Sloppy hashing and self-organizing clusters

Michael J. Freedman and David Mazières
NYU Dept of Computer Science
{mfreed,dm}@cs.nyu.edu

http://www.scs.cs.nyu.edu/coral/

Abstract

We are building Coral, a peer-to-peer content distribution system. Coral creates self-organizing clusters of nodes that fetch information from each other to avoid communicating with more distant or heavily-loaded servers. Coral indexes data, but does not store it. The actual content resides where it is used, such as in nodes' local web caches. Thus, replication happens exactly in proportion to demand.

We present two novel mechanisms that let Coral achieve scalability and high performance. First, a new abstraction called a *distributed sloppy hash table* (DSHT) lets nodes locate nearby copies of a file, regardless of its popularity, without causing hot spots in the indexing infrastructure. Second, based on the DSHT interface, we introduce a decentralized clustering algorithm by which nodes can find each other and form clusters of varying network diameters.

1 Introduction

The academic community has implemented a number of distributed hash tables (DHTs) as efficient, scalable, and robust peer-to-peer infrastructures. However, we should ask whether DHTs are well-suited for the desired applications of the wider Internet population. For example, can DHTs be used to implement file-sharing, by far the most popular peer-to-peer application? Or could DHTs replace proprietary content distribution networks (CDNs), such as Akamai, with a more democratic client caching scheme that speeds up any web site and saves it from flash crowds at no cost to the server operator?

Thus far, the answer to these questions is no. DHTs fail to meet the needs of real peer-to-peer applications for two main reasons.

DHTs provide the wrong abstraction. Suppose many thousands of nodes store a popular music file or cache CNN's widely-accessed home page. How might a hash table help others find the data? Using CNN's URL as a key, one might store a list of every node that has the web page. Of course, any single node responsible for such a URL-to-node-list mapping would quickly be overloaded. DHTs typically replicate popular data, but replication helps only with fetches, not stores. Any node seeking a web page will likely also cache it. Therefore, any URL-to-node-list mapping would be updated almost as frequently as it is fetched.

An alternative approach, taken by CFS [2], OceanStore [3], and PAST [8], is to store actual content in the hash table. This approach wastes both storage and bandwidth, as data must be stored at nodes where it is not needed. Moreover, while users have clearly proven willing to burn bandwidth by sharing files they themselves are interested in, there is less incentive to dedicate bandwidth to sharing unknown data. Worse yet, storing content in a DHT requires large amounts of data to be shifted around when nodes join and leave the system, a common occurrence [9].

DHTs have poor locality. Though some DHTs make an effort to route requests through nodes with low network latency, the last few hops in any lookup request are essentially random. Thus, a node might need to send a query half way around the world to learn that its neighbor is caching a particular web page. This is of particular concern for any peer-to-peer CDN, as the average DHT node may have considerably worse network connectivity than the web server itself.

This paper presents Coral, a peer-to-peer content distribution system we are building. Coral is based on a new abstraction we call a *distributed sloppy hash table* (DSHT). It is currently being built as a layer on the Chord lookup service [12], although it is equally designed to support Kademlia [5] or other existing systems [6, 7, 13]. Coral lets nodes locate and download files from each other by name. Web caches can use it to fetch static data from nearby peers. Users can employ it directly to share directories of files. Coral's two principal goals are to avoid hot spots and to find nearby data without querying distant nodes.

The DSHT abstraction is specifically suited to locating replicated resources. DSHTs sacrifice the consistency of DHTs to support both frequent fetches and frequent stores of the same hash table key. The fundamental observation is that a node doesn't need to know every replicated location of a resource—it only needs a single, valid, nearby copy. Thus, a sloppy insert is akin to an append in which a replica pointer appended to a "full" node spills over to the previous node in the lookup path. A sloppy retrieve only returns some randomized subset of the pointers stored under a given key.

In order to restrict queries to nearby machines, each Coral node is a member of several DSHTs, which we call *clusters*, of increasing network *diameter*. The diameter of a cluster is the maximum desired round-trip time between any two nodes it contains. When data is cached somewhere in a Coral cluster, any member of the cluster can locate a copy without querying machines farther away than the cluster's diameter. Since nodes have the same identifiers in all clusters, even when data is not available in a low-diameter cluster, the routing information returned by the lookup can be used to continue the query in a larger-diameter cluster.

Note that some DHTs replicate data along the last few hops of the lookup path, which increases the availability of popular data and improves performance in the face of many readers. Unfortunately, even with locality-optimized routing, the last few hops of a lookup are precisely the ones that can least be optimized. Thus, without a clustering mechanism, even replication does not avoid the need to query distant nodes. Perhaps more importantly, when storing pointers in a DHT, nothing guarantees that a node storing a pointer is near the node pointed to. In contrast, this property follows naturally from the use of clusters.

Coral's challenge is to organize and manage these clusters in a decentralized manner. As described in the next section, the DSHT interface *itself* is well-suited for locating and evaluating nearby clusters.

2 Design

This section first discusses Coral's DSHT storage layer and its lookup protocols. Second, it describes Coral's technique for forming and managing clusters.

2.1 A sloppy storage layer

A traditional DHT exposes two functions. $put(key, value)$ stores a value at the specified m-bit key, and $get(key)$ returns this stored value, just as in a normal hash table. Only one value can be stored under a key at any given time. DHTs assume that these keys are uniformly distributed in order to balance load among participating nodes. Additionally, DHTs typically replicate popular key/value pairs after multiple *get* requests for the same *key*.

In order to determine where to insert or retrieve a key, an underlying lookup protocol assigns each node an m-bit *nodeid* identifier and supplies an RPC $find\_closer\_node(key)$. A node receiving such an RPC returns, when possible, contact information for another a node whose *nodeid* is closer to the target key. Some systems [5] return a set of such nodes to improve performance; for simplicity, we hereafter refer only to the single node case. By iterating calls to $find\_closer\_node$, we can map a key to some closest node, which in most DHTs will require an expected $O(\log n)$ RPCs. This $O(\log n)$ number of RPCs is also reflected in nodes' routing tables, and thus provides a rough estimate of total network size, which Coral exploits as described later.

DHTs are well-suited for keys with a single writer and multiple readers. Unfortunately, file-sharing and web-caching systems have multiple readers *and* writers. As discussed in the introduction, a plain hash table is the wrong abstraction for such applications.

A DSHT provides a similar interface to a DHT, except that a key may have multiple values: $put(key, value)$ stores a value under key, and $get(key)$ need only return some subset of the values stored. Each node stores only some maximum number of values for a particular key. When the number of values exceeds this maximum, they are spread across multiple nodes. Thus multiple stores on the same key will not overload any one node. In contrast, DHTs replicate the exact same data everywhere; many people storing the same key will all contact the same closest node, even while replicas are pushed back out into the network from this overloaded node.

More concretely, Coral manages values as follows. When a node stores data locally, it inserts a pointer to that data into the DSHT by executing $put(key, nodeaddr)$. For example, the key in a distributed web cache would be $hash(URL)$. The inserting node calls $find\_closer\_node(key)$ until it locates the first node whose list stored under key is full, or it reaches the node closest to key. If this located node is full, we backtrack one hop on the lookup path. This target node appends $nodeaddr$ with a timestamp to the (possibly new) list stored under key. We expect records to expire quickly enough to keep the fraction of stale pointers below 50%.

A $get(key)$ operation traverses the identifier space and, upon hitting a node storing key, returns the key's corresponding contact list. Then, the requesting node can contact these nodes, in parallel or in some application-specific way, to download the stored data.

Coral's "sloppy store" method inserts pointers along the lookup path for popular keys. Its practice of "spilling-over" when full helps to balance load while inserting pointers, retrieving pointers, and downloading data. Rapid membership changes remain inexpensive as the system only exchanges pointers.

While sloppy stores eliminate hot spots, we still must address the problem of latency. In particular, $\mathit{find\_closer\_node}(key)$ may circle the globe to find some nearby host with the data. To take advantage of data locality, Coral introduces *hierarchical* lookup.

2.2 A hierarchical lookup layer

Instead of one global lookup system as in [2, 3, 8], Coral uses several *levels* of DSHTs called clusters. Coral nodes belong to one DSHT at each level; the current implementation has a three-level DSHT hierarchy. The goal is to establish many fast clusters with regional coverage (we refer to such "low-level" clusters as level-2), multiple clusters with continental coverage (referred to as "higher" level-1 clusters), and one planet-wide cluster (level-0). Reasonable round-trip time thresholds are 30 msec for level-2 clusters, 100 msec for level-1, and ∞ for the global level-0. Section 3 presents some experimental measurements to support these choices. Each cluster is named by an m-bit cluster identifier, cid_i; the global cid_0 is predefined as 0^m.

Coral uses this hierarchy for distance-optimized lookup, visualized in Figure 1 for both the Chord [12] and Kademlia [5] routing structures.

To **insert** a key/value pair, a node performs a put on all levels of its clusters. This practice results in a loose hierarchical data cache, whereby a higher-level cluster contains nearly all data stored in the lower-level clusters to which its members also belong.

To **retrieve** a key, a requesting node r first performs a get on its level-2 cluster to try to take advantage of network locality. $\mathit{find\_closer\_node}$ on this level may hit some node caching the key and halt (a *hit*). If not, the lookup will reach the node in that cluster closest to the target key, call it t_2. r then continues its search in its level-1 cluster. However, t_2 has already returned routing information in the level-1 cluster. Thus, r begins with the closest level-1 node in t_2's routing table. Even

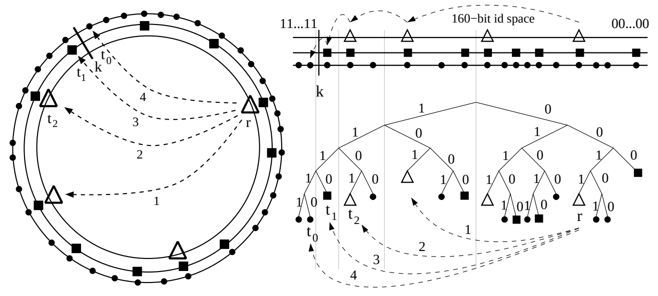

Figure 1: Coral's hierarchical lookup visualized on the Chord (left) and Kademlia (right) routing structures. Nodes maintain the same id in each of their clusters; smaller-diameter low-level lusters are naturally sparser. For a lookup on key k, a node first searches on its lowest cluster. This lookup fails on that level if the node closest to k, node t_2, does not store the key. If this occurs, Coral continues its lookup on a higher-level cluster, having already traversed the id space up to t_2's prefix. Route RPCs are shown with sequential numbering.

if the search eventually switches to the global cluster, Coral does not require any more RPCs than a single-level lookup service, as a lookup always restarts where it left off in the id space. Moreover, Coral *guarantees* that all lookups at the beginning are fast. This functionality arises naturally from a node having the same $nodeid$ in all DSHTs to which it belongs. Note that Coral achieves this property independent of any distance optimization in its underlying lookup protocol.

Two conflicting criteria impact the effectiveness of Coral's hierarchical DSHTs. First, clusters should be large in terms of membership. The more peers in a DSHT, the greater its capacity and the lower the *miss* rate. Second, clusters should have small network diameter to achieve fast lookup. That is, the expected latency between randomly-selected peers within a cluster should be below the cluster's specified threshold.

The remainder of this section describes Coral's mechanisms for managing its multiple DSHT clusters. These mechanisms are summarized in Table 1.

2.3 Joining a cluster

Coral largely inherits its join and leave protocols from its underlying lookup service, with one difference. Namely, a node will only join an *acceptable* cluster, that is, one in which the latency to 90% of the nodes is below the cluster's diameter. This property is easy for a node to test by collecting round trip times to some subset of

The Task	Coral's Solution
Discovering and joining a low-level cluster, while only requiring knowledge of *some* other node, not necessarily a close one.	Coral nodes insert their own contact information and Internet topology hints into higher-level clusters. Nodes reply to unexpected requests with their cluster information. Sloppiness in the DSHT infrastructure prevents hotspots from forming when nodes search for new clusters and test random subsets of nodes for acceptable RTT thresholds. Hotspots would otherwise distort RTT measurements and reduce scalability.
Merging close clusters into the same namespace without experiencing oscillatory behavior between the merging clusters.	Coral's use of cluster size and age information ensures a clear, stable direction of flow between merging clusters. Merging may be initiated as the byproduct of a lookup to a node that has switched clusters.
Splitting slow clusters into disjoint subsets, in a manner that results in an acceptable and stable partitioning without causing hotspots.	Coral's definition of a cluster center provides a stable point about which to separate nodes. DSHT sloppiness prevents hotspots while a node determines its relative distance to this known point.

Table 1: Overview of the Coral's design for self-organizing clusters

nodes in the cluster, perhaps by simply looking up its own identifier as a natural part of joining.

As in any peer-to-peer system, a node must initially learn about some other Coral node to join the system. However, Coral adds a RTT requirement for a node's lower-level clusters. A node unable to find an acceptable cluster creates a new one with a random cid. A node can join a better cluster whenever it learns of one.

Several mechanisms could have been used to discover clusters, including using IP multicast or merely waiting for nodes to learn about clusters as a side effect of normal lookups. However, Coral exploits the DSHT interface to let nodes find nearby clusters. Upon joining a low-level cluster, a node inserts itself into its higher-level clusters, keyed under the IP addresses of its gateway routers, discovered by `traceroute`. For each of the first five routers returned, it executes $put(hash(router.ip), nodeaddr)$. A new node, searching for a low-level acceptable cluster, can perform a get on each of its own gateway routers to learn some set of topologically-close nodes.

2.4 Merging clusters

While a small cluster diameter provides fast lookup, a large cluster capacity increases the hit rate in a lower-level DSHT. Therefore, Coral's join mechanism for individual nodes automatically results in close clusters merging if nodes in both clusters would find either acceptable. This merge happens in a totally decentralized way, without any expensive agreement or leader-election protocol. When a node knows of two acceptable clusters at a given level, it will join the larger one.

When a node switches clusters, it still remains in the routing tables of nodes in its old cluster. Old neighbors will still contact it; the node replies to level-i requests originating outside its current cluster with the tuple$\{cid_i, size_i, ctime_i\}$, where $size_i$ is the estimated number of nodes in the cluster, and $ctime_i$ is the cluster's creation time. Thus, nodes from the old cluster will learn of this new cluster that has more nodes and the same diameter. This produces an avalanche effect as more and more nodes switch to the larger cluster.

Unfortunately, Coral can only count on a rough *approximation* of cluster size. If nearby clusters A and B are of similar sizes, inaccurate estimations could in the worst case cause oscillations as nodes flow back-and-forth. To perturb such oscillations into a stable state, Coral employs a preference function δ that shifts every hour. A node selects the larger cluster only if the following holds:

$$\Big|\log(size_A) - \log(size_B)\Big| \;>\; \delta\left(\min(age_A, age_B)\right)$$

where age is the current time minus $ctime$. Otherwise, a node simply selects the cluster with the lower cid.

We use a square wave function for δ that takes a value 0 on an even number of hours and 2 on an odd number. For clusters of disproportionate size, the selection function immediately favors the larger cluster. However, should clusters of similar size continuously exchange members when δ is zero, as soon as δ transitions, nodes will all flow to the cluster with the lower cid. Should the clusters oscillate when $\delta = 2$, the one 2^2-times larger will get all members when δ returns to zero.

2.5 Splitting clusters

In order to remain acceptable to its nodes, a cluster may eventually need to split. This event may result from a network partition or from population over-expansion, as new nodes may push the RTT threshold. Coral's split operation again incorporates some preferred direction of flow. If nodes merely atomized and randomly re-merged into larger clusters, the procedure could take too long to stabilize or else form highly sub-optimal clusters.

To provide a direction of flow, Coral specifies some node c within cid as a *cluster center*. When splitting, all nodes near to this center c join one cluster; all nodes far from c join a second cluster. Specifically, define $cid^N = hash(cid)$ and let cid^F be cid^N with the high-order bit flipped. The cluster center c is the node closest to key cid^N in the DSHT. However, nodes cannot merely ping the cluster center directly, as this would overload c, distorting RTT measurements.

To avoid this overload problem, Coral again leverages its sloppy replication. If a node detects that its cluster is no longer acceptable, it performs a get first on cid^N, then on cid^F. For one of the first nodes to split, $get(cid^N)$ resolves directly to the cluster center c. The node joins cid_i based on its RTT with the center, and it performs a $put(cid_i, nodeaddr)$ on its old cluster and its higher-level DSHTs.

One concern is that an early-adopter may move into a small successor cluster. However, before it left its previous level-i cluster, the latency within this cluster was approaching that of the larger level-$(i-1)$ cluster. Thus, the node actually gains little benefit from maintaining membership in the smaller lower-level cluster.

As more nodes transition, their gets begin to hit the sloppy replicas of cid^N and cid^F: They learn a random subset of the nodes already split off into the two new clusters. Any node that finds cluster cid^N acceptable will join it, without having needed to ping the old cluster center. Nodes that do not find cid^N acceptable will attempt to join cluster cid^F. However, cluster cid^F could be even worse than the previous cluster, in which case it will split again. Except in the case of pathological network topologies, a small number of splits should suffice to reach a stable state. (Otherwise, after some maximum number of unsuccessful splits, a node could simply form a new cluster with a random ID as before.)

3 Measurements

Coral assigns system-wide RTT thresholds to the different levels of clusters. If nodes otherwise choose their own "acceptability" levels, clusters would experience greater instability as individual thresholds differ. Also, a cluster would not experience a distinct merging or splitting period that helps to return it to an acceptable, stable state. Can we find sensible system-wide parameters?

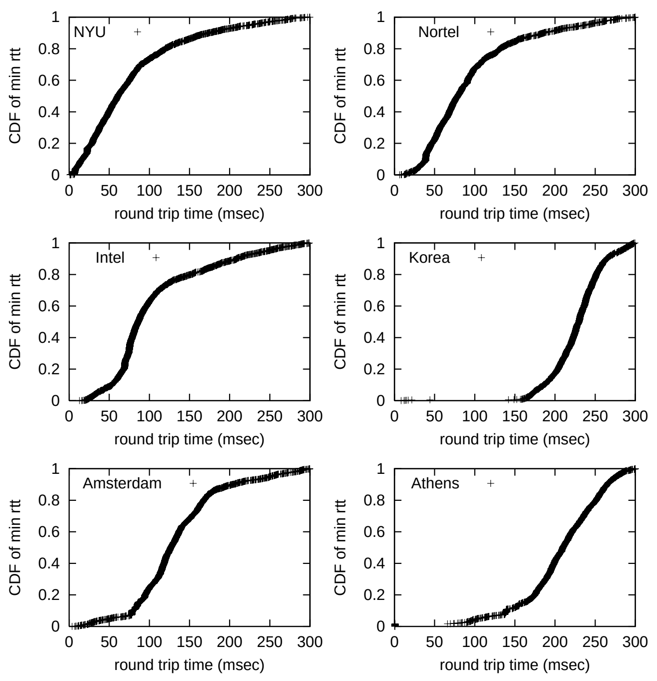

Figure 2: CDFs of round-trip times between specified RON nodes and Gnutella peers.

To measure network distances in a deployed system, we performed latency experiments on the Gnutella network. We collected host addresses while acting as a Gnutella peer, and then measured the RTT between 12 RON nodes and approximately 2000 of these Gnutella peers. Both operations lasted for 24 hours. We determined round-trip times by attempting to open several TCP connections to high ports and measuring the minimum time elapsed between the SYN and RST packets.

Figure 2 shows the cumulative distribution function (CDF) of the measured RTT's between Gnutella hosts and the following RON sites: New York University (NYU); Nortel Networks, Montreal (Nortel); Intel research, Berkeley (Intel); KAIST, Daejon (South Korea); Vrije University (Amsterdam); and NTUA (Athens).

If the CDFs had multiple "plateaus" at *different* RTT's, system-wide thresholds would not be ideal. A threshold chosen to fall within the plateau of some set of nodes sets the cluster's most natural size. However, this threshold could bisect the rising edge of other nodes' CDFs and yield greater instability for them.

Instead, our measurements show that the CDF curves are rather smooth. Therefore, we have relative freedom in setting cluster thresholds to ensure that each level of cluster in a particular region can capture some expected percentages of nearby nodes.

Our choice of 30 msec for level-2 covers smaller clusters of nodes, while the level-1 threshold of 100 msec spans continents. For example, the expected RTT between New York and Berkeley is 68 msec, and 72 msec between Amsterdam and Athens. The curves in Figure 2 suggest that most Gnutella peers reside in North America. Thus, low-level clusters are especially useful for sparse regions like Korea, where most queries of a traditional peer-to-peer system would go to North America.

4 Related work

Several projects have recently considered peer-to-peer systems for web traffic. Stading *et. al.* [10] uses a DHT to cache replicas, and PROOFS [11] uses a randomized overlay to distribute popular content. However, both systems focus on mitigating flash crowds, not on normal web caching. Therefore, they accept higher lookup costs to prevent hot spots. Squirrel [4] proposed web caching on a traditional DHT, although only for LANs. It examines storing pointers in the DHT, yet reports poor load-balancing. We attribute this result to the limited number of pointers stored (only 4), which perhaps is due to the lack of any sloppiness in the system's DHT interface. SCAN [1] examined replication policies for data disseminated through a multicast tree from a DHT deployed at ISPs.

5 Conclusions

Coral introduces the following techniques to enable distance-optimized object lookup and retrieval. First, Coral provides a DSHT abstraction. Instead of storing actual data, the system stores weakly-consistent lists of pointers that index nodes at which the data resides. Second, Coral assigns round-trip-time thresholds to clusters to bound cluster diameter and ensure fast lookups. Third, Coral nodes maintain the same identifier in all clusters. Thus, even when a low-diameter lookup fails, Coral uses the returned routing information to continue the query efficiently in a larger-diameter cluster. Finally, Coral provides an algorithm for self-organizing merging and splitting to ensure acceptable cluster diameters.

Coral is a promising design for performance-driven applications. We are in the process of building Coral and planning network-wide measurements to examine the effectiveness of its hierarchical DSHT design.

Acknowledgments

We thank David Andersen for access to the RON testbed, and Vijay Karamcheti, Eric Freudenthal, Robert Grimm, Sameer Ajmani, and Rodrigo Rodrigues for helpful comments. This research was conducted as part of the IRIS project (`http://project-iris.net/`), supported by the NSF under Cooperative Agreement No. ANI-0225660. Michael Freedman was supported by the ONR under an NDSEG Fellowship. This paper is hereby placed in the public domain.

References

[1] Yan Chen, Randy H. Katz, and John D. Kubiatowicz. SCAN: A dynamic, scalable, and efficient content distribution network. In *Proceedings of the International Conference on Pervasive Computing*, Zurich, Switzerland, August 2002.

[2] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, 2001.

[3] John Kubiatowicz *et. al.* OceanStore: An architecture for global-scale persistent storage. In *Proc. ASPLOS*, Cambridge, MA, Nov 2000.

[4] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *Principles of Distributed Computing (PODC)*, Monterey, CA, July 2002.

[5] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS02)*, Cambridge, MA, March 2002.

[6] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, San Diego, CA, Aug 2001.

[7] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, November 2001.

[8] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, 2001.

[9] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. In *Proc. ACM SIGCOMM Internet Measurement Workshop*, Marseille, France, November 2002.

[10] Tyron Stading, Petros Maniatis, and Mary Baker. Peer-to-peer caching schemes to address flash crowds. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS02)*, Cambridge, MA, March 2002.

[11] Angelos Stavrou, Dan Rubenstein, and Sambit Sahu. A lightweight, robust p2p system to handle flash crowds. In *IEEE International Conference on Network Protocol (ICNP)*, Paris, France, November 2002.

[12] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *IEEE/ACM Trans. on Networking*, 2002.

[13] Ben Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U.C. Berkeley, April 2000.