

Achieving One-Hop DHT Lookup and Strong Stabilization by Passing Tokens

Ben Leong and Ji Li
MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street
Cambridge, MA 02139, USA
{benleong, jli}@mit.edu

Abstract—Recent research has demonstrated that if network churn is not excessively high, it becomes entirely reasonable for a Distributed Hash Table (DHT) to store a global lookup table at every node to achieve one-hop lookup. We present a novel algorithm for maintaining global lookup state in a DHT with a Chord-like circular address space. In our DHT, events are disseminated with a parallelized token-passing algorithm using dynamically-constructed dissemination trees rooted at the source of the events. We show that we are able to achieve good one- and two-hop routing performance at a modest cost in bandwidth. Furthermore, our scheme is bandwidth-adaptive, and automatically detects and repairs global address space inconsistencies.

I. BACKGROUND

Most of the initial Distributed Hash Tables (DHTs) were designed to cope with membership changes in highly dynamic (i.e., high churn) networks [1], [2], [3], [4]. Since storage has grown significantly cheaper and higher bandwidth networks have become more common in recent years, recent research has demonstrated that if network churn is not excessively high, it has become entirely reasonable to store a global lookup table at every node to achieve one-hop lookup [5], [6].

In this paper, we present a DHT that uses a novel token-passing mechanism to maintain global lookup state at each node. We expect our DHT to be useful for peer-to-peer overlay networks where nodes need to look up the IP address of target nodes that are referenced by some identifier in a flat address space. An example of such an application would be the P6P IPv6-to-IPv4 overlay network [7]; another possibility is a DNS-over-DHT application that uses a DHT to look up DNS NS-entries, unlike previously proposed DNS-over-DHT schemes that used DHTs to store and retrieve DNS A-entries directly [8], [9]. Our DHT is also likely to be useful for supporting mobile hosts that obtain their IP addresses via DHCP; our DHT can be used in place of mobile IP as the host registration mechanism.

We compare our scheme to Gupta et al.'s one-hop routing scheme [5] and show that we are able to achieve similar levels of lookup performance with comparable amounts of background maintenance traffic, at a somewhat lower complexity by not requiring the network to maintain a fixed event-dissemination hierarchy and without asymmetric bandwidth consumption across nodes within the network. Our main contribution is a novel routing state dissemination algorithm

that broadcasts events efficiently to all nodes using small messages, which we call *tokens*.

In any DHT, nodes must periodically probe their immediate neighbors in the address space in order to ensure routing correctness. If a neighbor is found to have failed, a node must then attempt to repair the address space by contacting the next best replacement for the failed neighbor. Our key insight is that since the nodes in the network already have to exchange *stay-alive* messages periodically, it would be desirable to propagate node join/departure events by piggybacking them on stay-alive messages where possible and forwarding them along the ring. A naive scheme to pass tokens sequentially along the ring takes a long time and is relatively inefficient; so instead, we use a parallelized token-passing scheme.

A DHT with a circular address space can end up loopy after a network partition [10]. Our token-passing algorithm has the additional beneficial side-effect of detecting and automatically fixing global inconsistencies in the address space. The process that maintains and repairs the address space is called *stabilization*. The key idea is simple: to detect a loop in the address space, all we need to do is to traverse the entire ring and make sure that we come back to where we started. Our parallelized token-passing algorithm achieves this effect. Once a loop is detected by the appropriate nodes, the local stabilization protocol kicks in and repairs the inconsistency.

II. OVERVIEW

Like Chord [1], our DHT is organized as a one-dimensional circular address space where each node is assigned a node identifier (*nid*). As shown in Fig. 1, the node responsible for a key is the node whose *id* most closely follows the key, which we also call the *successor*. We use the cryptographic hash function SHA-1 [11] to determine the *nid* of a new node from some identifier (perhaps a hostname or an IP address), so that with high probability, the *nids* do not collide and are uniformly distributed over the entire circular address space.

Each node maintains its routing information in a cache. A node cache is simply an ordered list containing the IP addresses and status of the known nodes in the network, indexed by their *nids*. For efficiency, the list is stored as a B-tree.

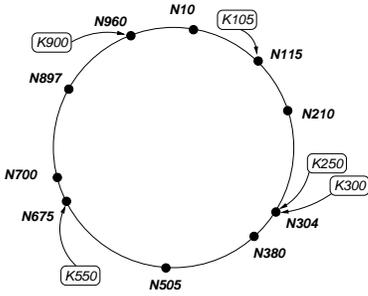


Fig. 1. Circular identifier address space with ten nodes and five keys.

A. Node Joins and Departures

Like all other DHTs, we assume that a node that wishes to join the network will know of at least one other node that is already in the network. The join operation is a two-step procedure: (i) The new node makes a regular iterative lookup for the successor to its nid in the ring using the node(s) that it knows about. (ii) Once the successor is found, the new node sends a request to its successor to import the entries required to initialize its cache. Simultaneously, it also generates a join token with its successor as the final destination node and passes the token anti-clockwise to its predecessor. A join is considered complete once a node finds its successor and completes the cache transfer.

Records of nodes that have recently failed or departed are also transferred in this initial cache transfer. Node failures or departures are detected by timeouts. When a node discovers that its predecessor has failed, it generates a token containing information about the failed node (i.e., a leave token) and passes the token to its new predecessor.

If a node finds that another node in the network already has its chosen nid , it obtains a new nid by adding an index to its original identifier and hashing again. It then attempts to join the network again at this new nid . Since the probability of collisions within the address space is extremely small, it is likely that a node will be able to find a unique nid without any difficulty.

B. Basic Lookup Algorithm

The basic lookup algorithm is straightforward. A node simply contacts the best-known successor in its cache for the queried id . If information on the destination had earlier been propagated successfully to the node performing the lookup, it would find the correct node in one hop. If not, the node contacted would most likely be able to provide a better next hop node. In the worst possible case, we can simply traverse the ring one node at a time to reach the destination node. We are guaranteed to eventually arrive at the correct node as long as the ring is not broken and the address space is consistent and not loopy. An example of a network with a loopy configuration is shown in Fig. 2. A quick note here is that when a node receives a query or reply, it updates the timestamp of the entry in its cache corresponding to the sender (or creates an entry for the sender if one does not already exist). In this way, the

lookup traffic is also used to help update and maintain the network routing state.

C. Distributed Token-Passing Algorithm

A token is a message containing information on network events like node joins and node failures, together with entries for the source and destination nodes. A node entry consists of a node's id and its IP address. The event description contains information on the type of event (i.e., join/leave) and also a timestamp.

When a node receives a token, the receiving node x can choose one of the two following actions:

- x can simply pass the token to its predecessor; or
- it can generate q secondary tokens.

The decision as to whether to generate secondary tokens depends on the policy adopted and the resources available. In general, a node checks its cache to see if it can find $q - 1$ node entries that are approximately uniformly spaced in the remaining segment of the address space to be traversed by the token. If so and if it decides that it has sufficient outgoing bandwidth, the node may choose to generate the q secondary tokens. If not, it simply passes the original token to its predecessor. If all nodes choose to generate q secondary tokens then each token will be propagated to all nodes in $\log_q n$ hops, where n is the network size.

Starting from a token with destination, n_d , a node x generates q secondary tokens as follows:

- node x (with identifier n_x) picks $q - 1$ nodes with identifiers n_1, n_2, \dots, n_{q-1} distributed approximately uniformly in the segment of address space (n_x, n_d) from its cache, where $n_x, n_1, n_2, \dots, n_{q-1}$ are monotonically decreasing in the anti-clockwise direction within the circular id address space.
- x generates a token with destination n_d and sends it to node n_{q-1} .
- x then proceeds to send node n_i a token with n_{i+1} marked as the destination, for $i = q - 2, \dots, 1$, 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 predecessor.

This is illustrated in Fig. 3.

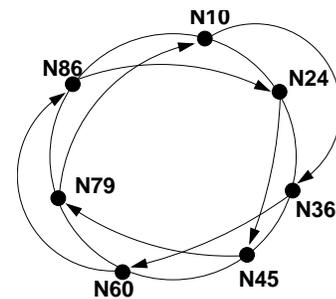


Fig. 2. An example of a loopy address space configuration. The arrows indicate the direction of successor pointers.

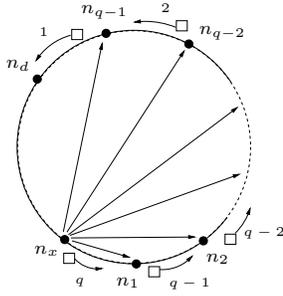


Fig. 3. Generation of q secondary tokens.

A token is destroyed under one of the following circumstances:

- 1) when it reaches a node with an identifier greater or equal to its intended destination;
- 2) if it reaches a node whose predecessor is the specified destination;
- 3) if it reaches a node that has on record an event on the specified node, which has a more recent timestamp, indicating that the event carried by the said token is out-of-date.

D. Merging of Tokens

Whenever a node receives a token, it updates its node cache with all the information contained within the token. Although we could conceivably generate one token for each network event, we can reduce message overhead by sending out tokens only at fixed time intervals and combining tokens that arrive within the same interval. Suppose a node x receives two tokens – (source : n_{s1} , destination : n_{d1} , events : e_1, e_2) and (source : n_{s2} , destination : n_{d2} , events : e_3), such that $n_{d1} < n_{d2}$ within a given time interval. Instead of forwarding two tokens, the node can choose to forward the token (source : n_x , destination : n_{d2} , events : e_1, e_2, e_3) instead. With such a scheme, an event is also stripped from a token if it had already been delivered to a node by an earlier token.

E. Fault Tolerance

With node failures and departures, it is likely that some tokens will be lost if a node receives a token but fails before it can successfully forward it. In such an event, the result is that a consecutive segment of the ring will not be notified of the dropped event. We fix this problem with repair tokens. Repair tokens are like regular tokens except that they do not have a destination identifier and can be forwarded in either direction, i.e. both clockwise and anti-clockwise. Since they have no destination, repair tokens cannot be split into secondary repair tokens like regular tokens. They are forwarded only in one direction until they reach a node that had already been notified of the event that they carry and are thereby destroyed. Repair tokens are generated under the following circumstances:

- 1) **Set Reconciliation.** At a very low frequency, nodes exchange node cache digests with neighboring nodes to reconcile their caches [12], [13], [14]. When a node

obtains a list of new node entries from one neighbor, it will generate a repair token containing all these entries to its other adjacent neighbor.

- 2) **Successful/failed Lookups.** If new nodes are discovered in the process of a lookup, a node will create two repair tokens containing the newly discovered node entries and forward them to its successor and predecessor. Repair tokens containing a leave event are generated instead in the case of a timeout.
- 3) **From being queried.** A node may receive a query from a node that it does not already know about. If so, the queried node also generates two repair tokens to inform its neighbors about the querying node.

Nodes will also keep track of recently received events. If a node discovers that it is receiving extremely frequent join and leave events from a given node, further tokens from the offending node will be dropped for a specified period of time. This is similar to BGP route flap damping [15].

F. Stabilization

Like Chord, the correctness of our lookup algorithm is dependent only on the correctness of the predecessor and successor pointers. When multiple nodes attempt to join the ring at approximately the same location, temporary inconsistencies may arise in the address space.

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 [10]).

1) *Weak Stabilization:* 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 or 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, 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. In addition, each node will also periodically check with its successor to ask for its successor's predecessor. This process is known as *weak stabilization*.

Theorem 1: The weak stabilization protocol will eventually cause our network to converge to a *weakly stable* state¹.

2) *Strong Stabilization:* Although, it is in generally highly improbable that a network will end up loopy (except perhaps after a network partition), it is still desirable for a DHT to have a scheme that will automatically detect and fix global inconsistencies in the address space, if such inconsistencies should arise. Our token-passing algorithm has a nice beneficial

¹We use the same weak stabilization algorithm as Chord. The proof can be found in the Chord Technical Report [10].

side-effect in that it automatically allows us to detect global inconsistencies in the address space.

Theorem 2: The combination of our parallel token-passing algorithm with the weak stabilization protocol will cause our network to converge to a *strongly stable* state within 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 allow at least one pair of nodes to detect an inconsistency; second, whenever such an inconsistency is detected, the weak stabilization algorithm will kick in and fix the inconsistency. To see that the token-passing algorithm will allow at least one pair of nodes to detect an inconsistency, the key is to recognize that the net effect of our parallelized token-passing algorithm is to choose $\frac{n}{q}$ nodes recursively from the set of all nodes and to send messages in one direction along the ring between each pair of consecutive nodes.

III. ANALYSIS

In this section, we estimate the predicted lookup performance and cost of our routing state dissemination algorithm and show that it scales well for networks with up to a million nodes.

We consider a network with a steady state size of n nodes, where n ranges from 2,000 to a million nodes. Saroiu et al. found in a measurement study of the Gnutella and Napster file sharing networks that the median lifespan of peer-to-peer hosts is about 60 minutes [16]. Another recently completed measurement study on the KaZaa filesharing network found that median lifespan of KaZaa hosts is 2.4 minutes while the 90th percentile is 28.25 minutes [17].

We consider a mean node lifetime of 60 minutes in our analysis. For $n = 10,000$, this translates to approximately 5.6 node join/leave events per second; for $n = 1,000,000$, it translates to approximately 560 events per second. In our worst-case analysis, we assume that tokens take one second to complete each hop and that tokens are not merged. These are very conservative assumptions since Internet latencies are in the order of a few hundred milliseconds and it is highly likely that tokens can be merged to save messaging overhead. Also, we assume for simplicity that each token carries only one join/leave event.

A. Expected One-Hop Lookup Performance

Our basic dissemination algorithm makes it extremely likely that a lookup will succeed within one hop since ideally, every node will have a record of all other nodes in its cache. Lookup will usually fail on the first try only if the information on a new node or on a departure has not propagated to the node performing the lookup. Table I shows the required propagation times and their associated estimated worst-case one-hop lookup failure rates for networks with various parameter values for n and q (the number of parallel tokens per hop) and for 3 different scenarios: (i) tokens are never dropped, (ii) tokens are dropped with probability 0.01% and (iii) tokens are dropped

TABLE I
ESTIMATED WORST-CASE PROPAGATION TIMES AND ONE-HOP FAILURE RATES

Network size, n	q	Event Propagation Time (s)	One-hop lookup failure rate		
			No drops	0.01% drop	0.1% drop
2,000	2	11	0.611%	0.720%	1.699%
	3	7	0.389%	0.459%	1.084%
	4	6	0.333%	0.393%	0.943%
10,000	2	17	0.944%	1.113%	2.615%
	3	11	0.611%	0.720%	1.699%
	4	9	0.500%	0.590%	1.392%
100,000	2	20	1.111%	1.309%	3.070%
	3	13	0.722%	0.851%	2.005%
	4	10	0.556%	0.655%	1.546%
1,000,000	2	24	1.333%	1.570%	3.674%
	3	15	0.833%	0.982%	2.310%
	4	12	0.667%	0.786%	1.852%

with probability 0.1%, on each token-passing hop. As tokens are acknowledged, a token is dropped only when a node fails immediately after receiving a token, without the opportunity to pass it on to the next node. Such occurrences are expected to be rare (which is why we consider only small drop probabilities of 0.1% or less).

We would like to highlight that the failure probabilities in Table I are the probabilities that a lookup does not succeed on the **first** try within 25 seconds immediately after a node first joins the network. Even under the worst-case scenario of a 0.1% drop probability on each token-passing hop in a network of a million nodes, the probability that a lookup will return the correct answer within two hops for $q = 2$ is $1 - (3.674\%)^2 = 99.9\%$.

B. Maintenance Bandwidth

In terms of the expected communication costs for our system, we have only two major components: (i) the initial join and cache transfer²; and (ii) join/leave tokens that are propagated to all nodes. A token containing a single event is 98 bytes in size and it consists of the following components:

- 28-byte UDP/IP header;
- Source entry (16 bytes *nid* + 4 bytes IP address);
- Destination entry (16 bytes *nid* + 4 bytes IP address);
- Token type (1 byte);
- Event entry (16 bytes *nid* + 4 bytes IP address); and
- Event description (4 bytes timestamp + 2 byte serial number³ + 3 byte description).

The acknowledgment for a token is expected to be about 30 bytes in size and it consists of the following components:

- 28-byte UDP/IP header;
- Token serial number (2 bytes).

Hence, each token exchange is expected to cost approximately 128 bytes. Based on these numbers, Table II shows

²We amortize the bandwidth costs for transferring an entire cache over a node's lifetime of 60 minutes.

³This serial number is simply a nonce that helps a node to keep track of token acknowledgments.

TABLE II
BACKGROUND BANDWIDTH REQUIRED

Network size, n	Size of initial cache transfer	Background bandwidth required		
		Cache transfer	Join/leave events	Total
2,000	40 kbytes	11.1 bps	1.14 kbps	1.15 kbps
10,000	200 kbytes	55.6 bps	5.69 kbps	5.75 kbps
100,000	2 Mbytes	0.556 kbps	56.9 kbps	57.5 kbps
1,000,000	20 Mbytes	5.56 kbps	569 kbps	575 kbps

TABLE III
REQUIRED BACKGROUND EVENT MAINTENANCE BANDWIDTH

Network size, n	Token-passing scheme	Gupta et al.'s One-Hop scheme [5]		
		Slice Leader	Unit Leader	Other
2,000	1.14 kbps	2.36 kbps	2.56 kbps	1.96 kbps
10,000	5.69 kbps	11.8 kbps	5.12 kbps	3.67 kbps
100,000	56.9 kbps	117 kbps	33.9 kbps	22.9 kbps
1,000,000	569 kbps	789 kbps	322 kbps	215 kbps

the expected background bandwidth costs to support networks of various sizes. Table III shows a comparison of the background event maintenance bandwidth between our token-passing scheme and Gupta et al.'s one-hop scheme, where the number of slices, k , and the number of units per slice, u , is optimized for minimal bandwidth consumption [5]. Even though our scheme imposes a higher average bandwidth per node (at least under the assumption that tokens are not merged), we are able to avoid the upstream bottlenecks associated with slice leaders.

IV. SIMULATION RESULTS

In order to evaluate our algorithm, we obtained the simulation code for the one-hop lookup scheme by Gupta et al.⁴, which was written in the *p2psim* [18] peer-to-peer protocol simulation framework, and compared its performance to the performance of an implementation of our one-hop token-passing DHT. We used a simulation topology with 2,000 nodes in the steady state. The node lifetimes are exponentially-distributed with a mean of 60 minutes. Initially, nodes join the network at a mean rate of 10 nodes per second for 200 s. Subsequently, nodes that depart (after their lifespans run out) will rejoin after an exponentially-distributed interval with a mean of 6 minutes. Routing state is cleared each time that a node leaves and rejoins the network. Gupta et al.'s one-hop scheme takes approximately 40 s after the initial wave of node joins ($t = 200$) to reach a steady state and as shown in Fig. 4, the one-hop steady state lookup failure rate is 0.4%. Our token-passing DHT (with $q = 4$) achieves the same steady state one-hop failure rate somewhat more rapidly. As shown in Fig. 5, the steady state two-hop failure rate for both schemes is less than 0.01% on average.

⁴Gupta et al.'s one-hop scheme divides the nodes in the network into k slices, each with u units. In our simulation, we used the optimal configuration of $k = 17$ slices and $u = 5$ units per slice as computed from the network size and churn rate. Please refer to [5] for the details.

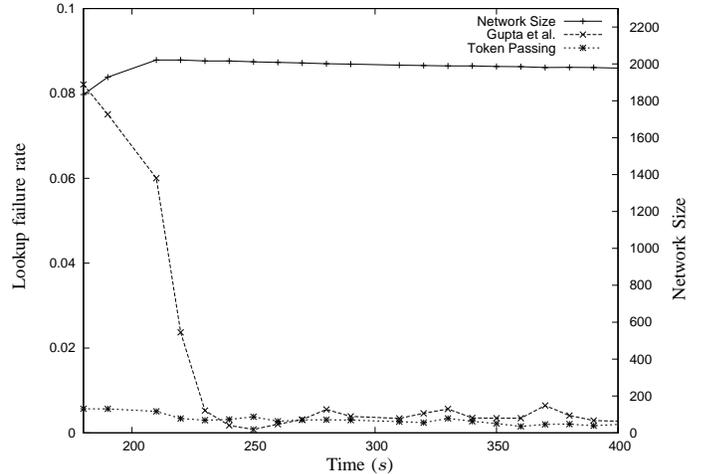


Fig. 4. Comparison of one-hop lookup failure rates between our token-passing scheme and Gupta et al.'s one-hop scheme (start up transients have been cropped for clarity).

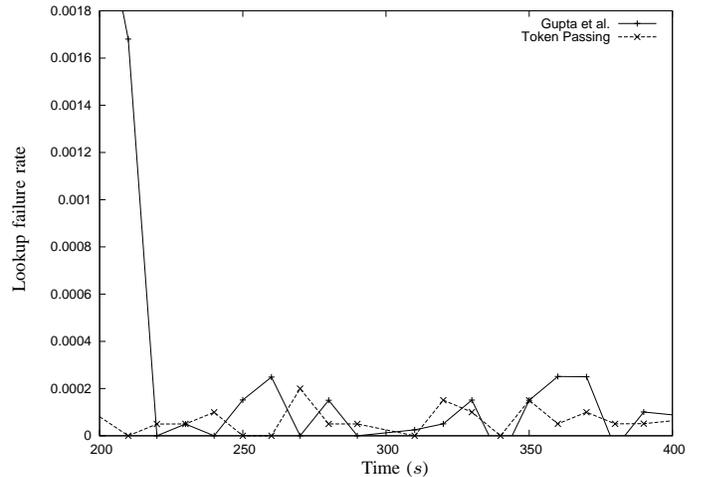


Fig. 5. Comparison of two-hop lookup failure rates between our token-passing scheme and Gupta et al.'s one-hop scheme (start up transients have been cropped for clarity).

The average background maintenance bandwidth for both schemes in the steady state is shown in Fig. 6. In Gupta et al.'s scheme, nodes can be slice leaders, unit leaders or regular nodes. We have explicitly shown the average maintenance bandwidth for slice leaders in Fig. 6 in addition to the average maintenance bandwidth per node. Our scheme (at approximately 3.6 kbps in the steady state) requires somewhat more bandwidth than the 1.15 kbps predicted by our analysis in Table II. The reason for this is that Table II lists only the bandwidth required for disseminating events and does not take into account the periodic weak stabilization/stayalive messages. This stabilization bandwidth is expected to be independent of network size and hence at large network sizes and aggregate event rates, the total maintenance bandwidth will be dominated by event dissemination.

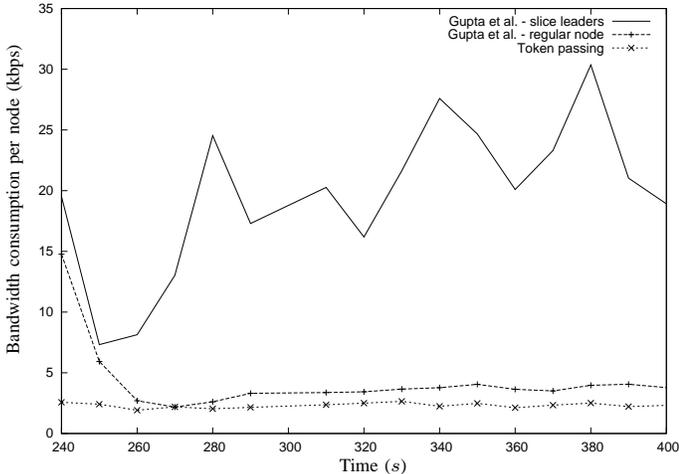


Fig. 6. Comparison of background maintenance bandwidth between our token-passing scheme and Gupta et al.’s one-hop scheme (start up transients have been cropped for clarity).

V. DISCUSSION

Our DHT lookup algorithm is suitable for networks that experience relatively small changes in membership (≤ 100 events globally per second) and that run applications for which it is reasonable to expect some lag in the time from which a node joins the system to when attempts are made to locate it. The former determines the background bandwidth required to maintain the routing state, while the latter provides the system with some time for node join/departure events to propagate to all nodes within the system. At the same time, even if a lookup is made for a node almost immediately after it joins, the access time required to perform the lookup is also expected to be small since the background propagation mechanism keeps most of the caches relatively up-to-date and our ring maintenance protocol guarantees that if a node exists, it will eventually be found.

Our token propagation protocol makes no assumptions on the size of the network and its feasibility depends only on the available bandwidth and churn rate. Since token creation is determined by the node joins/departures (which are uniformly distributed because we use a hash function to determine the mappings from the identifier to the *id* address space), load balancing happens automatically.

Our protocol is also able to accommodate heterogeneity in the available bandwidth among the nodes in a typical network. A node that is required by the protocol to generate secondary tokens may choose not to do so if it does not have sufficient bandwidth available. It may choose simply to forward the token to its predecessor. Similarly, a node that has access to a lot of excess bandwidth may choose to generate more than q secondary tokens.

VI. RELATED WORK

Rodrigues et al. proposed a one-hop DHT lookup scheme for networks with small and somewhat controlled peer dy-

namics like in a corporate environment, so they had to need to handle a large number of simultaneous membership changes efficiently [19]. Doceur et al. also proposed a lookup algorithm with a constant number of hops, assuming smaller peer dynamics and allowing for lossy lookups [20].

Gupta et al.’s one- and two-hop routing schemes use a static dissemination hierarchy to broadcast network events [5]. Configuration parameters like the number of slices and the number of units in a slice must be fixed beforehand with a good idea of the steady-state network size. Compared to our scheme, their system is significantly more complex, has many system parameters to tune and has somewhat asymmetric upstream/downstream bandwidths. Although our scheme is expected in theory to impose a slightly higher event notification communication cost (since each token essentially has to carry individual dissemination information and thus incurs a higher overhead), it turns out in practice, as demonstrated by our simulations, that our token-passing scheme achieves a similar level of performance with a comparable amount of maintenance bandwidth. In addition, our dissemination algorithm is highly bandwidth-adaptive and is expected to scale better because bandwidth consumption is uniformly distributed over all nodes; nodes can also choose to pass tokens without generating and forwarding secondary tokens, thereby bearing less than the average load.

Mizrak et al. also proposed a two-hop hierarchical routing scheme where some nodes are designated as *superpeers* [21]. Kelips [22] is another proposed lookup algorithm that achieves $O(1)$ lookup performance on average by dividing the network into $O(\sqrt{n})$ affinity groups of $O(\sqrt{n})$ nodes and using a gossip mechanism to propagate membership changes. Harchol-Balter et al. [23] proposed an epidemic algorithm (called *Name-Dropper*) for the *Resource Discovery Problem*, which allows all the machines in a weakly connected network to learn about every machine within $O(\log^2 n)$ rounds of message exchanges with high probability.

The proposed strong stabilization algorithm was inspired by our work on EpiChord [24]. To the best of our knowledge, among the other DHTs, only Chord [1] 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 [4], [3].

VII. CONCLUSION AND FUTURE WORK

We have proposed a DHT that stores a large amount of routing information per node in order to achieve one- and two-hop lookup performance. We may not always need or want one-hop lookup performance and we may wish to trade off some lookup performance in order to reduce the background maintenance bandwidth consumption. An interesting feature of our algorithm is that it can be scaled very naturally by pruning the events disseminated and by extending the lookup algorithm to allow for asynchronous parallel lookups [24].

It would be interesting to study how different event pruning schemes will affect lookup performance and explore if there is in fact an optimal scheme. We expect that the performance penalty associated with reducing stored state to be sub-linear with regards to the average lookup path length and latency.

For large networks with about a million nodes, the size of the initial cache transfer can be rather sizable (in the order of 20 Mbytes). It would be interesting to evaluate the effectiveness of a scheme where we do not initiate full cache transfers when a node first joins the network. Instead, their successors will transfer only a fraction (perhaps 10%) of their cache entries uniformly distributed over the entire address space.

We believe that our one-hop DHT is somewhat more scalable and less complex than the one-hop scheme previously proposed by Gupta et al. [5]. We are able to avoid the severe upstream bandwidth bottleneck associated with an event dissemination architecture that uses a fixed hierarchy, by using a parallelized token-passing algorithm that dynamically constructs per-event dissemination trees. Our token-passing algorithm also has a nice beneficial side-effect in that it automatically detects and fixes global inconsistencies in the address space.

ACKNOWLEDGMENTS

The authors wish to thank Anjali Gupta for sharing her *p2psim* simulation code for her one-hop scheme [5].

REFERENCES

- [1] 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.
- [2] 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.
- [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] 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.
- [6] R. Rodrigues and C. Blake, "When multi-hop peer-to-peer routing matters," in *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS '04)*, February 2004.
- [7] L. Zhou and R. van Renesse, "P6P: A peer-to-peer approach to internet infrastructure," in *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS '04)*, San Diego, CA, February 2004.
- [8] R. Cox, A. Muthitacharoen, and R. Morris, "Serving DNS using a peer-to-peer lookup service," in *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, MIT Faculty Club, Cambridge, MA, March 2002.
- [9] V. Ramasubramanian and E. G. Sirer, "Beehive: Exploiting power law query distributions for O(1) lookup performance in peer to peer overlays," in *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI 2004)*, March 2004.
- [10] 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.
- [11] "FIPS 180-1. secure hash standard," US Department of Commerce/NIST, Tech. Rep., April 1995.
- [12] J. Byers, J. Considine, and M. Mitzenmacher, "Informed content delivery across adaptive overlay networks," in *Proceedings of the 2002 ACM SIGCOMM Conference*, August 2002, pp. 47–60.
- [13] —, "Fast approximate reconciliation of set differences," Boston University, Tech. Rep. TR 2002-019, 2002.
- [14] Y. Minsky and A. Trachtenberg, "Scalable set reconciliation," in *Proceedings of the 40th Annual Allerton Conference on Communication, Control, and Computing*, October 2002.
- [15] C. Villamizar, R. Chandra, and R. Govindan, "BGP route flap damping," November 1998.
- [16] 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.
- [17] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, and J. Z. Henry M. Levy, "Measurement, modeling, and analysis of a peer-to-peer file-sharing workload," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '01)*, Bolton Landing, NY, USA, October 2003.
- [18] T. M. Gil, F. Kaashoek, J. Li, R. Morris, and J. Stribling, *p2psim: a simulator for peer-to-peer protocols*. [Online]. Available: <http://www.pdos.lcs.mit.edu/p2psim>
- [19] R. Rodrigues, B. Liskov, and L. Shrira, "The design of a robust peer-to-peer system," in *Proceedings of the 10th ACM SIGOPS European Workshop*, September 2002.
- [20] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer, "Reclaiming space from duplicate files in a serverless distributed file system," in *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*. IEEE Computer Society, 2002, p. 617.
- [21] 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.
- [22] 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.
- [23] M. Harchol-Balter, T. Leighton, and D. Lewin, "Resource discovery in distributed networks," in *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*. ACM Press, 1999, pp. 229–237.
- [24] 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.