Proceedings of the *12th International Conference on Networks 2004 (ICON 2004)*, Singapore, November 2004.

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

Ben Leong, Barbara Liskov, and Eric D. Demaine
MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street
Cambridge, MA 02139, USA
{benleong, liskov, 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 simulations show that 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 in general no more than that for the corresponding sequential Chord lookup algorithm.

I. Introduction

In recent years, many Distributed Hash Tables (DHTs) have been proposed [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. 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. 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 [11].

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.

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 piggybacking 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.

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.

Our main goal in this work 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 and 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 maintenance and lookup traffic. 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 and the node responsible for a key is the node whose identifier most closely follows the key, i.e., the successor (see Fig. 1). In addition to maintaining a successor list of k nodes, nodes in our network also maintain a predecessor list of k nodes. Nodes communicate with their

[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 [9] achieves this $O(\log n/\log\log n)$-hop lower bound.

[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.

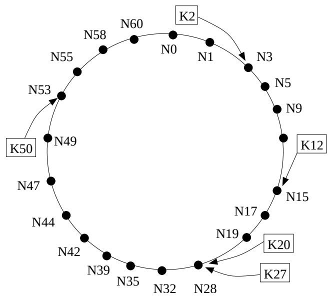

Fig. 1. Circular identifier address space with twenty nodes and five keys.

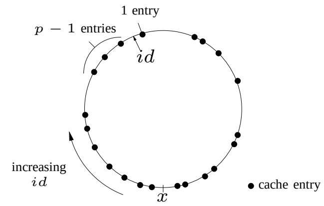

Fig. 2. Cache entries returned from cache for node x for a lookup of id.

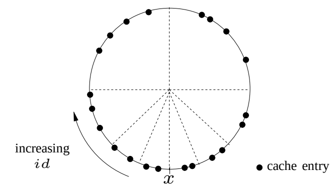

Fig. 3. Division of address space into slices with respect to node x.

immediate successor and predecessor periodically, exchanging their entire successor and predecessor lists. Instead of maintaining a finger table (set of pointers) with $O(\log n)$ entries, EpiChord maintains a cache that not only guarantees at least $O(\log n)$-hop performance, but can often do better.

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.

To look up a destination 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 Fig. 2). Probing the succeeding node gives us a chance of locating the destination node in one hop. When contacted, each of the p nodes will provide its l "best" next hops from its cache or if it owns id, it will say so. When these replies are received, further queries will be dispatched asynchronously in parallel if x learns about nodes that are closer to the destination id than the other queries that are still pending. We call an EpiChord network where p queries are made in parallel a *p-way* EpiChord.

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.

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.

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 sets of exponentially smaller slices as shown in Fig. 3[3]. For performance guarantees, each 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 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 $k \geq 2j$.

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 [12]). When $j = 1$, we get the same expected worst-case result as Chord does. However, for $j \geq 2$, we tend to do

[3] 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.

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[4]. 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 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[5]. 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 (which is a reasonable assumption since node ids are obtained using the SHA-1 hash [15] and are thus uniformly distributed), the proportion of lookup traffic that will help to satisfy the cache invariants for various network sizes (for $j = 2$) is shown in Fig. 4. With an amount of lookup traffic approximately equal to the required background maintenance traffic (i.e., $x = 1$ in Fig. 4), we can achieve a 42% reduction in the background maintenance traffic when $n = 2,000$. At larger network sizes, the savings in background maintenance traffic is reduced, but even at network sizes of 1,000,000 nodes, we can still expect a reduction of more than 25% on average. The reduction in background probes is relatively independent of j.

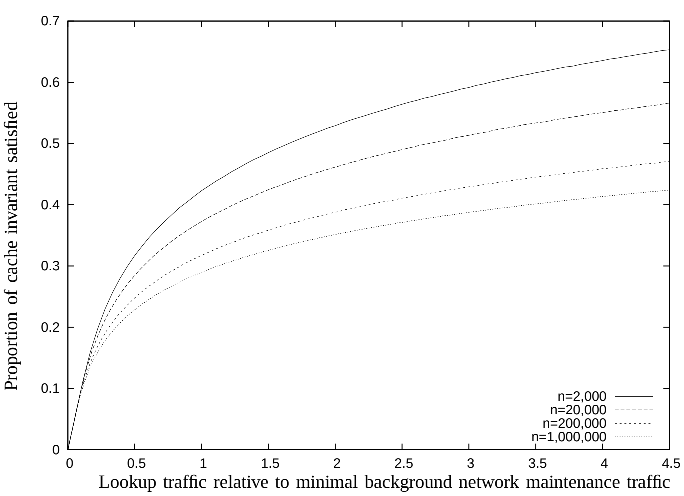

Fig. 4. Effect of network size (n) on the proportion of lookup traffic that helps to satisfy cache invariant (for $j = 2$).

[4]The expected lookup path length for Chord is $\frac{1}{2}\log_2 n$ [13].

[5]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. [14].

C. Cache Composition in the Steady State

The proportion of live entries[6] 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:

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

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:

$$\lim_{t\to\infty} x(t) = \frac{pQ}{pQ + (f + r - rf)n} n \quad (2)$$

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

$$\frac{d}{dt}y(t) = \overbrace{(1-f)rx}^{\substack{\text{stale entries}\\\text{not flushed}}} - \overbrace{fy}^{\substack{\text{stale entries}\\\text{flushed}}} - \overbrace{pQ(\frac{y}{x+y})}^{\substack{\text{stale entries discovered by}\\\text{timeouts of outgoing queries}}} \quad (3)$$

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

$$\gamma = \lim_{t\to\infty} \frac{y}{x+y} = \frac{1}{pQ}\left[(1-f)rx - fy\right] \quad (4)$$

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.

Next, we consider the case when $pQ \ll rn$. By setting $\frac{dx}{dt} = 0$ in (1) and $\frac{dy}{dt} = 0$ in (3), we obtain:

$$\gamma = \lim_{t\to\infty} \frac{y}{x+y} = \frac{\sqrt{1 + \frac{(1-f)r}{f}} - 1}{\sqrt{1 + \frac{(1-f)r}{f}}} \quad (5)$$

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

$$\gamma = \frac{\sqrt{2-f} - 1}{\sqrt{2-f}} \leq 1 - \frac{1}{\sqrt{2}} = 0.292 \quad (6)$$

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.

[6]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.

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* [16] 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 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. 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.

Li et al. highlighted that the assumed workload will affect the result of comparisons between DHTs significantly [17]. 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 pre-determined 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[7]. 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 $j = 2$.

1) Lookup Performance: The average latency and the average hop count per lookup for successful lookups in the steady state are shown in Figs. 5 and 6 respectively. From Fig. 5, we see that having more parallelism reduces the lookup latency. In Fig. 6, 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 $\frac{1}{2}\log_\alpha n = \frac{1}{2}\log_{7.2} 1,200 = 1.80$.

Lookup failure rates are relatively low ($< 0.1\%$) under a lookup-intensive workload. This is not surprising since under the lookup-intensive workload, the large number of lookups keep the routing state for most nodes generally up-to-date. Our results also show that adding more parallelism (increasing p) reduces the probability of lookup failure significantly[8]. The lookup failure probability falls by approximately an order of magnitude when p is increased by one.

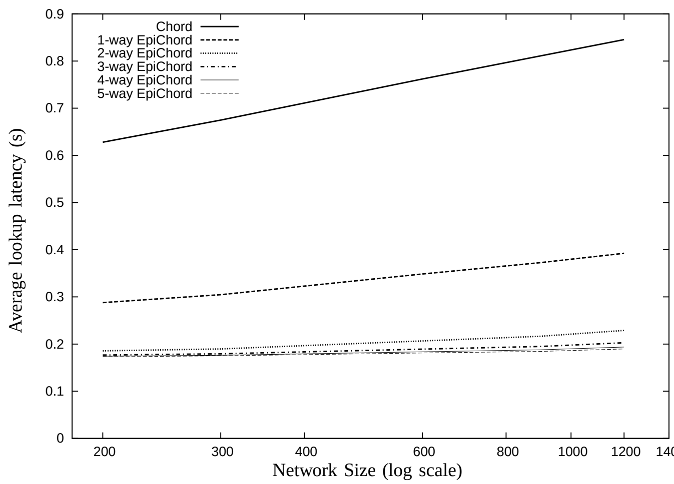

Fig. 5. Comparison of lookup latency between Chord and p-way EpiChord under lookup-intensive workload.

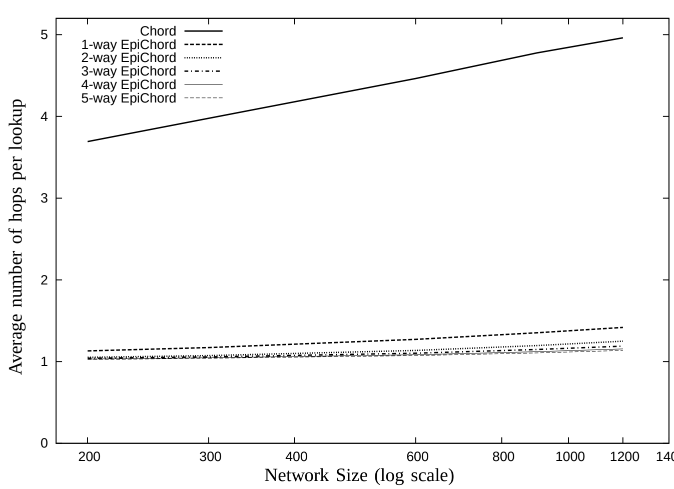

Fig. 6. Comparison of lookup path length between Chord and p-way EpiChord under lookup-intensive workload.

[7]The cache entry expiration period of 120 s is chosen to ensure that the amount of maintenance traffic required is provably less than that for a corresponding Chord network at the same level of network churn. Details can be found in the EpiChord technical report [12].

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

2) Message Count: It is clear that a parallel lookup algorithm will generate more lookup messages when there are more parallel queries per lookup. Fig. 7 shows that for our given parameter settings, the average number of query and reply messages that are required for a sequential Chord network is approximately equal to that for a 3-way EpiChord network. 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 Figs. 8 and 9 respectively. Again, we see from Fig. 9 that adding more parallelism reduces the lookup latency significantly. As shown in Fig. 10, the lookup failure probabilities under the churn-intensive workload are higher than those under the lookup-intensive workload (which are $\leq 0.1\%$). From Fig. 10, we see that the failure rates for the 4- and 5-way EpiChord networks are higher than that for the 3-way EpiChord network, which is somewhat counter-intuitive. We discovered that the explanation for 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. Our simulations also suggest that with $p \geq 2$, successful lookups will almost never experience timeouts.

2) Message Count: As shown in Fig. 11, 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.

C. Varying Other System Parameters

Our simulations also show that:

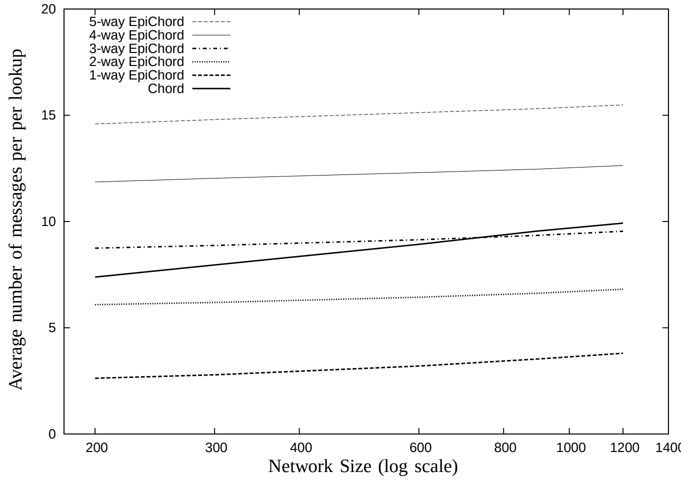

Fig. 7. Comparison of lookup message count between Chord and p-way EpiChord under lookup-intensive workload.

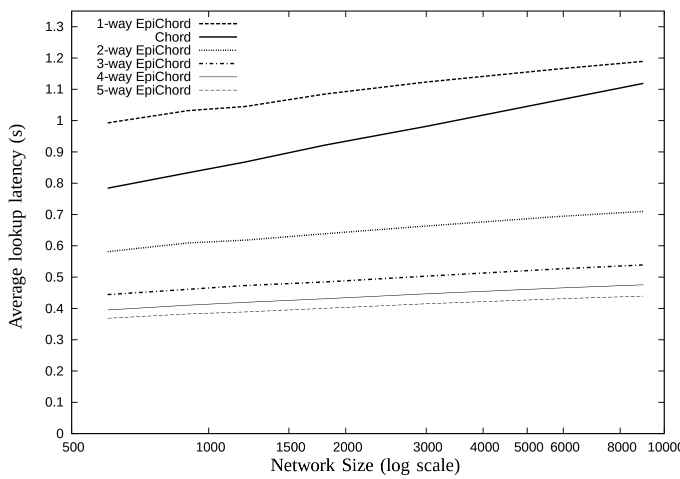

Fig. 8. Comparison of lookup latency between Chord and p-way EpiChord under churn-intensive workload.

- Holding p and l constant at 3, increasing the amount of lookup traffic per node Q (varying between 0.01 and 2 lookups per second) 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., $Q = 0.5$).
- Similarly, holding p constant at 3 and the amount of lookup traffic Q constant at 0.01 lookups per node per second, the number l of "best entries" returned per response (varying between 2 and 4) has a negligible 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

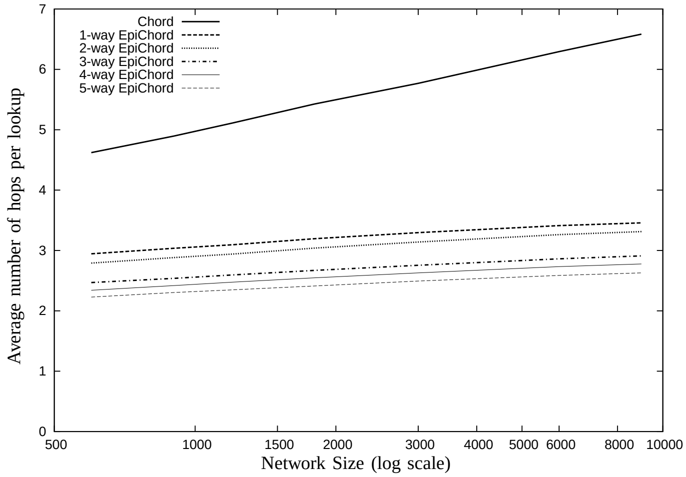

Fig. 9. Comparison of lookup path length between Chord and p-way EpiChord under churn-intensive workload.

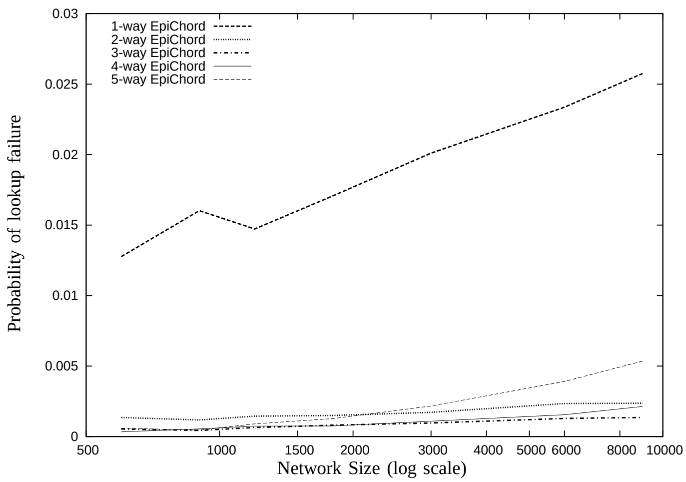

Fig. 10. Lookup failure rates for p-way EpiChord networks under churn-intensive workload.

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 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 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[9]. Although performance is better with

[9]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.

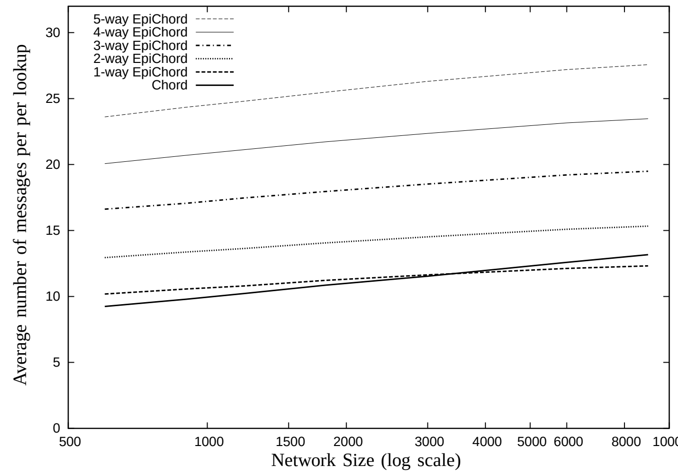

Fig. 11. Comparison of lookup message count between Chord and p-way EpiChord under churn-intensive workload.

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 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) [18]. 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 [13] 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 unfortunately, 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 [19].

In addition to proximity neighbor selection [18], Dabek et al. recently investigated the effectiveness of a combination of techniques in improving lookup latency for DHash++ [20] (an $O(\log n)$-state DHT based on Chord), including synthetic coordinates, erasure coding, 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 [11]. Other two-hop schemes that have been proposed include Kelips [8] and Structured Superpeers [10]. 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.

VII. 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 [18]. 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 also 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.

REFERENCES

[1] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A scalable content-addressable network,” in *Proceedings of the 2001 ACM SIGCOMM Conference*, August 2001.

[2] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. 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,” UC Berkeley, Tech. Rep. UCB/CSD-01-1141, April 2001.

[4] A. Rowstron and P. 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] D. Malkhi, M. Naor, and D. Ratajczak, “Viceroy: A scalable and dynamic emulation of the butterfly,” in *Proceedings of 21st ACM Conf. 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] G. Manku, M. Bawa, and P. Raghavan, “Symphony: Distributed hashing in a small world,” in *Proceedings of 4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

[8] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse, “Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead,” in *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.

[9] F. Kaashoek and D. 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.

[10] A. Mizrak, Y. Cheng, V. Kumar, and S. Savage, “Structured superpeers: Leveraging heterogeneity to provide constant-time lookup,” in *Proceedings of the 4th IEEE Workshop on Internet Applications*, June 2003.

[11] A. Gupta, B. Liskov, and R. 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.

[12] B. Leong, B. Liskov, and E. D. Demaine, “Epichord: Parallelizing the chord lookup algorithm with reactive routing state management,” MIT, Cambridge, MA, Technical Report MIT-LCS-TR-963, August 2004.

[13] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” MIT LCS, Tech. Rep., 2002.

[14] S. Saroiu, P. K. Gummadi, and S. 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.

[15] “FIPS 180-1. secure hash standard,” US Department of Commerce/NIST, Tech. Rep., April 1995.

[16] The ssfnet project. [Online]. Available: http://www.ssfnet.org

[17] J. Li, J. Stribling, R. Morris, M. F. Kaashoek, and T. 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.

[18] 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*, August 2003, pp. 381–394.

[19] L. Zhuang and F. Zhou, “Understanding Chord performance and topology-aware overlay construction for Chord,” 2003. [Online]. Available: http://www.cs.berkeley.edu/~zf/papers/chord_perf.pdf

[20] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. 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.