has only a finite number of possibilities (exactly $n - 1$) for its predecessor and successor. For a node u such that $predecessor(successor(u)) \neq u$, u would eventually probe its successor and both would update their predecessor and successor pointers accordingly. Each predecessor/successor update event monotonically improves the consistency of the address space, i.e., a node only adopts a new predecessor or successor if it is strictly better than its previous successor. Therefore, the address space pointers will eventually converge to a *weakly stable* state, which is the only state where updates will no longer happen.

2) Strong Stabilization Protocol: Although, it is in generally highly improbable that a network will end up loopy (except perhaps after a network partition), for completeness, it is still desirable to have a scheme that will detect and fix global inconsistencies in the address space. Our strong stabilization algorithm is based on a very simple idea: to detect loops, all we need to do is to traverse the entire ring and make sure that we come back to where we started. Figure 5 shows graphical example of a loopy, but locally consistent address space. In this example, node n forwards a packet containing its identifier along the ring. When the packet reaches node m, m realizes that n exists and initiates the weak stabilization protocol with n to repair the address space. A naive scheme to pass a single token along the ring will take a long time and is relatively inefficient, so instead, we implement a parallelized token-passing scheme.

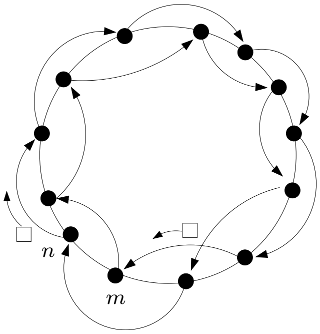

Fig. 5. An example of a loopy address space configuration. The arrows indicate the direction of the successor pointers.

As loopy configurations are expected to be rare, strong stabilization needs to be performed only infrequently. The key idea in our strong stabilization protocol is to generate and pass q tokens (which are simply messages) along the ring using only the successor pointers. In our protocol, immediately after a node sees a stabilization token (or immediately after it joins the network), it will pick a random waiting period from the interval (t_{min}, t_{max}) after which it will initiate strong stabilization. If a node sees a token before its timer runs out, it will reset its timer and choose again. In this way, we can control the number of concurrent tokens that are passed in the ring at any given instant in time in a distributed fashion.

To initiate the strong stabilization process:

- a node x (with identifier n_x) picks q nodes with identifiers $n_1, n_2, \cdots, n_q$, distributed approximately uniformly in the address space, from its cache, where q is the degree of parallelization and $n_x < n_1 < n_2 < \cdots < n_q$.
- x sends node n_q a token with n_x (itself) marked as the destination.
- x then proceeds to send node n_i a token with n_{i+1} marked as the destination, for $i = q - 1, \cdots, 1$ in order. If a given node n_j is found to have failed, another node in its vicinity is chosen instead.
- finally, x generates a token with destination n_1 and passes it to its successor.

This is illustrated in Figure 6.

When a node receives a token, it passes the token to its successor. A token is destroyed when it reaches a node with an identifier greater or equal to its intended destination (modulo the circular address space). When a token is destroyed, one of two possibilities can occur:

1) the segment of the address space traversed by the token is not loopy, in which case, the token either

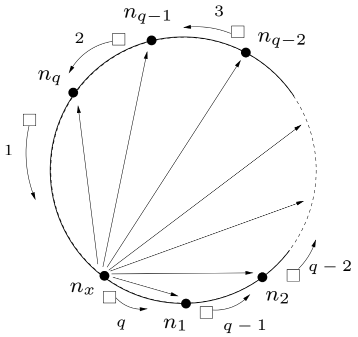

Fig. 6. Example on the generation of q stabilization tokens.

ends up at its intended destination or its successor (if the destination node failed at the meantime) and nothing happens. All the nodes in the path of the token would however have learned about the destination node.

2) the segment of the address space traversed by the token is loopy and the token does not end up at the intended destination. Again, however, the nodes in the path of the token would have learned about the destination node and as a result, two of the nodes on the alternate segment in the vicinity of the destination node would start to probe for the destination node because of weak stabilization and the loop will eventually be eliminated.

If the network is large, $\frac{n}{q}$ is large and it will still take a long time (and many hops per token) to complete one round of token-passing. To avoid this problem, nodes generate secondary tokens. For example, node y with identifier n_y receives a token destined for n_z. Instead of just passing the token to its successor, a node can also choose q nodes with identifiers $n_1, n_2, \cdots, n_q$ such that $n_y < n_1 < n_2 < \cdots < n_q < n_z$ and generate the corresponding q tokens. With this recursive process, each token-passing round can be completed in $O(\log n)$ time.

Theorem 2: The combination of our recursive parallel token-passing algorithm with the weak stabilization protocol will cause an EpiChord network to converge to a *strongly stable* state after at most $O(n^2)$ rounds of token-passing.

There are two key intuitions behind the correctness of this theorem. First, if the network is loopy, the token-passing algorithm will cause at least one pair of nodes to detect an inconsistency. Next, whenever such an inconsistency is detected, the pair of nodes that detect the inconsistency will update each other and strictly improve the state of the network. Since each node in the network has one correct successor and the only stable state is when the network is no longer loopy, we conclude that the network must eventually become strongly stable. The bound is obtained from observing that each node has only n possible choices for its successor. Since each round of token-passing updates at least one node, we know that it will take at most $O(n^2)$ rounds to update the successor (and predecessor) pointers to the correct values.

To see that the token-passing algorithm will allow at least one pair of nodes to detect an inconsistency, consider a network that is weakly stable, i.e., if we followed the successor pointers we would eventually end up where we started. Suppose we choose r nodes arbitrarily such $n_1 < n_2 < \cdots < n_r$. Take a node, say n_x and follow the successor pointers. Repeat this process for all nodes. If the network is not loopy, it is clear that the node ids would increase monotonically (modulo the address space) until we reach node n_{x+1}; If the network is loopy, for at least one node n_y, we would eventually reach a node n_z such that $n_y < n_{y+1} < n_z$ (modulo the address space). The key is to recognize that the net effect of our secondary token generation mechanism is to choose these r nodes recursively.

Intuitively, it is quite easy to see that if we choose a set of r nodes in the ring and have them forward messages to adjacent nodes in this set along the ring, we can detect inconsistencies. What is interesting about our algorithm is that we have demonstrated that we can choose this set of r nodes recursively in a distributed way and still preserve the correctness of this approach.

III. Analysis

A. Worst-Case Lookup Performance

If we assume a uniformly distributed workload, we can show that the worst-case lookup performance is $O(\log n)$ hops. In addition, the expected worst-case lookup path length is at most $\frac{1}{2}\log_\alpha n$, where $\alpha = 3j + \frac{6}{j+3}$. Here, n is the size of the network, and j is the minimum number of cache entries per slice (see Appendix B). When $j = 1$, we get the same expected worst-case result as Chord does. However, for $j \geq 2$, we tend to do much better: for $j = 2$, $\alpha = 7.2$ and the EpiChord expected lookup path lengths are at most only $\frac{\frac{1}{2}\log_2 n}{\frac{1}{2}\log_\alpha n} = \log_\alpha 2 \approx \frac{1}{3}$ of that for Chord[7]. Our analysis implicitly assumes that the queries in each hop are synchronized. Because our lookup algorithm is asynchronous, actual lookup path lengths will tend to be slightly larger.

B. Reduction in Background Probes

EpiChord exploits information gleaned from observing lookup traffic to improve lookup performance, and only

[7] The expected lookup path length for Chord is $\frac{1}{2}\log_2 n$ [20].

sends network probes when necessary. To see the bandwidth savings with our approach, we consider a network with a steady state size of 20,000 nodes and nodes that have an median lifespan of 60 minutes[8]. This translates to a node failure rate of approximately 0.03% (or 5 nodes) per second. Assuming that the application-level lookup traffic received by a node is approximately uniformly distributed (this is a reasonable assumption since node ids are obtained using the SHA-1 hash [19] and are thus uniformly distributed), the proportion of lookup traffic that will help to satisfy the cache invariants for various values of lookup traffic and j is shown in Figure 7. With an amount of lookup traffic approximately equal to the required background maintenance traffic (i.e., $x = 1$ in Figure 7), we can achieve a 35% reduction in the background maintenance traffic. At larger network sizes, the savings in background maintenance traffic is reduced. However, as shown in Figure 8, even at network sizes of 1,000,000 nodes, we can still expect a reduction of more than 25% on average.

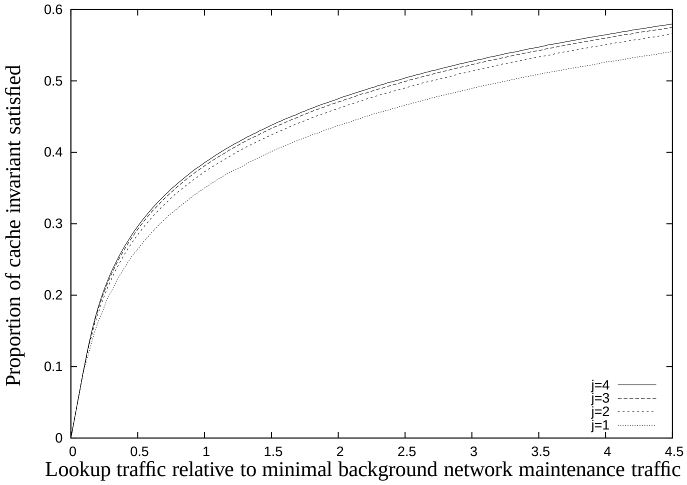

Fig. 7. Effect of j on the proportion of lookup traffic that helps to satisfy cache invariant for 20,000-node network.

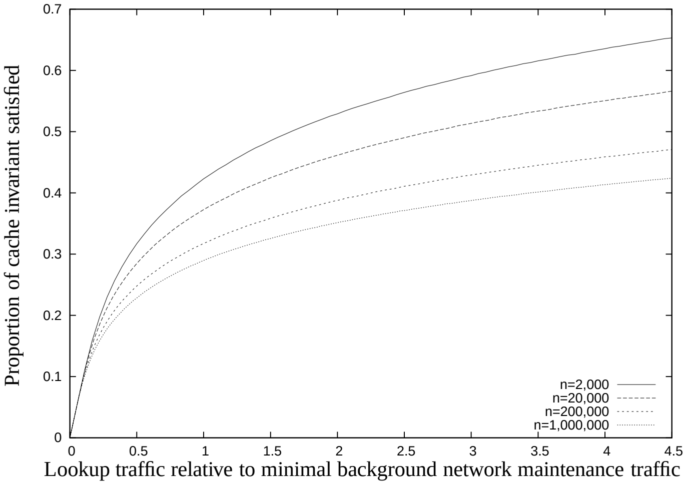

Fig. 8. Effect of network size (n) on the proportion of lookup traffic that helps to satisfy cache invariant (for $j = 2$).

C. Cache Composition in the Steady State

The proportion of live entries[9] in the cache is an important system parameter because it determines the probability of a timeout occurring during a lookup. To obtain an estimate of the number of live entries in a cache in the steady state, we consider a network of size n such that in a fixed time interval, a fraction r of the nodes in the network leave, a fraction f of the cache entries are flushed and each node makes Q lookups uniformly over the id address space and sends out p queries in parallel for each lookup. Where x is the number of live nodes that is known to a node at time t, we obtain the following relation:

$$\frac{d}{dt}x(t) = \overbrace{pQ(1-\frac{x}{n})}^{\text{incoming queries}} - \overbrace{fx}^{\substack{\text{entries}\\ \text{flushed}}} - \overbrace{(1-f)rx}^{\substack{\text{nodes departed but}\\ \text{not flushed}}} \tag{2}$$

We have assumed that new knowledge comes only from the incoming queries as a node would have to know about a node in order to send an outgoing query to it. This is conservative and will tend to under-estimate the increase in x. We have also assumed that the probability that a cache entry is flushed is independent of the probability of failure for the associated node. The steady state solution to x is:

$$\lim_{t\to\infty} x(t) = \frac{pQ}{pQ + (f + r - rf)n} n \tag{3}$$

In addition, where y is the number of stale cache entries at time t, we have the following relation:

$$\frac{d}{dt}y(t) = \overbrace{(1-f)rx}^{\substack{\text{stale entries}\\ \text{not flushed}}} - \overbrace{fy}^{\substack{\text{stale entries}\\ \text{flushed}}} - \overbrace{pQ(\frac{y}{x+y})}^{\substack{\text{stale entries discovered by}\\ \text{timeouts of outgoing queries}}} \tag{4}$$

In a network with high churn, the proportion of stale entries in the cache, γ, is a key system parameter:

$$\gamma = \lim_{t\to\infty} \frac{y}{x+y} = \frac{1}{pQ}\left[(1-f)rx - fy\right] \tag{5}$$

[8] These figures are representative of both the Napster and Gnutella peer-to-peer file-sharing networks as reported in a measurement study by Saroiu et al. [21].

[9] An entry is *live* if its associated node is still online. The set of cache entries for a node will in general consist of some live entries and some unexpired, stale entries.

If $pQ \gg rn$ and $f = 0$, then $x \approx n$ and $\gamma \approx \frac{rn}{pQ} \approx 0$. This implies that if the level of lookup traffic is high enough, the performance of the system is somewhat independent of the cache maintenance protocol. This agrees with our intuition, since with a high level of lookup traffic, most nodes will know about a large number of other nodes and many stale cache entries would be discovered and eliminated during the lookup process.

Next, we consider the case when $pQ \ll rn$. In the steady state,

$$\lim_{t\to\infty} x(t) \approx \frac{pQ}{f+r-rf} \tag{6}$$

By setting $\frac{dy}{dt} = 0$ in (4), we obtain:

$$fy^2 + [pQ + (f - r + rf)x]y - (1-f)rx^2 = 0 \tag{7}$$

Substituting (6) yields:

$$fy^2 + \left[\frac{2fpQ}{f+r-rf}\right]y - \frac{(1-f)r}{(f+r-rf)^2}(pQ)^2 = 0 \tag{8}$$

$$\Rightarrow y = \left(\sqrt{1 + \frac{(1-f)r}{f}} - 1\right)\frac{pQ}{f+r-rf} \tag{9}$$

$$\Rightarrow \gamma = \lim_{t\to\infty} \frac{y}{x+y} = \frac{\sqrt{1+\frac{(1-f)r}{f}} - 1}{\sqrt{1+\frac{(1-f)r}{f}}} \tag{10}$$

If cache entries are flushed at a rate that is at least as fast as the node failure rate, i.e. $f \approx r$, then

$$\gamma = \frac{\sqrt{2-f} - 1}{\sqrt{2-f}} \le 1 - \frac{1}{\sqrt{2}} = 0.292 \tag{11}$$

Thus, our model predicts that even when the churn rate is high ($pQ \ll rn$), at most 30% of the cache entries will be stale (and this result is independent of the level of lookup traffic pQ). This result was verified by our simulations. To get a simpler close form for the expected proportion of stale entries, let $f = cr$, i.e., we flush entries at a rate that is c times faster than the node failure rate. Next, assume that f is small and $\frac{r}{f} < 1$, then

$$\gamma = \frac{\sqrt{1+\frac{(1-f)r}{f}} - 1}{\sqrt{1+\frac{(1-f)r}{f}}} \tag{12}$$

$$\approx \frac{1 + \frac{1}{2}\cdot\frac{r}{f} - 1}{1 + \frac{1}{2}\cdot\frac{r}{f}} \tag{13}$$

$$= \frac{1}{2c+1} \tag{14}$$

IV. Simulation Results

To understand the trade-offs when we move from an $O(\log n)$-state-per-node DHT to an unlimited-state-per-node DHT with the same basic routing topology, we compare EpiChord to a corresponding optimal iterative Chord network of the same size using our simulation built on the *ssfnet* [22] simulation framework. We run the simulations on a 10,450-node network topology organized as 25 autonomous systems, each with 13 routers and 405 end-hosts. The simulated network topology is represented graphically in Figure 9.

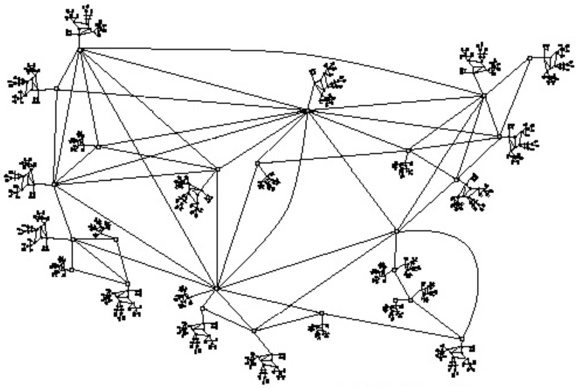

Fig. 9. Simulation Network Topology.

The average roundtrip time (RTT) between nodes in the topology is approximately 0.16 s. Hence, we set timeouts at 0.5 s for all simulations. The cumulative distribution of the RTTs in the simulation topology between any two pairs of nodes is shown in Figure 10. Since all query packets are UDP-based and packets may be lost, we retransmit twice after a timeout and will decide that a node has failed if we do not hear from it after 3 tries.

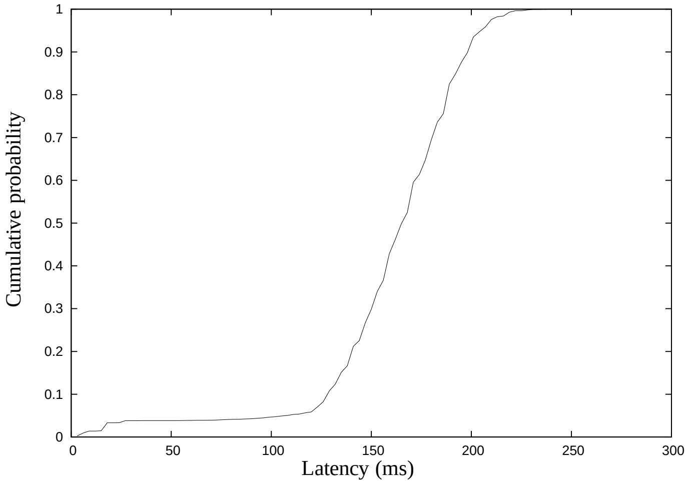

Fig. 10. Cumulative distribution of RTTs in simulation topology.

Li et al. highlighted that the assumed workload will affect the result of comparisons between DHTs significantly [23]. They proposed two generic classes of workloads – *lookup-intensive* and *churn-intensive*. Although they did not propose exact definitions for these two classes of workloads, we do have a very natural way of defining these two classes of workloads for EpiChord based on our steady-state cache model. In particular, we consider a workload to be lookup-intensive if $pQ \gg rn$, and churn-intensive if $pQ \ll rn$.

In our simulations, we first generate a sequence of node joins/departures and queries according to a predetermined set of network parameters. Subsequently, we run the same set of traces on the EpiChord networks of varying degrees of parallelism and on a corresponding Chord network. This ensures that the results can be compared fairly across the two algorithms without bias in the choice of node ids and lookup ids.

A. Lookup-Intensive Workload ($pQ \gg rn$)

In our lookup-intensive workload simulation, node lifespans are exponentially distributed with a mean of 600 s. We experiment with a range of network sizes by varying the rate of node joins from 0.33 to 2 nodes per second. Each node in the network makes on average 2 lookups per second. In steady state, the network sizes range from 200 to 1,200 nodes and the overall system query rate ranges from 400 to 2,400 lookups per second. The stabilization interval is 60 s (i.e., nodes probe their successors and predecessors once a minute) and the lifetime of a cache entry is 120 s. Since the expected background maintenance traffic is negligible compared to the active lookup rate, $Q \approx 2$ and rn ranges from 0.33 to 2. Also, $r \approx \frac{1}{600}$, $f \approx \frac{1}{120}$ $(f > r)$ and $j = 2$.

Since we are comparing EpiChord to Chord, we had to pick an appropriate cache entry TTL to ensure that the maximal background maintenance traffic generated by EpiChord does not exceed that for a corresponding Chord network. If we assume that nodes have exponentially distributed (memoryless) lifetimes with mean T, where X is the random variable representing the time of failure, $P(X \leq t)$ is given by:

$$P(X \leq t) = e^{-\frac{t}{T}} \qquad (15)$$

Hence, if nodes have mean lifetimes of 600 s, cache entries would have to be probed at least once every 60 s to ensure that they have a 90% probability of being valid. It is thus reasonable to assume a periodic probe rate of at least 60 s for a Chord network. The minimal routing set for an EpiChord network of comparable size with $j = 2$ would have slightly less than 4 times as many entries. However, since the cache slices for EpiChord is symmetric, we need only half the number of probes required by Chord and so with a mean cache entry TTL of 120 s $> 2 \times 60$ s, the maximum background maintenance traffic for the EpiChord networks (even in the absence of lookups) in our simulations are guaranteed not to exceed that for the corresponding Chord network.

1) Lookup Performance: The average latency and the average hop count per lookup for successful lookups in the steady state are shown in Figures 11 and 12 respectively. From Figure 11, we see that having more parallelism reduces the lookup latency. In Figure 12, the hop count for EpiChord is defined as the minimum number of nodes that have to be contacted in the final (successful) lookup sequence. We see that the average steady-state hop count varies from 1.1 to 1.4. This means that at least 60% of the lookups succeed within the initial wave of lookup queries. This result is actually not surprising since we know from our analysis that the expected worst-case hop count is $\frac{1}{2}\log_{\alpha} n = \frac{1}{2}\log_{7.2} 1{,}200 = 1.80$.

We consider a lookup to be successful if it locates the correct node within 5 minutes. This means that it is occasionally possible for a lookup to time out while waiting for the response from some failed node, and then subsequently proceed to continue the lookup process with another node and succeed. The distribution of latencies is thus strongly bi-modal, with the majority of lookups succeeding relatively quickly while a small fraction succeeding only after a timeout. These timeouts explain why the average latency as reported in Figure 11 for the 1-way EpiChord network is about twice the average RTT instead of being approximately equal to the RTT even though the hop count is 1.4. The timeout probabilities are shown in Figure 14. The latencies for successful lookups that experience timeouts is generally more than 10 times of that for lookups that do not time out, though fortunately, the former occur much more infrequently than the latter.

As shown in Figure 13, the lookup failure rates are relatively low ($< 0.1\%$). This is not surprising since under the lookup-intensive workload, the large number of lookups keep the routing state for most nodes mostly up-to-date. Adding more parallelism (increasing p) reduces the probability of lookup failure significantly[10]. The lookup failure probability falls by approximately an order of magnitude when p is increased by one.

2) Message Count: It is clear that a parallel lookup algorithm will generate more lookup messages when there are more parallel queries per lookup. Figure 15 shows that for our given parameter settings, the average number of query and reply messages that are required for a se-

[10] The competing optimal Chord network has perfectly accurate fingers at all times and thus lookups never fail.

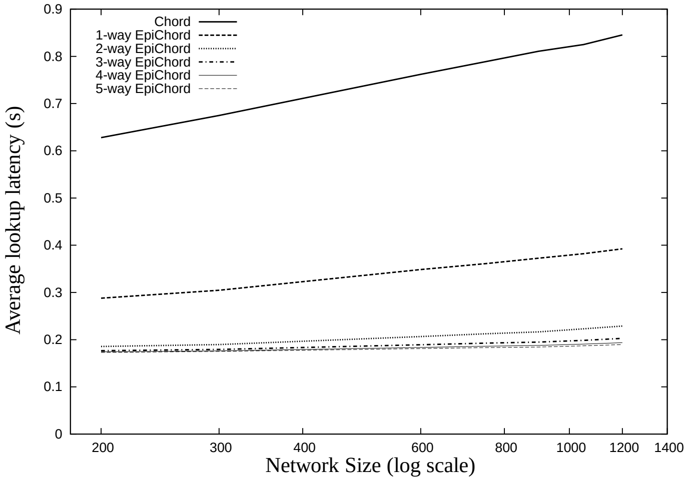

Fig. 11. Comparison of lookup latency between Chord and p-way EpiChord under lookup-intensive workload.

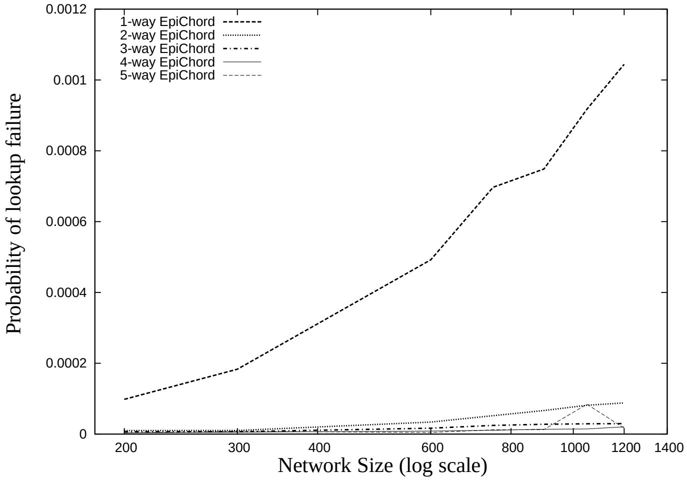

Fig. 13. Lookup failure rates for p-way EpiChord networks under lookup-intensive workload.

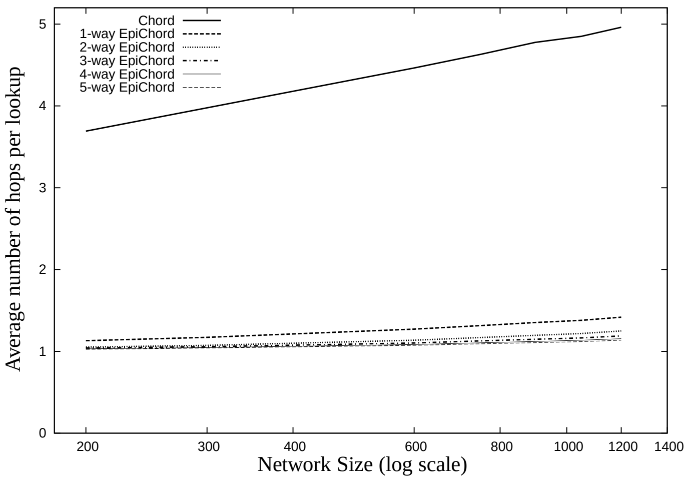

Fig. 12. Comparison of lookup path length between Chord and p-way EpiChord under lookup-intensive workload.

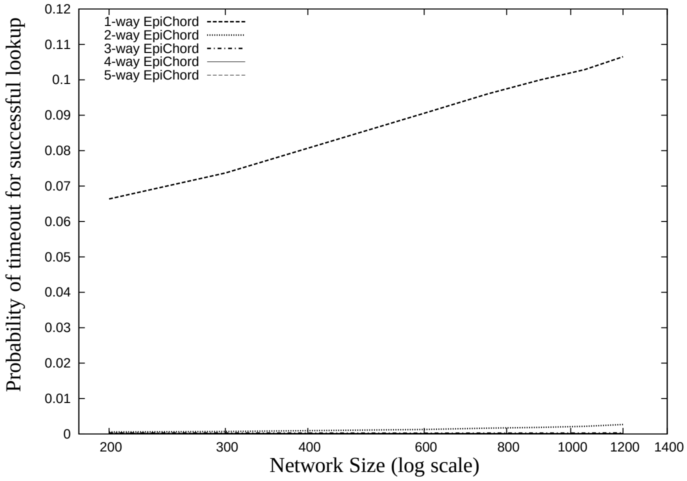

Fig. 14. Probability of timeouts for p-way EpiChord networks under lookup-intensive workload.

quential Chord network is approximately equal to that for a 3-way EpiChord network. As mentioned in Section II-A, the main reason why the number of lookup messages does not increase in proportion with p is that with iterative lookups, the querying node can avoid sending duplicate and redundant queries.

B. Churn-Intensive Workload ($pQ \ll rn$)

In our churn-intensive workload simulation, node lifespans are exponentially distributed with a mean of 600 s. The stabilization interval is 60 s and the lifetime of a cache entry is 120 s. We experiment with a range of network sizes by varying the rate of node joins from 1 to 15 nodes per second. Each node in the network makes on average 0.01 lookups per second. Because the lookup rate is so low, most of the lookups captured in our results are lookups arising from node joins and cache maintenance. In steady state, the network sizes range from 600 to 9,000 nodes and the overall system query rate ranges from 40 to 700 lookups per second. Including the minimal expected background maintenance traffic, $Q \approx 0.05$ to 0.08 and rn ranges from 1 to 15. As before, $r \approx \frac{1}{600}$, $f \approx \frac{1}{120}$ ($f > r$) and $j = 2$.

1) Lookup Performance: The average latency and the average hop count per lookup for all successful lookups are shown in Figures 16 and 17 respectively. Again, we see from Figure 17 that adding more parallelism reduces the lookup latency significantly. As shown in Figure 18, the lookup failure probabilities under the churn-intensive workload are higher than those under the lookup-intensive workload (which are $\leq 0.1\%$). This is to be expected since the churn rate is higher and the information propagation rate is significantly lower. From Figure 18, we see that the failure rates for the 4- and 5-way EpiChord

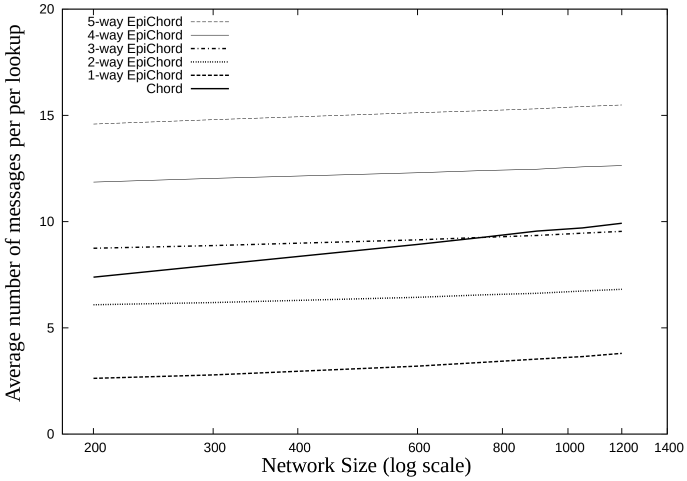

Fig. 15. Comparison of lookup message count between Chord and p-way EpiChord under lookup-intensive workload.

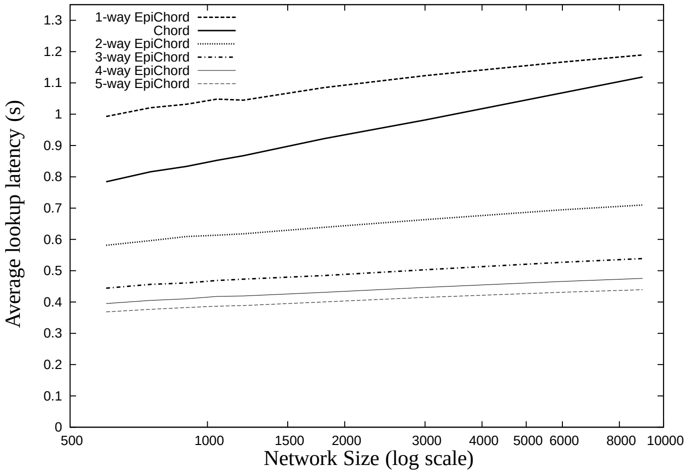

Fig. 16. Comparison of lookup latency between Chord and p-way EpiChord under churn-intensive workload.

networks are higher than that for the 3-way EpiChord network, which is somewhat counter-intuitive. We discovered that the cause of this phenomenon is that with a larger p, each lookup invoked for cache maintenance satisfies the cache invariant for more nodes and so the 4- and 5-way EpiChord networks generate fewer cache-refreshing lookups than a 3-way EpiChord network. This lower rate of background maintenance traffic accounts for the marginally higher failure rates for larger network sizes. As shown in Figure 19, with $p \geq 2$, successful lookups will almost never experience timeouts.

2) Message Count: As shown in Figure 20, more messages are required to complete a lookup under a churn-intensive workload. However, the increase in message count over the lookup-intensive workload is quite modest: a 1-way EpiChord network requires approximately the same number of messages per lookup as the corresponding Chord network, while a 3-way EpiChord network incurs approximately 50% more lookup traffic.

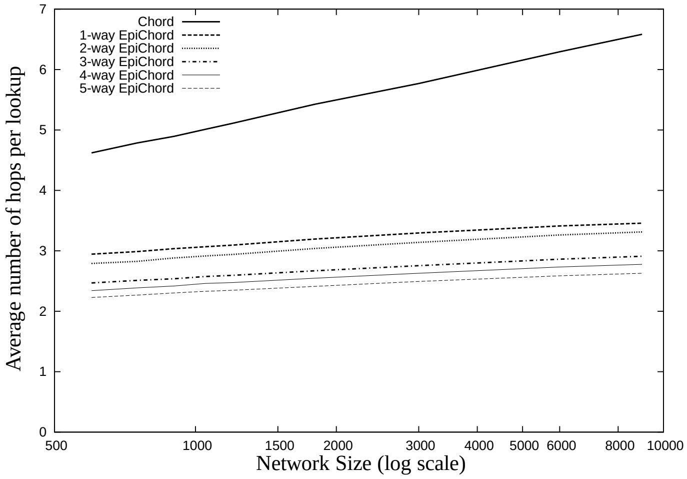

Fig. 17. Comparison of lookup path length between Chord and p-way EpiChord under churn-intensive workload.

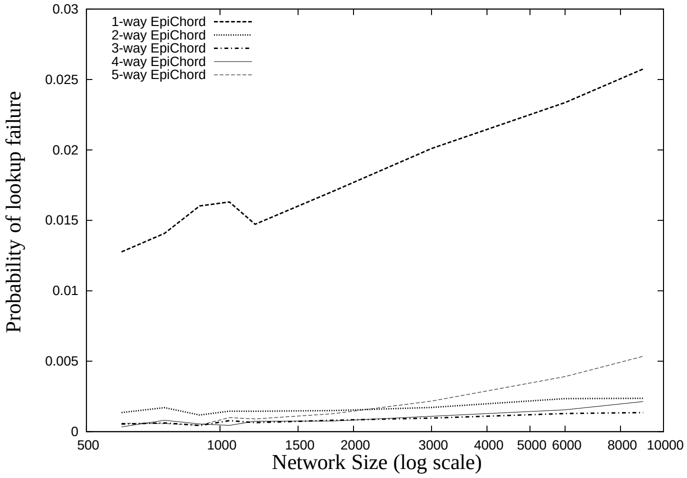

Fig. 18. Lookup failure rates for p-way EpiChord networks under churn-intensive workload.

C. Cache Composition

Figures 21 and 23 show the average number of live and stale entries in the caches of the nodes for the EpiChord networks under a lookup-intensive workload and a churn-intensive workload respectively, while Figures 22 and 24 show the fraction of stale entries in the respective caches. These results seem to support the conclusion from our analysis in Section III-C that the fraction of stale entries depends only on the node failure rate and the frequency at which entries are flushed from the cache.

According to our cache model for churn-intensive workloads,

$$\gamma \;\approx\; \frac{1}{2c+1}, \quad c = \frac{f}{r} \approx \frac{\frac{1}{120}}{\frac{1}{600}} = 5$$

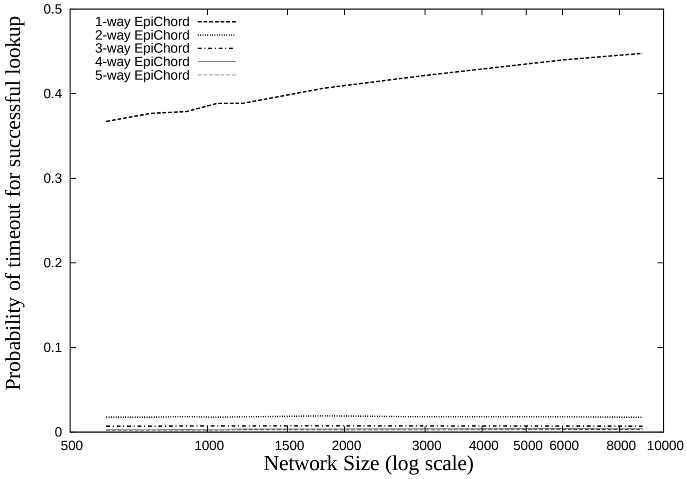

Fig. 19. Probability of timeouts for p-way EpiChord networks under churn-intensive workload.

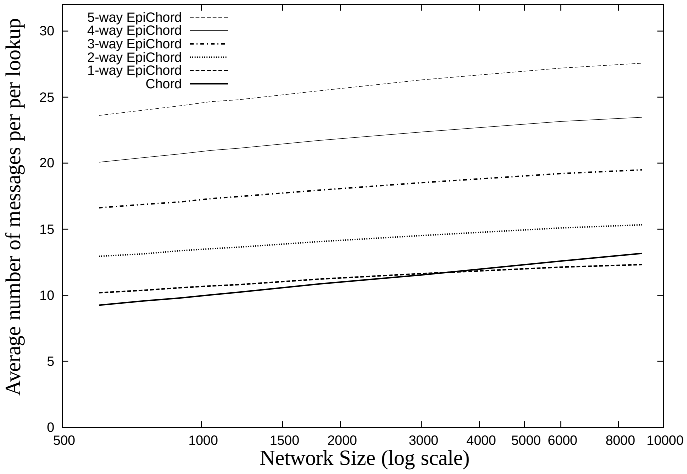

Fig. 20. Comparison of lookup message count between Chord and p-way EpiChord under churn-intensive workload.

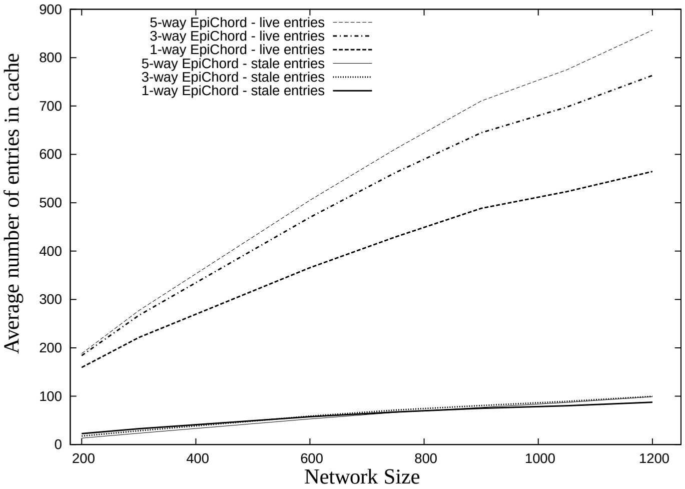

Fig. 21. Cache composition for p-way EpiChord networks under lookup-intensive workload.

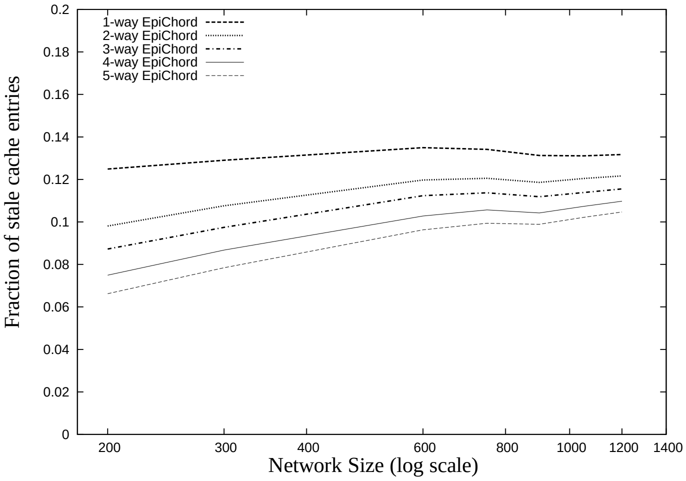

Fig. 22. Fraction of stale entries for p-way EpiChord networks under lookup-intensive workload.

$$= \quad \frac{1}{2 \times 5 + 1} = 9\%$$

This means that the predicted fraction of stale cache entries in the steady state is approximately 9% for the expected node lifespans and cache flush rates in our experiments. From Figures 23 and 24, we see that our estimate of 9% is somewhat smaller than the actual value ($\approx 12.5\%$). This is likely due to the neglected terms in our approximation and also to the fact that f is smaller than $\frac{1}{120}$ in practice, i.e., although cache entries are flushed every 120 s, the probably of a cache entry being flushed out every second is smaller than $\frac{1}{120}$.

D. Effect of Lookup Traffic

To investigate the effect of lookup traffic on lookup performance, we hold p and l constant at 3 and vary the amount of lookup traffic per node Q between 0.01 and 2.0 per second for a range of networks with sizes from 600 to 1,200 nodes. As shown in Figures 25, 26 and 27, increasing the amount of lookup traffic reduces the lookup path length, lookup latency, and the number of messages sent per lookup. There are however decreasing marginal returns with increasing traffic and the EpiChord lookup algorithm achieves close to optimal performance with a reasonably small amount of lookup traffic (i.e., $Q = 0.5$).

E. Effect of Number of Entries Returned Per Query

To investigate the effect of the number of "best entries" returned per response, l, on lookup performance, we hold p constant at 3 and the amount of lookup traffic Q constant at 0.01 per node per second (to minimize the *lookup-traffic effect*) and vary l between 2 and 4 for a set of network with sizes from 600 to 1,200 nodes. As demonstrated by our results shown in Figures 28, 29 and 30, l has a negligible

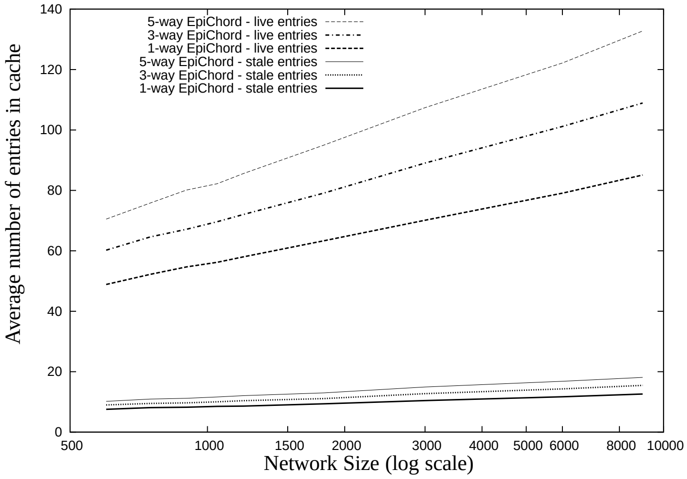

Fig. 23. Cache composition for p-way EpiChord networks under churn-intensive workload.

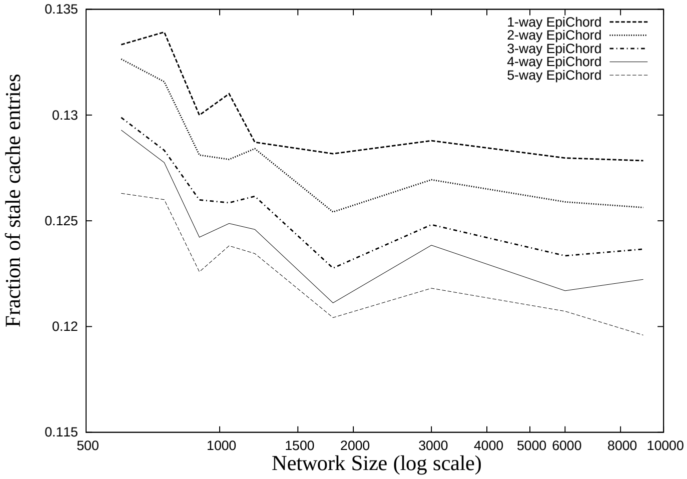

Fig. 24. Fraction of stale cache entries for p-way EpiChord networks under churn-intensive workload.

effect on the lookup path length, lookup latency and the number of messages sent per lookup. We thus conclude that we can keep l small and set $l = 3$.

V. Discussion

Our analysis and simulations have shown that by using parallel lookups and by amortizing the network maintenance costs into the lookup costs, our approach offers significantly better lookup path lengths and latencies with little additional costs in terms of bandwidth consumption. Our simulations have also shown that even though multiple messages are sent per lookup step, the lookup traffic generated is not significantly larger than that for a sequential lookup algorithm because the lookup path lengths are significantly shorter. In fact, the lookup traffic generated by a 3-way EpiChord network is comparable to that for a corresponding Chord network. This is a desirable

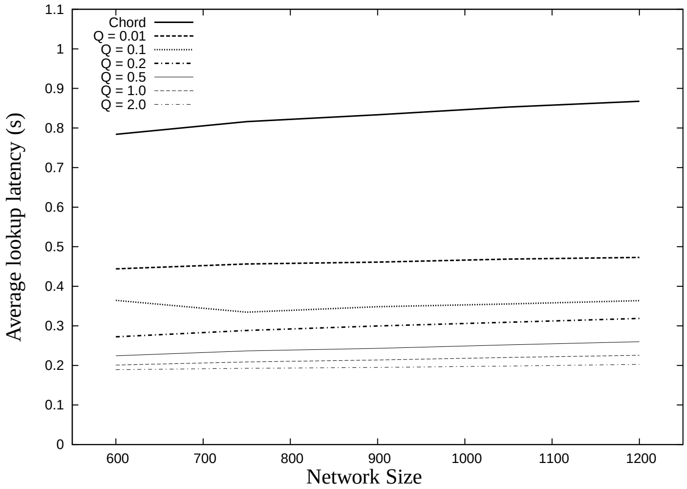

Fig. 25. Comparison of lookup latency between Chord and 3-way EpiChord under varying amounts of traffic.

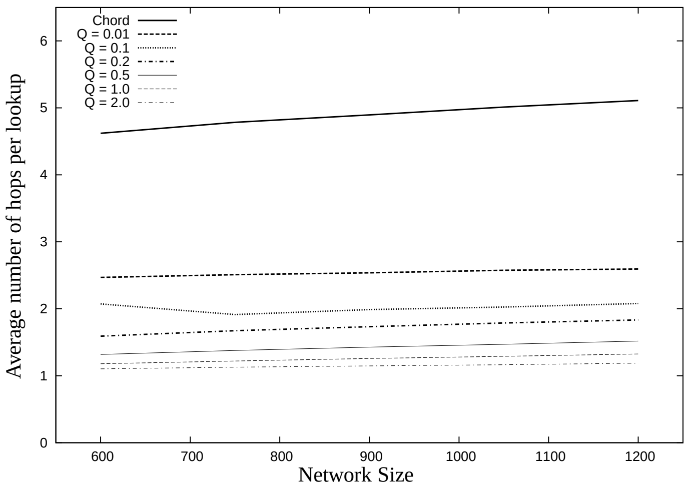

Fig. 26. Comparison of lookup path length between Chord and 3-way EpiChord under varying amounts of traffic.

trade-off because lookup latency is the principal measure of lookup performance.

Our new algorithm yields substantial savings in terms of setup time and the number of messages sent when a node first joins the network, compared to Chord and many other DHTs. To join the network, a node need only perform one lookup, contact its successor and predecessor, and perform an initial cache transfer[11]. Although performance is better with a full initial cache transfer, a minimal transfer of $O(\log n)$ entries is sufficient to guarantee worst-case $O(\log n)$-hop lookup performance. In contrast, $O(\log n)$ lookups ($O(\log^2 n)$ messages) are required

[11] Adjacent nodes in an EpiChord network usually have a similar set of address space slices for their cache invariants. This means that after a node completes a cache transfer from either its successor or predecessor, it will generally have a cache that already satisfies the invariant.

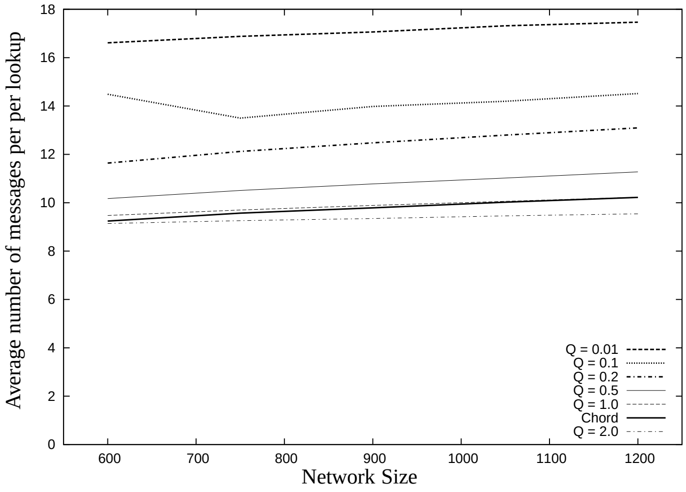

Fig. 27. Comparison of lookup message count between Chord and 3-way EpiChord under varying amounts of traffic.

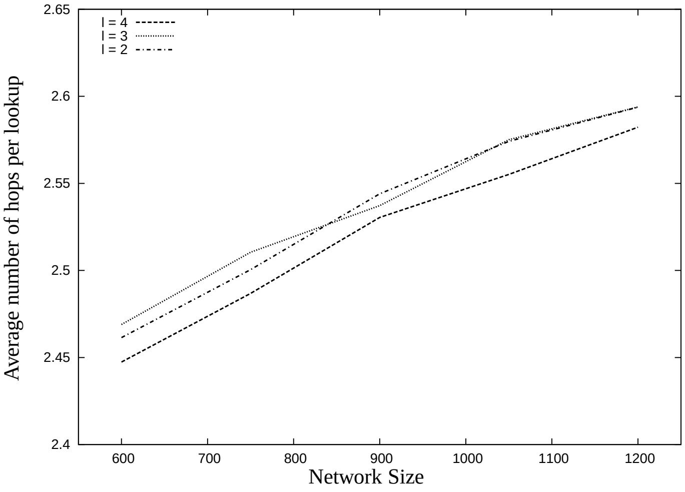

Fig. 29. Effect of l on lookup path length for a 3-way EpiChord network.

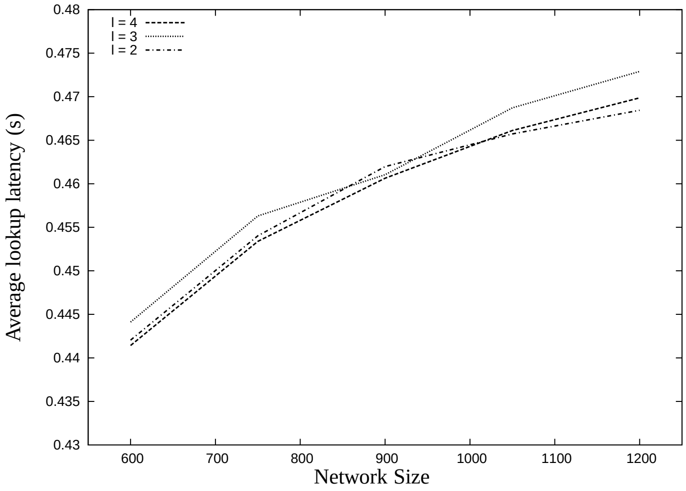

Fig. 28. Effect of l on lookup latency for a 3-way EpiChord network.

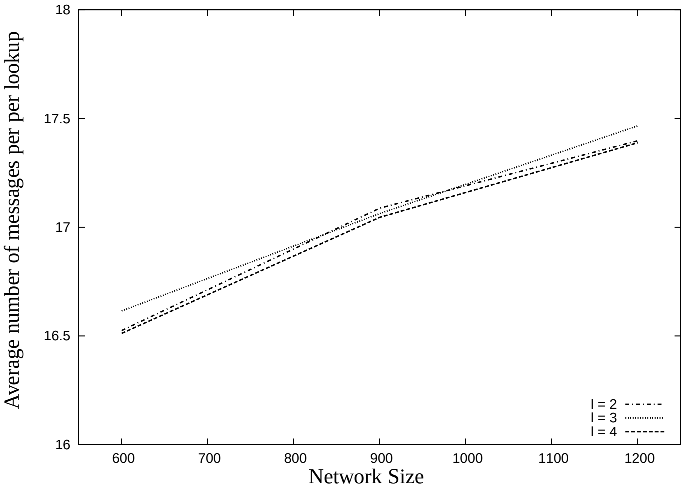

Fig. 30. Effect of l on lookup message count for a 3-way EpiChord network.

for a Chord node to fully initialize its finger table.

Although our reply messages will tend to be larger than those of traditional sequential lookup algorithms, since l “best” entries are returned, even with the increase in size, the reply messages are only about 100 bytes in size (including the 28-byte UDP/IP header) at a reasonable setting of $l = 3$. Hence, the increased size of the responses is not an issue even for nodes behind a 56k modem line since the packets are relatively small.

VI. Related Work

Our parallelized lookup algorithm and reactive cache management strategy can be applied to any of the existing DHT routing topologies that have some flexibility in the choice of neighbors (i.e., ring, tree or xor) [14]. We chose to implement our proof-of-concept DHT using the Chord ring [2] as the underlying routing topology because of its simplicity.

Like EpiChord, Kademlia [6] gathers routing information from observing lookup traffic and uses parallel lookups to improve lookup resilience. The organization of its routing entries is also somewhat analogous to that for EpiChord, albeit in a different address space. One key difference between Kademlia and EpiChord is that Kademlia limits the amount of routing state to $O(\log n)$ while EpiChord does not. By limiting its routing state to $O(\log n)$, Kademlia lookups take on average $O(\log n)$ hops while EpiChord can often achieve one- or two-hop lookup performance with its large routing state. While Kademlia employs parallel lookups mainly to improve lookup performance, EpiChord actually *requires* parallel lookups to cope with possible timeouts arising from maintaining a large amount of routing state.

The MIT Chord [20] implementation includes a *location cache*, i.e., nodes remember the IP address and ids of nodes that recently contacted them and use this information in their lookup. Zhuang and Zhou showed that the Chord location cache is able to reduce lookup path length by 1/2 of the logarithm of the cache size, but it does not scale to more than 2,000 nodes in a typical network setting because of stale cache entries, which cause timeouts and redundant hops [24].

In addition to proximity neighbor selection [14], Dabek et al. recently investigated the effectiveness of a combination of techniques in improving lookup latency for DHash++ [18] (an $O(\log n)$-state DHT based on Chord), including synthetic coordinates [15], erasure coding [17], integration of key lookups and data fetches and an integrated transport protocol (STP). EpiChord is certainly not as sophisticated, but we are not seeking to be. Most of the techniques in DHash++ are orthogonal to our lookup algorithm and can be integrated into EpiChord if so desired.

Gupta et al. proposed one- and two-hop schemes that disseminate global network membership changes using a background broadcast process that scales up to a million nodes [13]. Other two-hop schemes that have been proposed include Kelips [9] and Structured Superpeers [12]. The major drawbacks of these schemes are that they either impose a fixed (and relatively high) amount of constant background traffic on all nodes (even ones that are relatively inactive), and/or impose significant asymmetry in the bandwidth consumption across nodes in the network. In return, they are in general able to achieve somewhat better one- and two-hop lookup performance than EpiChord, which also often achieves $O(1)$-hop lookups, but only in an incidental and *laissez faire* manner and at a somewhat lower cost.

To the best of our knowledge, only Chord [20] has a strong stabilization algorithm that will provably fix loopy network configurations and their stabilization algorithm is specific to their lookup algorithm and cannot be applied generally to other DHT routing algorithms. Our *token-passing* stabilization mechanism can be applied to any DHT that has a circular address space.

VII. Future Work

Instead of limiting the number of concurrent queries that we allow a lookup to have in parallel at any instant in time to p, it might be desirable to let the number of concurrent queries be $p_{max} (> p)$ if the number of nodes in the network is large and the node caches are relatively sparse, since under such circumstances, the initial p nodes are separated from the node corresponding to id by many intermediate nodes. Having more concurrent queries p_{max} improves lookup latency and allows the querying node to learn about more nodes, thereby improving the quality of its node cache. Of course, there is a trade-off of increased lookup traffic.

Conceptually, $\hat{\gamma}$ can be used to adaptively adjust the cache entry expiration period. We can choose a target γ_t and the cache entry expiration period is incrementally decreased when $\hat{\gamma} > \gamma_t$, until $\hat{\gamma} \leq \gamma_t$. We have not implemented such a scheme, but it is straightforward to do so.

EpiChord is currently not fully optimized. There is still significant flexibility for nodes to adopt individual policies to further enhance and optimize their individual (and thereby global) lookup performance, if so desired. For example, a node that discovers a high rate of node failures within the network (i.e., from the fact that many queries are unacknowledged) can adaptively increase the number of parallel queries per lookup as well as be more aggressive in flushing old entries from its cache. One can also imagine improving the dissemination of routing state by piggybacking additional random node entries on requests or responses. If the system lookup rate is low or a higher level of background traffic can be tolerated, EpiChord can generate additional queries, or employ a hierarchical broadcast scheme [13] or a provably efficient epidemic cache exchange mechanism [25], to proactively increase the number of cached entries per node. Finally, it might perhaps be possible to formulate the performance optimization problem as a learning problem and apply some existing AI technique to optimize overall system performance by tuning system parameters at runtime depending on the operating conditions and constraints (i.e., amount of lookup traffic and available background bandwidth).

VIII. Conclusion

Our goal in this work is not to design the perfect DHT. Instead, our objectives are: (i) to explore the effectiveness of our new technique, where we combine parallel queries with a reactive cache management strategy, in allowing us to move from an $O(\log n)$-state-per-node DHT topology to an unlimited-state-per-node architecture; and (ii) to understand the trade-offs within the unlimited-state-per-node DHT design space.

Proximity routing has been shown to be effective in reducing DHT routing latency [14]. Although we do not track latency information or actively decide on which nodes to query based on proximity, our parallel asynchronous lookup approach in fact exploits proximity indirectly. The key observation here is that the final sequence of lookups that returns the correct answer first in our asynchronous parallel lookup algorithm is approximately equivalent to a proximity-optimized lookup sequence for the corresponding sequential lookup algorithm.

Our parallel lookup algorithm is simple and effective, and our reactive approach to routing state maintenance allows our DHT to adapt naturally to a range of lookup workloads. We have quantified the performance-cost trade-offs for our lookup algorithm and showed that we can reduce both lookup latencies and path lengths by a factor of 3 by issuing only 3 queries asynchronously in parallel per lookup and that the number of messages thus generated is in general no more than that for the corresponding sequential Chord lookup algorithm, and at most up to 50% more under high churn rates.

ACKNOWLEDGMENTS

The authors wish to thank Dina Katabi and John Wroclawski for useful discussions in the early stages of this work and Steve Bauer for his helpful comments on the initial draft of this paper. This research was supported by the NSF under Grant No. ANI-0082503 and Cooperative Agreement ANI-0225660.

REFERENCES

[1] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, "A scalable content-addressable network," in *Proceedings of the 2001 ACM SIGCOMM Conference*, August 2001.

[2] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan, "Chord: A scalable Peer-To-Peer lookup service for internet applications," in *Proceedings of the 2001 ACM SIGCOMM Conference*, August 2001, pp. 149–160.

[3] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Tech. Rep. UCB/CSD-01-1141, UC Berkeley, April 2001.

[4] Antony Rowstron and Peter Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.

[5] Dahlia Malkhi, Moni Naor, and David Ratajczak, "Viceroy: A scalable and dynamic emulation of the butterfly," in *Proceedings of 21st ACM Symposium on Principles of Distributed Computing (PODC'02)*, July 2002.

[6] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

[7] Gurmeet Manku, Mayank Bawa, and Prabhakar Raghavan, "Symphony: Distributed hashing in a small world," in *Proceedings of 4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

[8] Gurmeet Singh Manku, "Routing networks for distributed hash tables," in *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC 2003)*, Boston, Massachusetts, July 2003, ACM.

[9] Indranil Gupta, Ken Birman, Prakash Linga, Al Demers, and Robbert van Renesse, "Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead," in *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.

[10] Frans Kaashoek and David Karger, "Koorde: A simple degree-optimal distributed hash table," in *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.

[11] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman, "Skipnet: A scalable overlay network with practical locality properties," in *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, March 2003.

[12] Alper Mizrak, Yuchung Cheng, Vineet Kumar, and Stefan Savage, "Structured superpeers: Leveraging heterogeneity to provide constant-time lookup," in *Proceedings of the 4th IEEE Workshop on Internet Applications*, June 2003.

[13] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues, "Efficient routing for peer-to-peer overlays," in *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI 2004)*, March 2004, pp. 113–126.

[14] K. Gummadi, G. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica, "The impact of DHT routing geometry on resilience and proximity," in *Proceedings of the 2003 ACM SIGCOMM Conference*, 2003, pp. 381–394.

[15] Russ Cox, Frank Dabek, Frans Kaashoek, Jinyang Li, and Robert Morris, "Practical, distributed network coordinates," in *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, Cambridge, Massachusetts, November 2003, ACM SIGCOMM.

[16] T. Ng and H. Zhang, "Towards global network positioning," in *Proceedings of IEEE Infocom '02*, June 2002.

[17] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proceedings of ACM ASPLOS*. ACM, November 2000.

[18] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris, "Designing a DHT for low latency and high throughput," in *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI 2004)*, March 2004, pp. 85–98.

[19] FIPS 180-1, "Secure hash standard," Tech. Rep., US Department of Commerce/NIST, April 1995.

[20] Ion Stoica, Robert Morris, David Liben-Nowell, David Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," Tech. Rep., MIT LCS, 2002.

[21] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble, "A measurement study of peer-to-peer file sharing systems," in *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[22] ssfnet.org, "Scalable simulation framework," http://www.ssfnet.org.

[23] Jinyang Li, Jeremy Stribling, Robert Morris, M. Frans Kaashoek, and Thomer M. Gil, "DHT routing tradeoffs in network with churn," in *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS '04)*, February 2004.

[24] Li Zhuang and Feng Zhou, "Understanding Chord performance and topology-aware overlay construction for Chord," 2003, http://www.cs.berkeley.edu/˜zf/papers/chord_perf.pdf.

[25] Mor Harchol-Balter, Tom Leighton, and Daniel Lewin, "Resource discovery in distributed networks," in *Proceedings of the 18th annual ACM Symposium on Principles of Distributed Computing (PODC 1999)*. 1999, pp. 229–237, ACM Press.

APPENDIX

A. Pseudocode for Basic Lookup Algorithm

```
Let tried_set= set of nodes that have already been probed (initially empty)
    pending_set= set of queries that are currently pending (initially empty)
    answer= final answer for this query (initially $null$)
    best_predecessor= best known predecessor of $id$ probed
    best_successor= best successor heard from for $id$ probed

// To start the successor node for the identifier $id$.
findSuccessor($id$)
    // Gets from cache the best known successor of $id$,
    // excluding entries already found in tried_set.
    try_entry ← cache.getNext($id$, tried_set);
    sendQuery($id$, try_entry);
    for $i \leftarrow 0$ upto $p - 1$
        // Gets from cache the best known predecessor of $id$,
        // excluding entries already found in tried_set.
        try_entry ← cache.getPrev($id$, tried_set);
        if try_entry$\neq null$
        sendQuery($id$, try_entry);

// To send a query to node $n$ to look up identifier $id$.
sendQuery($id$, $n$)
    // Sends a UDP packet to node $n$ to lookup identifier $id$,
    // with information on the nodes currently being probed.
    sendLookupMessage($id$, $n$, pending_set)
    // Sets a timeout for node $n$.
    setTimeout($n$)
    tried_set.add($n$)
    pending_set.add($n$)

// This function is called when node $n$ receives a reply.
receiveReply($n$, $success$, reply_set)
    // Add all the entries received from $n$ to the cache.
    cache.addEntries(reply_set)
    pending_set.remove($n$)
    if $n.id \in (owner.id,$best_successor$)$
        best_successor ← $n$
    if $success = \mathbf{true}$
        answer ← reply_set.getAnswer();
        // return answer to the query.
        lookup_success(answer);
    else
        sendMoreQueries();

// This function is called if a timeout occurs for the query to node $n$.
timeout($n$)
    pending_set.remove($n$)
    sendMoreQueries();

// This function is called to send out more concurrent queries, if necessary.
sendMoreQueries();
    // Gets from cache the node try_entry which closest to $id$
    // such that try_entry.$id \in$ (best_predecessor,best_successor),
    // excluding entries already found in tried_set.
    while (|pending_set| $< p_{max}$) $\bigwedge$ (try_entry$\neq null$)
        if try_entry.$id \in (owner.id, id)$
            best_predecessor ← try_entry
        sendQuery($id$, try_entry)
        try_entry ← cache.getBestEntry($id$, tried_set);
    if |pending_set| $= 0$
        // return lookup failure.
        lookup_failure();
```

B. Analysis of Expected Worst-Case Lookup Performance

To analyze the expected worst-case lookup performance, we consider the following scenario. Suppose we are at a node with id x and we are trying to resolve an id y, s.t. $y \in (x+2^i, x+2^{i+1})$. The range $(x+2^i, x+2^{i+1})$ is the size of one bucket in x's cache. This means that we have at least j entries in the bucket and hence we can certainly find node z s.t. $z \in (x+2^i, x+2^{i+1})$.

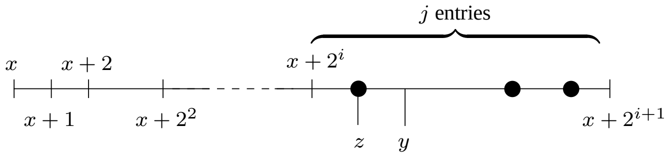

Fig. 31. Analysis of expected worst-case lookup performance.

Since $y \in (x+2^i, x+2^{i+1})$, so we know that $|x-y| \leq 2^{i+1}$, but because $z \in (x+2^i, x+2^{i+1})$, we know that $|z-y| \leq 2^i$. Hence, in each lookup step, even if the actual distance to the destination id is not reduced, the maximum possible distance is steadily reduced by at least a factor of two even in the worst case. This implies that lookups can be made in $O(\log n)$ hops in the worst case.

The bound derived from the above analysis is very loose because it is based only on the assumption that there is at least one other entry in the same bucket as the destination node. Since we have at least j entries in the bucket, we can clearly do significantly better. Under the assumption that x, y and all j entries in the $(x+2^i, x+2^{i+1})$ bucket are independent and uniformly distributed, we can show with some elementary probability that:

$$\frac{E(|x-y|)}{E(|z-y|)} \quad = \quad 3j + \frac{6}{j+3}$$

Proof:

$$Pr(min \geq y|x) \quad = \begin{cases} (1-x-y)^j & \text{if } x < y \\ (1-2y)^j & \text{if } y < x < 1-y \\ (x-y)^j & \text{if } 1-y < x \end{cases}$$

If x is uniformly distributed,

$$\begin{aligned} Pr(min \geq y) &= \int_0^1 Pr(min \geq y|x)p(x)dx \\ &= \int_0^1 Pr(min \geq y|x)dx, \text{ since } p(x) = 1 \end{aligned}$$

If $y < 0.5$,

$$\begin{aligned} Pr(min \geq y) &= \int_0^y (1-x-y)^j dx + \int_y^{1-y} (1-2y)^j dx \\ &\quad + \int_{1-y}^1 (x-y)^j dx \\ &= \left[\frac{-1}{j+1}(1-x-y)^{j+1}\right]_0^y + \left[x(1-2y)^j\right]_y^{1-y} \\ &\quad + \left[\frac{1}{j+1}(x-y)^{j+1}\right]_{1-y}^1 \end{aligned}$$

$$\begin{aligned}
&= \frac{-1}{j+1}\left[(1-2y)^{j+1}-(1-y)^{j+1}\right] \\
&\quad +((1-y)-y)(1-2y)^j \\
&\quad +\frac{1}{j+1}\left[(1-y)^{j+1}-(1-2y)^{j+1}\right] \\
&= \frac{2}{j+1}\left[(1-y)^{j+1}-(1-2y)^{j+1}\right] \\
&\quad +(1-2y)^{j+1} \\
&= \frac{2}{j+1}(1-y)^{j+1}+\frac{j-1}{j+1}(1-2y)^{j+1}
\end{aligned}$$

If $y > 0.5$,

$$\begin{aligned}
Pr(min \geq y) &= \int_0^{1-y}(1-x-y)^j dx + \int_y^1 (x-y)^j dx \\
&= \left[\frac{-1}{j+1}(1-x-y)^{j+1}\right]_0^{1-y} \\
&\quad + \left[\frac{1}{j+1}(x-y)^{j+1}\right]_y^1 \\
&= \frac{2}{j+1}(1-y)^{j+1}
\end{aligned}$$

$$\begin{aligned}
Pr(min \leq y) &= 1 - Pr(min \geq y) \\
\Rightarrow p(min) &= \begin{cases} 2(1-y)^j & \\ +2(j-1)(1-2y)^j, & \text{if } y < 0.5 \\ 2(1-y)^j, & \text{if } y > 0.5 \end{cases} \\
E(min) &= \int_0^1 yp(y)dy \\
&= \int_0^1 2y(1-y)^j dy \\
&\quad + \int_0^{0.5} 2(j-1)y(1-2y)^j dy \\
&= \left[-\frac{2}{j+1}y(1-y)^{j+1}\right]_0^1 \\
&\quad + \int_0^1 \frac{2}{j+1}(1-y)^{j+1}dy \\
&\quad - \left[\frac{j-1}{j+1}y(1-2y)^{j+1}\right]_0^{0.5} \\
&\quad + \int_0^{0.5} \frac{j-1}{j+1}(1-2y)^{j+1}dy \\
&= \left[-\frac{2}{(j+1)(j+2)}(1-y)^{j+2}\right]_0^1 \\
&\quad - \left[\frac{j-1}{2(j+1)(j+2)}(1-2y)^{j+2}\right]_0^{0.5} \\
&= \frac{2}{(j+1)(j+2)} + \frac{j-1}{2(j+1)(j+2)} \\
&= \frac{j+3}{2(j+1)(j+2)}
\end{aligned}$$

If we now consider the original scenario, where $y, z \in (x+2^i, x+2^{i+1})$, by the fact that the node ids are uniformly distributed,

$$\begin{aligned}
E(|x-y|) &= \frac{2^i+2^{i+1}}{2} \\
&= 2^{i-1}+2^i \\
E(|z-y|) &= \frac{j+3}{2(j+1)(j+2)}2^i \\
\frac{E(|x-y|)}{E(|z-y|)} &= \frac{3(j+1)(j+2)}{j+3} \\
&= 3j + \frac{6}{j+3}
\end{aligned}$$