EpiChord: Large Routing State plus Parallel Queries equals Improved DHT Lookup Performance and Resilience

Ben Leong
benleong@mit.edu

Barbara Liskov
liskov@csail.mit.edu

Erik D. Demaine
edemaine@mit.edu

***Abstract*— We present EpiChord, a distributed hash table (DHT) that achieves lookup resilience by issuing parallel asynchronous queries. EpiChord caches large amounts of routing state at each node and uses multiple parallel queries to mitigate the effects of outdated entries, thus ensuring that lookup performance does not degrade even when network churn is high. By ensuring cached entries are well-distributed in the id namespace, EpiChord guarantees $O(\log n)$-hop lookup performance in the worst case and can often attain $O(1)$-hop lookup performance in the common case if network churn is low. EpiChord has a low maintenance cost since it exploits information gleaned from observing lookup traffic to improve lookup performance, and only sends network probes occasionally as a backup mechanism. Our experience with EpiChord demonstrates that we can achieve significant gains in lookup performance and resilience by moving from a limited-state-per-node to a large-state-per-node DHT architecture without incurring a significant cost in additional network traffic.**

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]. Most initial DHT research was directed towards minimizing the amount of routing state per node. The motivation for minimizing routing state is clear. In a dynamic network with a relatively high churn rate, routing state will inevitably become outdated over time and timeouts will occur when a node tries to query another node that has failed or has left the network. The general assumption is that if the number of fingers is reasonably small ($O(\log n)$), a node can periodically refresh all the fingers at a frequency high enough to make the probability of timeouts negligible. This is a reasonable approach, but it does have two major problems: (i) it is not easy to determine the optimal refresh rate adaptively – setting it too low will result in excessive timeouts, while setting it too high will result in excessive background maintenance traffic; (ii) there will always be a non-zero probability of timeouts under any reasonable setting for the refresh rate and it is still costly to recover from timeouts.

In this paper, we describe EpiChord, a DHT that achieves lookup resilience by issuing parallel asynchronous queries to avoid the need of having to recover from timeouts. We can afford to do so without generating excessive amounts of lookup traffic because we maintain a large amount of routing state at each node, which reduces the number of hops per lookup and thereby the number of lookup messages significantly. Furthermore, parallel lookups allow us to store a large amount of routing state without excessive cost because we can tolerate outdated state without having to worry about timeouts.

Existing DHTs tend to decouple the lookup process from routing state maintenance. In EpiChord, routing state maintenance costs are amortized into the lookup costs as 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.

Although our parallelized lookup algorithm 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) [10], we chose to implement our proof-of-concept DHT using the Chord ring [2] as the underlying routing topology. We chose Chord because of its simplicity and its many provable guarantees on stability and robustness [11], which EpiChord automatically inherits by adopting Chord's stabilization algorithm.

EpiChord ensures that entries in the cache are distributed with a particular bias towards the nearby nodes, which allows us to achieve a worst-case lookup performance guarantee of $O(\log n)$ hops. Nodes are expected to have βn $(0 < \beta < 1)$ up-to-date cache entries in steady state, so lookups can be resolved in one hop with probability β.

Although one might expect a parallel-lookup algorithm to generate significantly more network traffic, we show that in practice we are able to achieve significantly better lookup performance on average than that the corresponding sequential Chord lookup algorithm with comparable amounts of lookup traffic.

II. Overview of EpiChord

Like Chord, EpiChord is organized as a one-dimensional circular namespace and the node responsible for a key is the node whose identifier most closely follows the key, i.e., the successor. 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 immediate successor and predecessor periodically, exchanging their entire successor and predecessor lists. Instead of maintaining a finger table 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.

To allow nodes to learn about other nodes efficiently, we use iterative lookups. We also 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 preceeding id, within the set of nodes known to it (see Figure 1). 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 simply 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 lookups are made in parallel a *p-way* EpiChord.

Each cache entry has an associated time. When a node receives a query or reply, it sets (or resets) the time of 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 that node'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. Nodes are flushed if they don't respond to some number of queries or when their lifetime exceeds some limit. Note that this scheme requires synchronized clock rates but not synchronized clocks. Furthermore, our preliminary experiments indicate that the cache management scheme does not have a big impact on performance and we suspect that any reasonable scheme will work.

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 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. Similarly, Epi-

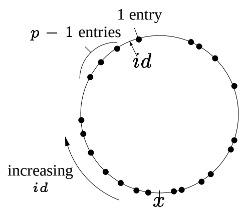

Fig. 1. Cache entries returned from cache for node x for a lookup of id.

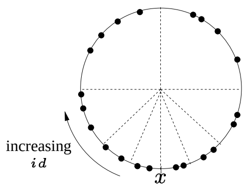

Fig. 2. Division of address space into slices with respect to node x.

Chord divides the namespace into two symmetric sets of exponentially smaller slices as shown in Figure 2. For performance guarantees, a node simply enforces the following invariant:

Cache Invariant: *Every slice contains at least j cache entries at all times.*

The key idea is that to provide an $O(\log n)$-hop guarantee on the lookup pathlength, the density of entries per slice must increase exponentially as we get nearer to the reference node. 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 $k \geq 2j$.

Since we need only $O(\log n)$ cache entries to guarantee $O(\log n)$-hop performance and we have $O(n)$ entries in the cache, the probability that the invariant is satisfied is very high if the $O(n)$ entries are approximately uniformly distributed. In the unlikely event that a node discovers there are insufficient cache entries for a given slice, it simply makes a lookup to the midpoint of that slice.

III. Analysis

A. Lookup Performance

If we assume a uniformly distributed workload, we can show that worst-case lookup performance is $O(\log n)$ hops. The expected worst-case lookup pathlength 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. (The proof is omitted because of space constraints.) This result indicates that for $j=1$ 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 pathlengths are at most only $\log_\alpha 2 \approx \frac{1}{3}$ of that for Chord. Nodes are expected to have βn $(0 < \beta < 1)$ cache entries in the steady state, so lookups are resolved in one hop with probability β.

B. Cache Size and Composition in the Steady State

The proportion of live[1] entries in the cache compared to the total network size is an important system parameter

[1] 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, outdated entries.

because it determines the probability of one-hop success and the overall performance of the system. 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 and each node makes Q lookups uniformly over the id namespace and sends out p queries in parallel for each lookup. Where x is the number of nodes that is known to a node at time t, we obtain the following relation:

$$\frac{d}{dt}x(t) = \overbrace{Qp(1-\frac{x}{n})}^{\text{outgoing queries}} + \overbrace{Qp(1-\frac{x}{n})}^{\text{incoming queries}} - \overbrace{rx}^{\text{nodes departing}} \quad (1)$$

The solution to (1) is:

$$x(t) = x_0 + \left(\frac{2pQ}{2pQ+rn}n - x_0\right)\left(1 - e^{-\frac{2pQ+rn}{n}t}\right) \quad (2)$$

where x_0 is the number of node entries that a node gets from an existing node when it first joins the network. It is not hard to show that $\lim_{t\to\infty} x_0 = \frac{2pQ}{2pQ+rn}n$, i.e., $\beta \approx \frac{2pQ}{2pQ+rn}$. We can show with a similar analysis that if queries are made uniformly over the entire id namespace, the distribution of entries within the cache is approximately uniform in the steady state.

Suppose we have a network of 100,000 nodes and 0.001% (or 10) of the nodes leave on average each second[2]. If the nodes each make one lookup on average per minute and 5 query messages are sent in parallel for each lookup, nodes can be expected to know about $\frac{2\times5\times100{,}000}{2\times5+10\times60} \approx 1{,}600$ live nodes at any one time in the steady state when doing hardly any work. On the other hand, we only need approximately $2\log_2(100{,}000) \approx 34$ nodes for $j = 2$ to guarantee $O(\log n)$ routing performance. Since there is significant flexibility in the choice of the 34 nodes, the 1,600 live cache entries are likely to be able to satisfy the cache invariant.

C. Tradeoff in Messaging Costs between Small-State Sequential Lookup and Large-State-Per-Node (LSPN) Parallel Asynchronous Lookup

EpiChord join costs are low because a new node only needs to register with its two immediate neighbors and it can obtain its entire initial routing cache from either of them. In contrast, a Chord node has to make $O(\log n)$ lookups to locate a set of valid fingers. Although EpiChord requires a larger amount of initial state ($O(n)$) to be transferred, a resource-constrained node can choose to obtain only $j\log n$ entries from its neighbor instead of βn entries at the small cost of some degradation in its initial lookup performance.

A Chord node has to periodically refresh all the fingers. In contrast, the number and frequency of routing state maintenance messages for EpiChord are expected to be significantly lower as most of the cost associated with such maintenance is amortized into the lookup cost.

An iterative Chord lookup will require $\log_2 n$ messages on average, while a corresponding EpiChord lookup will require $p\log_\alpha n$ messages in the worst-case. This means that even though several lookup messages can be generated in parallel per lookup hop for EpiChord, the number of query messages for EpiChord exceeds Chord only when $p\log_\alpha n > \log_2 n$, i.e., $p > \log_2 \alpha$ (in the worst case). For $j = 2$, this means that we can support $p = 3$ (which gives a good degree of lookup resilience) with no increase in lookup traffic.

Furthermore, all lookup messages for EpiChord are useful: If a node sends a lookup and the queried node responds, both of the nodes have learnt that each other is alive. If the queried node fails to respond and the query times out, the effort is not wasted either because the querying node would have learnt that the queried node is either temporarily unavailable, has failed, or has left the network.

IV. Simulation Results

To further understand the tradeoffs when we move from a limited-state-per-node DHT to a large-state-per-node DHT with the same basic routing topology, we compared EpiChord to a corresponding optimal[3] iterative Chord network of the same size and with the same set of node ids using our network simulator, which is written in Java. The resulting lookup performance in terms of number of hops and messaging costs is shown in Figures 3 and 4 respectively. Although one would expect our algorithm to generate significantly more traffic because of parallel lookups, these results show that this is not the case because fewer hops are needed per lookup. In particular, for our chosen parameter settings ($k = 4$, $Q = 3$, $r = 0.1\%$, $l = 3$), the amount of lookup traffic generated by a 4-way EpiChord network is roughly the same as that of a corresponding optimal Chord network.

Although increasing the number of parallel queries p does not significantly improve lookup pathlength, it reduces the probability of timeouts significantly in the face of network churn as shown in Figure 5, where we simulated a relatively dynamic network ($r = 5\%$) with a fairly *laissez faire* cache maintenance policy (i.e. cache entries are flushed at a rate much slower than the churn rate).

[2]This is the approximate node departure rate of the Gnutella network derived from the results of a recent measurement study [12], [9].

[3]By optimal, we mean that the finger tables of the Chord nodes have accurate finger entries at all times regardless of node failures.

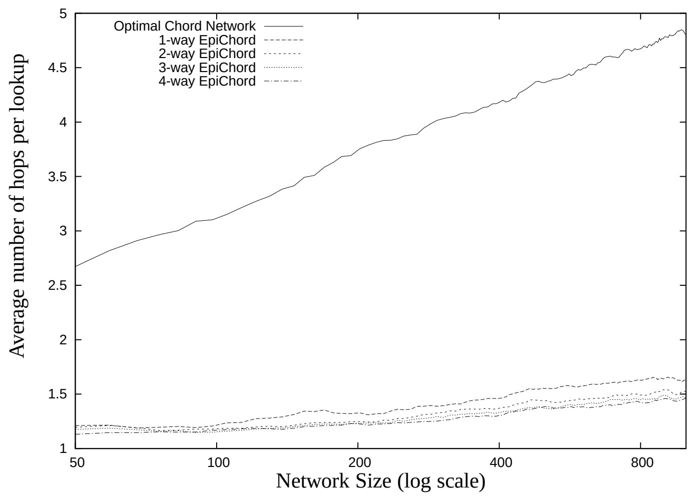

Fig. 3. Comparison of lookup performance between Chord and p-way EpiChord ($k = 4, Q = 3, r = 0.1\%, l = 3$).

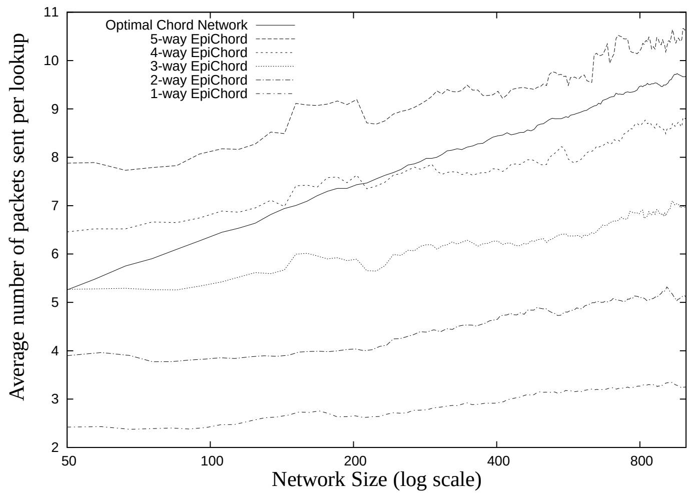

Fig. 4. Comparison of lookup messaging cost between Chord and p-way EpiChord ($k = 4, Q = 3, r = 0.1\%, l = 3$).

Our simulations also show that:

- Holding p and l constant at 3, the amount of lookup traffic Q (varying between 1 and 6) has only a marginal effect on either the lookup pathlength or the number of messages sent for the range of r from 0.1% to 5%. This implies that although our scheme requires a sufficient level of traffic to perform well, its actual level of performance is not highly dependent on the amount of traffic observed.
- Holding p constant at 3 and Q constant at 1, the number l of "best entries" returned per response (varying between 3 and 5) has neglible effect on either the lookup pathlength or the number of messages sent for the range of r from 0.1% to 5%. We thus conclude that we can keep l small and set $l = 3$.

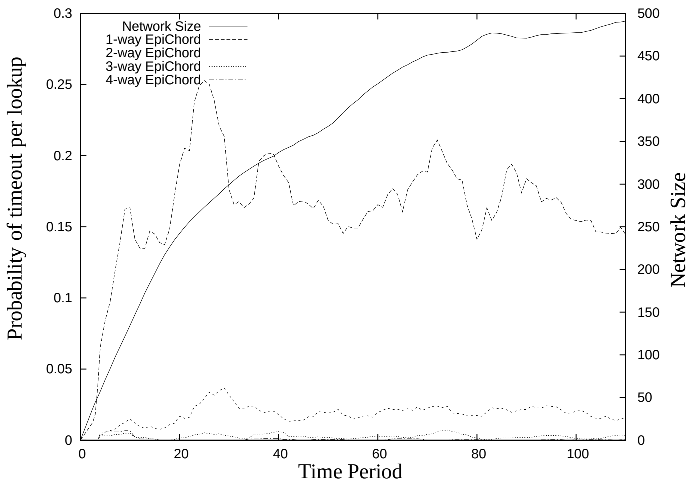

Fig. 5. Probability of timeouts with a fairly *laissez faire* cache maintenance policy ($k = 4, Q = 1, r = 5\%, l = 3$).

V. Related Work

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 namespace. The key difference between EpiChord and Kademlia 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. This is means that with the same degree of parallelism, Kademlia is likely to generate significantly more lookup traffic than EpiChord. While Kademlia employs parallel lookups mainly to improve lookup performance, EpiChord employs parallel lookups mainly to cope possible timeouts arising from maintaining a large amount of routing state. As shown in Figure 3, more parallelism in the lookups for EpiChord improves lookup pathlength only marginally. The main lookup performance gain for EpiChord comes from maintaining large routing state.

Proximity routing has been shown to be effective in improving DHT routing performance [10]. As observed in Kademlia, having parallel asynchronous lookup queries indirectly optimizes for proximity and hence will improve lookup latency in general. This also applies to EpiChord. The key observation here is that the final sequence of lookups that returns the correct answer first in an asynchronous parallel lookup algorithm is equivalent to a proximity-optimized lookup sequence for the corresponding sequential lookup algorithm. We have not attempted to characterize this performance gain because it is clearly highly dependent on the network topology.

Gupta et al. [9] proposed a scheme that disseminates global network membership changes to all nodes using a background broadcast process. They further showed that their global broadcast scheme is feasible in terms of both storage and bandwidth consumption for large networks with up to a million nodes. Kelips [7] uses an epidemic algorithm to propagate changes through the system. Kelips does not provide any guarantees on worst-case performance and relies on a random walk-based search to locate a file when information on the required file is not cached locally. Both these schemes impose a fixed amount of constant background traffic on all nodes, even ones that are relatively inactive. We recognize however that if we want to provide stringent guarantees on routing pathlength (i.e. one-hop), a proactive scheme is likely to be necessary.

VI. Conclusion

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 pathlengths and latencies with little additional costs in terms of complexity and bandwidth consumption.

Our simulations have also shown that even though multiple messages are sent per lookup step, the number of packets sent per lookup is not significantly larger than that for a sequential lookup algorithm because lookup pathlengths are significantly shorter. This is a desirable tradeoff because lookup latency is the principal measure of lookup performance.

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 messages are still less than 100 bytes in size (including the TCP/IP headers) 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.

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. Finally, if a higher level of background traffic can be tolerated, EpiChord can also employ a provably efficient epidemic cache exchange mechanism [13] to increase the number of cached entries.

This paper describes work in progress. EpiChord is an attempt at understanding the tradeoffs within the large-state-per-node DHT design space. Our experience with EpiChord demonstrates that we can achieve significant gains in lookup performance and resilience by moving from a limited-state-per-node to a large-state-per-node DHT architecture without incurring a significant cost in additional network traffic.

References

[1] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, "A scalable content addressable network," Tech. Rep. TR-00-010, Berkeley, CA, 2000.

[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*, 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, Apr. 2001.

[4] Antony Rowstron and Peter Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *Lecture Notes in Computer Science*, vol. 2218, pp. 329–??, 2001.

[5] 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.

[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] 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)*, 2003.

[8] 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)*, 2003.

[9] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues, "One hop lookups for peer-to-peer overlays," in *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, May 2003.

[10] 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.

[11] 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.

[12] 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.

[13] Mor Harchol-Balter, Tom Leighton, and Daniel Lewin, "Resource discovery in distributed networks," in *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*. 1999, pp. 229–237, ACM Press.