Democratizing content publication with Coral

Michael J. Freedman, Eric Freudenthal, David Mazières
New York University
{mfreed,freudent,dm}@cs.nyu.edu

Draft – Do not distribute
December 18, 2003

Abstract

Coral is peer-to-peer content distribution network that allows a user to run a web site that offers high performance and meets huge demand, all for the price of a $50/month cable modem. Coral takes advantage of users’ demonstrated willingness to re-distribute content they themselves consider useful. Sites that volunteer to run Coral automatically replicate content as a side effect of users accessing it. Publishing through Coral is as simple as prepending a pseudo-hostname to objects’ URLs; a peer-to-peer DNS layer transparently redirects browsers to participating cache nodes, which in turn cooperate to minimize load on the source web server. One of Coral’s key goals is to avoid ever creating hot spots that might dissuade volunteers from running the software for fear of load spikes. It achieves this through a novel indexing abstraction we introduce called a distributed sloppy hash table, or DSHT.

1 Introduction

The availability of content on the Internet is more-or-less a direct function of the cost shouldered by the publisher. A well-funded web site can reach huge numbers of people though some combination of load-balanced servers, fast network connections, collocated machine rooms, and commercial content distribution networks (CDNs). Publishers who cannot afford such amenities, however, are much more limited in the size of audience and type of content they can serve. Moreover, their sites risk sudden overload following publicity, a phenomenon named the “slashdot” effect, after a popular web site that periodically links to under-provisioned servers, driving unsustainable levels of traffic to them. Because of such problems, even struggling content providers are often forced to expend significant resources on content distribution.

Fortunately, at least with static content, there is an easy way for popular data to reach many more people than the publisher can afford to serve himself or herself—volunteers can step forward to re-publish or mirror the data on their own servers and networks. Indeed, the Internet has a long history of organizations with good connectivity mirroring data they consider to be of value to the community. More recently, the peer-to-peer file sharing phenomenon has demonstrated the willingness of even individual broadband users to dedicate upstream bandwidth to redistributing content the users themselves enjoy.

This paper describes Coral, a system that effectively automates the volunteer mirroring process. Willing participants mirror web content as a simple side effect of downloading it. Clients get transparently redirected to nearby mirrors. As a result, the availability of static web content becomes proportional to the content’s popularity, rather than to the publisher’s cost of distributing it.

Coral is a CDN with a completely decentralized architecture. Anyone can participate in content distribution by running the software and seeding it with the identity of an existing node. To take advantage of Coral, a publisher simply prepends the pseudo-hostname http.nyud.net to the real URL of the published object. A peer-to-peer DNS layer then redirects browsers to nearby Coral HTTP servers, which in turn attempt to fetch objects from each other whenever possible so as to minimize the load on the publisher’s source web server.

Coral is built on a novel distributed lookup abstraction we call a distributed sloppy hash table, or DSHT. DSHTs simply map keys to values, but do so in a way that avoids hot spots when large numbers of (key, value) pairs have the same key. Even if every node of a DSHT repeatedly stores the same key, the rate of store requests at the most heavily-loaded node is only logarithmic in the total number of nodes. When nodes retrieve a key from a DSHT, they receive only a subset of the values stored. DSHTs are well-suited to indexing the locations of replicated objects, because while many replicas may need to be indexed, those seeking objects really only need to find one good replica.

Coral additionally achieves good latency and returns desirably-located replicas by restricting queries to nearby machines whenever possible. Each Coral node is actually a member of several DSHTs, which we call clusters. While one can imagine a variety of clustering criteria, Coral currently clusters by network round trip time. We define the diameter of a cluster to be the maximum de-
sired 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. Because nodes have the same routing identifiers in all clusters, even when data is unavailable in a low-diameter cluster, the information gathered during a failed low-diameter lookup can be used to shorten the lookup path in a larger-diameter cluster.

One of Coral’s challenges is to organize and manage these clusters in a decentralized manner. It does so in part by exploiting the DSHT interface to store network topology information nodes can use as clustering hints. The DNS layer also exploits these hints, along with Coral’s clustering and a light-weight probing mechanism, to chose appropriately located HTTP servers in response to client DNS requests.

The remainder of this paper is structured as follows. Section 2 describes related work and their relation to Coral. In Section 3, we provide a high-level description of Coral’s system structure, the underlying DSHT layer, and the clustering algorithms. We describe the DNS system and web caching applications in Section 4. We describe Coral’s implementation in Section 5, present experimental results in Section 6, and then conclude.

2 Related work

Coral builds on a wide range of previous work in peer-to-peer systems, web caching, and content delivery.

2.1 Structured routing overlays

Structured peer-to-peer overlays create a special graph structure among peers that enables efficient key-based routing. Nodes are typically assigned uniformly distributed node IDs in these systems. Moreover, given an ID value, a peer can locate the node closest to that ID in some small expected number of RPCs (e.g., $O(\log n)$ for $n$ total nodes in the system). Some maintenance protocol is generally required to cope with nodes dynamically joining and dropping off the network. Examples of structured overlays include CAN [21], Chord [27], Kademia [17], Pastry [22], and Tapestry [33]. Coral borrows most directly from Kademlia, which defines the distance between two IDs as their bitwise exclusive or.

2.2 DHTs and directory services

A distributed hash table (DHT) exposes two basic functions to the application: $put(key, value)$ stores a value at the specified $m$-bit key, and $get(key)$ returns this stored value, just as in a normal hash table. Most DHTs use a key-based routing layer, as described above, and store keys on the node whose ID is closest to the key. This means that keys must be well-distributed to balance load amongst nodes. DHTs often replicate multiply-fetched key/value pairs for scalability. One common replication technique is for peers to copy a key onto the second-to-last peer they contacted as part of a $get$ request. Such on-demand replication makes it difficult to store multiple values under the same key.

DHTs can act either as actual data stores or merely as directory services storing pointers. CFS [4] and PAST [23] take the former approach in order to build a distributed file system, and thus require true read/write consistency among operations. Using the network only as a directory or rendezvous service, Tapestry’s DOLR [34, 5] and Coral relax the consistency of storage in the network. To publish a key, Tapestry recursively routes along fast hops between peers, placing at each hop a pointer back to the sending node, until it reaches the node closest to the key. Nearby nodes routing to the same key are likely to follow similar paths and discover these cached pointers to nearby nodes.

2.3 Clustering and measurement systems

A separate body of work focuses on accurate Internet distance estimation, including triangulation [9], IDMaps [7], and dynamic distance maps [30]. These schemes seek to make precise measurements and often require centralized knowledge, although some recent work has approached the problem using peer-to-peer techniques [19]. We note that these papers conclude that organizational classification on the Internet (such as AS number or DNS name) should not be used to estimate topological distance. SkipNet [10] similarly builds a hierarchy of lookup groups, although they explicitly divide the key-space by DNS name to support organizational disconnect.

Akamai [1] and other commercial CDNs have built detailed, accurate maps of the Internet through a combination of routing information from BGP feeds and their own network measurements. These maps allow them to direct clients to nearby content distribution nodes. Similarly, RocketFuel [24] performs traceroutes from numerous vantage points to build accurate topological maps. Coral’s use of traceroute-like probing of DNS clients is a similar mechanism. Coral is somewhat constrained in what it can do compared to the best known mapping techniques. In particular, Coral nodes do not have BGP feeds and are under tight latency constraints to avoid delaying DNS replies while probing. Nonetheless, Coral’s open architecture has the advantage that if many people adopt the system, it will build up a rich database of neighboring networks from many vantage points.
2.4 Web caching systems

Web caching systems fit within a large class of content distribution networks that handle high demand through diverse replication. This section reviews some distributed systems for cooperative web caching.

Prior to the recent interest in peer-to-peer systems, several projects have proposed cooperative Web caching [2, 6, 8, 16, 14]. However, these systems either multicast queries or require that caches know some or all other servers, impacting their scalability, fault-tolerance, and susceptibility to hot spots. While cooperative web caching has client-performance benefits only within limited population bounds [32], perhaps questioning the need for largely-scalable systems, this result does not apply to the server-side benefit that results in increased system throughput.

Several projects have considered peer-to-peer systems for web caching. Stading et al. [25] uses a DHT to cache replicas, and PROOFS [26] uses a randomized overlay to distribute popular content. However, both systems focus solely on mitigating flash crowds, and therefore accept high request latency. Squirrel [11] proposed web caching on a traditional DHT, although only for organization-wide networks. Squirrel reported poor load-balancing when the system stored pointers in the DHT. We attribute this to the DHT’s inability to handle to many values for the same key—Squirrel only stored 4 pointers per object in the DHT. Coral’s DSHT, by contrast, can efficiently store as many pointers to a particular object as there are nodes in the system. SCAN [3] examined replication policies for data disseminated through a multicast tree from a DHT deployed at ISPs.

Akamai [1] and other commercial CDNs use DNS redirection to reroute client requests to local clusters of machines, where hashing schemes [13, 29] map requests to specific machines to increase capacity. These systems require deploying large numbers of highly-provisioned servers, and typically result in very good performance (both latency and throughput) for customers. In comparison, Coral offers less storage capacity, but can provide better performance for organizations hosting Coral nodes, as the Coral nodes can be closer to browsers (for instance on the local side of a bottleneck link to the outside world).

More recently, the CoDeeN [20] content distribution network has provided users with a set of open web proxies. Users can reconfigure their browsers to use CoDeeN as a proxy, and subsequently enjoy better web performance. The system has been deployed, and anecdotal evidence suggests it has been very successful at distributing content efficiently. Earlier simulation results [31] show that certain policies should indeed achieve high system throughput and low request latency. (The specific details of the deployed system have not yet been published.)

While CoDeeN gives a well-defined group of users better web performance to all (or at least most) web sites, Coral’s goal is in contrast to improve performance for all users accessing a well-defined set of web sites—namely those whose publishers have “Coralized” the URLs. The two design points pose somewhat different challenges. For instance, Coral takes pains to minimize the load on under-provisioned source servers, and therefore must tackle the problem of peer-to-peer DNS redirection. CoDeeN has tighter latency requirements, however, because it is on the critical path for all web sites, not just those that would otherwise be overloaded. Finally, we note that CoDeeN has suffered a number of administrative headaches, due in large part to providing malicious users with open proxies. Many of these problems do not apply to Coral, because Coral’s HTTP server is not a full-fledged web proxy. In particular, Coral does not allow POST operations or SSL tunneling. It can also be barred from accessing particular web sites without affecting users’ browsing experience (since users continue to access non-Coralized URLs normally).

3 Coral’s Indexing Infrastructure

Coral is comprised of three main parts: (1) a network of HTTP servers that handle users’ requests, (2) a network of DNS nameservers that map clients to nearby HTTP servers, and (3) the underlying DSHT indexing infrastructure and clustering machinery on which the first two applications are built. This section presents the DSHT and clustering infrastructure, while the next section describes the two applications that yield a peer-to-peer CDN suitable for immediate use.

Coral’s DSHT layers share many desirable properties with most structured peer-to-peer overlays, including full decentralization, efficient routing, and scalability. Additionally, its routing operations within clusters enjoy strong locality—both for finding pointers and for retrieving data referenced by these pointers—and load-balance operations within both the indexing infrastructure and between application-level components.

The DSHT abstraction is designed primarily for environments in which the access pattern to the distributed index is characterized by both multiple readers and multiple writers, yet read/write consistency between operations is not necessary. For example, when indexing cached web content, all Coral HTTP servers cache content once they have downloaded it, and can redistribute that content to other Coral HTTP servers that need it. Thus, HTTP servers must insert their own address under the key of each web page they download. We therefore expect as many puts as gets on the key of each web page. While

---

1This design naturally yields the local surrogate [28] model of data placement optimization.
the puts must be efficient, it is okay to be inconsistent under high load, as the high load precisely means there are plenty of replicas to chose from for that key.

In this section, we first describe Coral’s peer-to-peer key-based routing layer, then describe the implementation of Coral’s basic put and get operations with a DSHT, the changes necessary to enable this operations within a hierarchical set of DSHTs, and Coral’s clustering mechanisms.

### 3.1 Key-Based Routing

Coral’s key-based routing layer is based on Kademlia-style routing tables, in which the distance between two IDs is their bitwise exclusive or, interpreted as a number. This XOR metric is symmetric and offers a single-phase lookup algorithm to find a specified key in the network in $O(\log n)$ hops. All keys in the system, whether identifying nodes, clusters, or pointer values, are opaque, 160-bit numbers. Nodes’ routing table data structures can be visualized as a binary tree whose leaves are known as “k-buckets”; each k-bucket contains nodes with some common node ID prefix corresponding to the bucket’s position in the tree. A node’s ID is formed by taking the (SHA1) hash of the host’s IP address and port.

Kademlia’s routing structure is self-maintaining, i.e., routing operations help refresh the k-buckets by updating node state as a usual side-effect. Instead of each iteration of find key returning a single node closer to key, it returns k such nodes, improving lookup latency and minimizing the need for special maintenance traffic. Still, Coral refreshes its k-buckets explicitly when no lookups have occurred to their range a given keep-alive epoch. Additionally, information sharing during requests includes backpointers to enable more efficient keep-alive techniques [35].

Liberal result caching within each bucket enables lookups to span fewer than $\log(n)$ hops if desired. Additionally, the routing structures support concurrent lookups, such that $\alpha$ RPCs are kept outstanding during a lookup, ensuring that both (1) the lookup progresses through the key-space via nodes with lower round trip time, and (2) stale references in each k-bucket can be removed lazily and yet rarely impact the latency of individual lookups.

Coral’s hierarchical DSHT operations, which we describe later, requires some changes to the basic routing algorithms of Kademlia. The headers of every packet within Coral include the sender’s cluster information: the identifier, age, policy, and known logsize of each of its non-global clusters. The recipient uses this information to demultiplex requests properly, i.e., a recipient should only consider a put and get for those levels on which it shares a cluster with the sender. Additionally, this information drives routing table management: (1) nodes are added or removed from the local cluster-specific routing tables accordingly; (2) cluster information is accumulated to drive cluster management (see Section 3.4).

To avoid hotspots and randomize the results of get operations on popular keys, Coral explicitly limits its use of result caching during routing lookups. Each successive iteration of the routing algorithm attempts to find a node whose common prefix with the target key is $b$ bits longer than in the last iteration.

More concretely, a requesting node $r$ begins routing by setting $targ ← nodeid$ (its own node ID). It then determines the length of the common prefix between $targ$ and $key$ and rounds this value down to a multiple of $b$, call the resulting value $t$. $r$ then sets $targ$ to the id defined by the $(t+b)$-bit prefix of $key$ and $(160-t-b)$-bit suffix of $targ$. It then routes to the $\alpha$ closest nodes $targ$ it knows about (asking them to return nodes near the next value of $targ$). As intermediate nodes respond, node $r$ computes a new $targ$ and repeats the routing process until it finds the desired node or data, or else detects overload on the key.

### 3.2 Sloppy Storage

The DSHT interface is designed for applications that have multiple readers and multiple writers, yet don’t require consistency. This design space differs significantly from the P2P applications such as file systems, which require that values stored in DHTs always be retrievable later. In Coral, stored values are intended to serve merely as pointers, e.g., to map real URLs to one or several of the myriad of Coral HTTP servers caching the content.

Coral’s distributed sloppy hash table (DSHT) exposes a simple client interface, which includes the following:

- **put(key, ptr, ttl, ?levels)**: Inserts a mapping from the random $m$-bit key to some arbitrary value into the system, specifying the time-of-live (TTL) of the reference. This insertion may occur at some subset of cluster hierarchy based on the start and stop levels specified.

- **get(key, ?levels)**: Retrieves some subset of the values stored under a key. Some subset of the cluster hierarchy may optionally be specified.

- **nodes(level, count, ?target, ?services)**: Returns count number of the node’s neighbors from its cluster of specified level. If a target node’s address is included, attempts to find nodes specifically near the target, as opposed to just within the cluster. If services is specified, only returns nodes that are running the particular service (e.g., the Coral HTTP or DNS server).
the same node “assigned” the key, *i.e.*, that node with a nodeid closest to key in some lexicographical ordering. (We hereafter refer to this node as the key’s target.)

In contrast, each Coral node monitors load on its keys to minimize excessive numbers of requests from hitting the key’s target, yet still allow “enough” requests to propagate to keep values at the target fresh. Thus multiple stores of the same key will not overload any one node. Moreover, the pointers returned to the application are well-distributed among those stored, providing load balancing both within Coral and between application servers using Coral. For demonstrative purposes, we first present Coral’s abstraction on a single cluster.

To *insert* a value, Coral performs a two-phase operation. First, it traverses the key-space to find the closest node to the target that potentially will store the value, then it backtracks until the insertion succeeds.

Thus, in the first phase, Coral’s key-based routing layer makes repeated find_node(key) calls to contact nodes successively closer to key. The DSHT module on each of these nodes returns (1) whether the key is loaded, (2) the number of values under key, and (3) the minimum expiry of any such values. A key is considered loaded for all but \( \beta \) insertion attempts during any given minute (\( \beta \) is 12 in the experiments we report). The client node uses this information to determine whether its store would succeed, which occurs if either the node is not yet at capacity on key, or if the node can evict an older value, which can occur if the new value’s expiry time is 50% greater than that of the old value. Call such a node, in which either property holds, as available to store. Any contacted node which is available is placed on an insertion stack, sorted by “closeness” to key. Upon hitting non-available node that is currently loaded, the key-space traversal halts.

This halting process rate-limits the number of put requests received by nodes close to key. When a key has been repeatedly stored to the extent that some node r is at capacity for the key, only \( \beta \) nodes per minute that contact r (including r itself) will go on to contact closer nodes to the key. With a perfectly balanced node ID distribution on \( n \) nodes, this means the node closest to the key only receives a maximum of \( (2^b - 1)\beta \log n / b \) store requests per second. Irregularities in the node ID distribution will increase this slightly, but the overall quantity of traffic should still be logarithmic, compared to traditional DHTs that would be linear in the number of puts.

In the second phase, Coral pops this stack until it finds a node on which the insertion succeeds. Note that, due to the asynchrony of the network, nodes that were available during the first phase may no longer be available.

This two-phase approach have several effects. (1) Coral stores pointers progressively further from the target in proportionality to their popularity. (2) Eviction ensures

---

**Figure 1**: Coral’s hierarchical routing structure. Nodes maintain the same IDs in each of their clusters; low-level clusters are naturally sparser. For a lookup on key \( k \), a node first searches on its lowest-level cluster. This lookup fails on that level if the target node closest to \( k \), node \( t_2 \), does not store the key. Upon failure, the lookup continues progressively on a next-higher cluster, having already traversed the id space up to \( t_2 \)’s prefix \( x \), and so on. Route RPCs are shown with sequential numbering. Note that while low-level neighbors usually share higher-level clusters are shown, this is not necessarily so.

- **levels()**: Returns the number of levels in Coral’s hierarchy and their corresponding round-trip-time (RTT) thresholds.

While exposing such functionality, Coral’s main task is to provide such operations efficiently and balancing load between nodes while doing so. To enable fast lookup, Coral stores keys at nodes whose nodeid’s are close to the keys in question, as defined by the XOR metric. This placement enables efficient search. More interesting, perhaps, Coral introduces “sloppiness” in its storage layer to prevent hot-spots in the indexing infrastructure, such that no node should experience disproportionately high load when serving either get or put requests for some key.

 Virtually all DHTs consider mechanisms to balance load when many users retrieve the same key, usually by replicating the key/value pair in some manner proportional to its popularity. However, to ensure their sought-after consistency semantics, put operations still all hit
that nodes near the key’s target retain long-lived values to ensure that live keys remain reachable. (3) Under higher load, find_node requests never propagate to the target. If a node fails to find free space remotely, it attempts to insert the value locally (placing itself initially at the bottom of the stack), although this operation may fail as well. However, it is acceptable that put operations fail under such high load: The indexing infrastructure already stores numerous pointers under key given this state; further instances are not necessary for the application layer.

To retrieve a key, a node simply traverses the identifier space via find_key(key) calls. Upon hitting a node storing key, the node receives the key’s corresponding list of values, and get returns. Then, the requesting client application can handle these addressing pointers in some application-specific way, e.g., contacting them in parallel to download cached web content.

### 3.3 Hierarchical Operations

Instead of one global lookup system as in [4, 15, 23], Coral uses several levels of DSHTs called clusters. While we describe latency-driven clustering for the remainder of this paper, we note here that the clustering algorithms are driven by generic policies that specify clusters’ acceptability. This flexibility could allow hierarchy creation based on a variety of criteria. For example, one could provide a clustering policy by IP routing block or by AS name, in order to build a hierarchy that follows the underlying Internet architecture, although such criteria do not always guarantee good latency.

Coral’s implementation allows for an arbitrarily-deep DSHT hierarchy. For the remainder of the paper, we consider a three-level hierarchy for latency-driven clustering. 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). Each level-i cluster is named by its randomly-chosen m-bit cluster identifier; the global identifier is predefined as 0m. In Section 6, we parameterize clusters’ expected round-trip-times and present experimental evidence to the client-side benefit of clustering.

Coral uses this hierarchy for distance-optimized lookup, visualized in Figure 1 for the Kademlia routing structure. A Coral node has the same nodeid in all clusters to which it belongs; we can view a node as projecting its presence to the same location in each of its clusters.

To retrieve a key, a requesting node r specifies the starting and stopping levels at which Coral should search. By default, it initiates the get query on its lowest (level-2) cluster to try to take advantage of network locality. If find_key on this cluster hits some node caching the key (with a pointer inserted in the same level-2 cluster), the lookup halts and returns the corresponding stored values (a hit), without ever searching higher-level clusters.

If no such node is found, the lookup will reach the target node in its cluster, call it t2, and recognize failure on this level. Node r then continues its search in its level-1 cluster. However, t2 exists at the identical location in the identifier space in its level-1 cluster. Thus, r can begin searching onward from t2 in its level-1 cluster. Even 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 identifier space. Moreover, Coral’s clusters guarantee (1) that all lookups at the beginning are fast, (2) that the system can use tight bounds for packet timeouts, and (3) that all pointers reference local data. Note that Coral achieves this property independent of any distance optimization in its underlying routing protocol.

To insert a key/value pair, an inserting node i starts by performing a put on its level-2 cluster, so that other nearby nodes can take advantage of its locality, as specified in Section 3.2. However, this node is only “correct” within the context of the level-2 cluster, i.e., it represents only some local maximum. Therefore, provided that the key is not already loaded, the node continues its insertion on the level-1 cluster, from the point onwards from where the key was inserted in level 2, much as in the retrieval case.

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. Again, Coral only traverses the identifier space once.

Two conflicting criteria impact the effectiveness of Coral’s hierarchical DSHTs. First, clusters should be large in terms of membership. The more nodes 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 nodes within a cluster should be below the cluster’s specified threshold.

### 3.4 Joining a Cluster

As in any peer-to-peer system, a peer must initially learn about some other Coral node to join the system. After contacting this well-known host, a new node proceeds to make queries to each of its k-buckets to seed its routing table. (See [17] for more information.) However, for non-global clusters, Coral adds one important requirement: A node will only join an acceptable cluster. In the case of Coral’s latency-driven clustering policy, acceptability requires that the latency to 80% of the nodes be below the cluster’s threshold. This property is easy for a node to es-
timate by collecting minimum round-trip-times (RTTs) to some subset of nodes in the cluster.

Several mechanisms could have been used to discover existing 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 quickly find nearby clusters based on network topology.

Upon joining a low-level cluster with some cluster identifier $cid$, a node inserts several mappings in its higher-level clusters, each keyed under the $/24$ IP prefix of one of its gateway routers, discovered by our fast, asynchronous traceroute tool (see Section 4). For each of the first three publicly-routable router IP addresses returned, it executes $put(hash(ip)/24, ptr)$, where $ptr$ references itself. A new node, upon joining the network, can perform a $get$ on each of its own gateway routers to learn some set of topologically-close nodes if such are available.

As packet headers contain clustering information from other nodes, each node builds, for each non-global level, a hash table that maps cluster identifiers to known peers. At the end of a clustering epoch, a node traverses through all clusters that were changed during the epoch, and chooses the cluster from the set that appears most promising. The node then actively attempts a preliminary join operation to this cluster, to gain more accurate, fresh information of its current state.

If the new cluster is better than its current, it switches clusters. If no clusters are acceptable, and the latency to fewer than 50% of the nodes in its current cluster are below the RTT threshold, it creates a new singleton cluster. This behavior prevents a poorly-connected node from impacting the performance of others at low-level clusters.

In either case, it still remains in the routing tables of nodes in its old cluster. Thus, old neighbors will still contact it and learn of its new, potentially-better cluster. This produces an avalanche effect as more and more nodes switch to the larger cluster. We now describe this decision process.

### 3.5 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.

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 (though we have not observed such behavior). To perturb such oscillations into a stable state, Coral employs a preference function $\delta$ that shifts every hour. A node selects the larger cluster only if the following holds:

$$|\log(size_A) - \log(size_B)| > \delta \min(age_A, age_B)$$

where $age$ is the current time minus the cluster’s creation time. Otherwise, a node simply selects the cluster with the lower cluster id.

We use a square wave function for $\delta$ 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. Otherwise, $\delta$’s transition perturbs clusters to a steady state.

### 4 Coral Applications

This section describes the two programs that run on top Coral’s DSHT interface to implement a content distribution network. The Coral DNS server, $dnssrv$, redirects web browsers fetching Coralized URLs to Coral HTTP servers (attempting to find ones near the requesting client). The Coral HTTP server, $CoralProxy$, functions much like caching HTTP proxy (though its request syntax differs from that of traditional HTTP proxies). $CoralProxy$ nodes cooperate so as to minimize transfer latency and minimize the load on source web servers.

#### 4.1 DNS server

When a user accesses a Coralized URL such as http://http.nyud.net:8090/www.foo.com/, the goal of DNS redirection is to have the browser resolve the hostname http.slopnet.net to the address of a nearby Coral HTTP server. To do this, Coral assumes that web browsers are close to their resolvers on the network, so that the source address of a DNS request reflects the network location of the browser. This holds to varying degrees for different clients, but is a good enough assumption that Akamai [14], Digital Island, and Mirror-image have all successfully deployed commercial CDNs based on DNS redirection.

DNS redirection poses a slightly different challenge for Coral’s peer-to-peer architecture than for centralized CDNs. In particular, centralized CDNs know the precise location of all their nodes, and can negotiate with network providers for helpful information such as BGP feeds to help map clients to appropriate nodes. In contrast, Coral is completely decentralized—its design assumes no single node even knows the identity of all other nodes in the system, let alone their precise network location. Moreover, while a centralized CDN’s DNS redirector might chose
between several hundred to a thousand groups of well-connected machines, Coral is designed for a much larger number of individual machines, many of which may be behind bottleneck links on organizations’ local networks.

Also unlike centralized CDNs, Coral is designed to be an open system. Rather than carefully provision a network of strategically-located proxy caches, we intend to make Coral freely available for anyone to run. At sites running Coral, all local users should benefit from redirection to the local Coral HTTP server. For sites not running Coral, our primary objective is to avoid bad performance, not necessarily to achieve optimal redirection. Indeed, evidence suggests that even commercial CDN’s do not always direct clients to the best cache nodes, but rather benefit by avoiding particularly suboptimal assignments [12]. Coral exploits clustering to avoid bad assignments; it uses network hints to direct clients to appropriate nodes, and ensures that once a client has hit a good node, that client will stay within the appropriate cluster for both DNS and HTTP requests.

\textit{dnssrv}. Coral’s DNS redirector, acts as an authoritative name server for the \texttt{nyud.net} domain. \textit{dnssrv} is designed to run on every Coral node (though Coral doesn’t require this, as some nodes may have the required UDP port 53 firewalled or otherwise unavailable). Though only two well-known Coral nodes are registered at the top-level .\texttt{net} DNS registry, most DNS replies contain a set of authoritative name servers, which can overwrite the list supplied by the parent zone. \textit{dnssrv} heavily exploits this mechanism to redirect subsequent DNS requests from the same client.

\textit{dnssrv} always resolves the name http.\texttt{nyud.net} as an alias or “CNAME record” for name http.L2.L1.L0.\texttt{nyud.net}. The CNAME is returned with a long time-to-live, ensuring that on subsequent lookups, clients directly request the address of http.L2.L1.L0.\texttt{nyud.net}. How \textit{dnssrv} resolves this latter name and what nameservers it piggybacks onto the reply depend on the results of network measurements it takes of the DNS client.

Upon receiving a DNS request, \textit{dnssrv} measures its round trip time and determines the addresses of the last five network hops to the client using a highly-efficient, concurrent traceroute mechanism. The probes typically require two round trip times and roughly 350 bytes of bandwidth. In order not to delay DNS clients, \textit{dnssrv} aggressively times out probes, limits the number of concurrent clients it probes, and caches the results of previous probes. Based on the measured round trip time, \textit{dnssrv} categorizes a DNS client as either level 2, level 1, or level 0, using the timing thresholds as reported by Coral.

\textit{dnssrv} uses whatever probe information (if any) it collected within the time bounds to look for location hints in the DSHT. It looks up the networks of intermediary hops to see if any coral nodes are on them or nearby. When it finds hints, \textit{dnssrv} returns the address of the best node running an HTTP server, and a set of DNS servers that includes both itself and the best DNS server with the best hint. If \textit{dnssrv} finds no hints, it returns the addresses of up to three random, recursive HTTP servers within the cluster whose level matches the client’s round trip time. In this case, it also piggybacks nameserver records for up to five nodes running \textit{dnssrv} (including itself) within the cluster.

When \textit{dnssrv} receives a DNS request from a nearby DNS client, it needs to ensure that all future DNS requests from client will stay within the appropriate level 1 or 2 ring. This is both so that the client can enjoy low-latency DNS lookups, and more importantly, so that the client’s address is more likely to hit in the traceroute cache, and the HTTP server addresses returned are more likely to be near the client. Coral exploits the hierarchical nature of DNS to lock clients into good clusters. When returning name server records, it returns them for a subdomain disproportionate to the level of the client. For example, a DNS client within 60 msec might receive a list of nameservers for domain L1.L0.\texttt{nyud.net}. One within 20 msec would get a list of nameservers for L2.L1.L0.\texttt{nyud.net}. These subdomain nameserver records have a time-to-live of one hour, and are constantly refreshed with each DNS request so long as the requester remains below the RTT threshold. Clients, once they have been given addresses of nameservers for the L2.L1.L0.\texttt{lopnet}.\texttt{net} subdomain, will no longer go back to querying the more distant nameservers they were using for parent domains.

Unfortunately, though resolvers can to some extent tolerate a fraction of unavailable DNS servers, web clients do not handle bad HTTP servers gracefully. Thus, we set a short time-to-live of 30 (for level 0) to 60 (for level 2) seconds for the address records of http.L2.L1.L0.\texttt{nyud.net}. Moreover, a Coral node only returns address records of HTTP servers whose existence it has recently verified first-hand. This sometimes means synchronously checking a Coral HTTP server’s status (via a UDP RPC) before replying to a DNS request.

### 4.2 HTTP server

The Coral HTTP server, \textit{CoralProxy}, satisfies HTTP requests for Coralized URLs. In contrast to traditional CDNs, several factors make it more important for \textit{CoralProxy} to minimize the number of accesses to source servers. First, customers of commercial CDNs are generally companies with reasonable network bandwidth, even if they need the additional capacity of a CDN for some particular application. Coral, however, is also intended to
accommodate servers behind comparatively slow network links such as home broadband connections.

Second, CDNs typically select proxies in part based on the URL requested. For example, a CDN’s DNS redirector might select a cluster of machines because of its proximity to a client, but then chose a proxy within the cluster by a hash of the source server’s hostname. In such a scenario, even if all contacted proxies fetch a URL from the source server, the server need only serve the URL a number of times proportional to the number of clusters in the CDN, typically an order of magnitude less than the total number of machines. In Coral, on the other hand, DNS redirection is based entirely on a client’s network location. Indeed, people who wish to contribute only upstream, not downstream bandwidth to the system can flag their Coral redirector will only return that node to web clients on desired networks.

In order to minimize load on the source servers, then, CoralProxy fetches web pages from other CoralProxies whenever possible. As usual, each HTTP server keeps a local cache from which it can immediately fulfill recent requests. When a client requests a non-resident URL, CoralProxy initially attempts to locate a cached copy of the referenced resource using Coral. If another Coral node has the data, CoralProxy fetches it from that node rather than contacting the source server. Of course, if Coral provides no valid referrals, CoralProxy must fetch the resource directly from the source.

While CoralProxy is fetching a web page (either from the source or from another CoralProxy), it inserts a reference to itself in the DSHT layer with a time-to-live of 20 seconds. Thus, if a flash crowd suddenly fetches a web page, except for the very first simultaneous requests which contact the source, the remaining CoralProxies will naturally form a kind of multicast tree for retrieving the web page. Once any CoralProxy obtains the full file, it inserts a much longer-lived reference to itself (e.g., 1 hour). Because the insertion algorithm accounts for TTL, these longer-lived references will overwrite shorter-lived ones, and can be stored on nodes with close IDs even under high insertion load.

Coral HTTP servers periodically renew referrals to resources in their caches. A server must not evict a web object from its cache while a reference to it may persist in the DSHT. Ideally, servers would adaptively set TTLs based on cache capacity, though this is not currently implemented. The experiments reported in this paper are sufficiently short that no objects are evicted from web caches.

5 Implementation

The Coral software is comprised of a client library and stand-alone daemon. The simple client library allows applications to connect to and interface with Coral. Coral client applications communicate over a named socket to the Coral daemon. This daemon implements multi-level put and get operations using out Kademia-style routing structures on the “client-side”. On the “server-side”, it provides access to key-value databases, by demultiplexing RPCs based on the clusters specified in packet headers. Additionally, it incorporates a cluster manager to handle cluster membership and routing table maintenance, as Coral maintains separate routing tables per cluster. Lastly, this daemon monitors the presence of Coral applications, in order to insert proper references to itself into the Coral indexing infrastructure (and allow such references to expire when appropriate). This practice enables service discovery, e.g., providing dnssrv to find A records for recursive CoralProxies.

Coral is implemented in 12,000 lines of C++. It runs in user-space with untrusted permissions, although dnssrv requires root permissions to bind port 53, and our asynchronous traceroute tool also requires root permissions to bind raw sockets. If Coral cannot bind raw sockets, it will try to exec a setuid program such as the standard traceroute, although this behavior adds system overhead and offers much poorer performance for dnssrv.

Coral uses the asynchronous I/O and RPC packages provided by the SFS toolkit [18] and is structured by asynchronous events and callbacks. All network communication is via RPC over UDP. It uses the SleepyCat software Berkeley DB package for persistent key/value storage. We have successively run Coral on Linux, OpenBSD, and FreeBSD.

The current CoralProxy is written as threaded Python program and does not support cut-through routing. A fully asynchronous server, built within the libasync framework, is in preparation and should help improve end-to-end performance.

6 Evaluation

In this section, we provide experimental results that support our following hypotheses:

1. Coral dramatically reduces load on servers, solving the “flash crowd” problem.
2. Clustering provides performance gains for popular data, resulting in good client performance.
3. Coral naturally forms suitable clusters.
4. Coral avoids hotspots within its indexing infrastructure.
To examine all claims, we present wide-area test-bed measurements of Coral on Planet Lab, an internationally-deployed test bed. The basic structure of the experiments were as follows. First, on approximately 120 Planet Lab machines, geographically distributed mainly over North America and Europe, we launch a Coral daemon, as well as a DNS nameserver and web proxy that both connect to this local daemon. In most experiments, we set the TTL threshold of level 1 to 60 msec and level 2 to 20 msec. For simplicity, all nodes are seeded with the same well-known host. The network is allowed to stabilize for 45 minutes.

Second, we ran an unmodified Apache webserver sitting behind a cable modem with 110 KB/s upstream bandwidth, on which we place 12 different 190KB files representing groups of three embedded images referenced by four web pages.

Third, we launch client processes on each machine that, after an additional random delay between 0 and 180 seconds for asynchrony, begin making HTTP GET requests to Coralized URLs. Each client generates requests for the group of three files corresponding to a randomly selected web page for a period of 30 minutes. While we recognize that web traffic generally has a Zipf distribution, we are attempting merely to simulate a flash crowd to a popular web page with three large multiple embedded images, i.e., the Slashdot effect. With approximately 120 clients, we are generating 72 req/sec on average, resulting in a cumulative download rate approximately 13.4 MB/sec.

For Experiment 4 (Section 6.4), we do not run any such clients. Instead, Coral nodes merely generate traffic at a very high rate, all for the same key, to examine how the DSHT indexing infrastructure prevents unlucky nodes from becoming overloaded.

Most of these measurements were taken while Planet-Lab was experiencing very high load and severe network congestion. We fully expect performance improvements under normal conditions.

### 6.1 Server Load

Figure 2 shows how well CoralProxy can migrate load off of a web server. We see that, during the initial minute, 17 requests hit the web server (for 12 unique files). The 5 redundant lookups are due to the simultaneity at which requests are generated, i.e., repeated requests are issued faster that the 10s of milliseconds required to propagate the first CoralProxy put on a given file. During the initial minute, slightly more requests were handled by the level 1 cluster. However, as the files propagated into CoralProxy caches, requests began quickly to be resolved within faster level 2 clusters. Within 5 minutes, the files became replicated at nearby servers, so few client requests went further than the local cache. Repeated runs of this experiment yields some variance in the relative magnitude of the clusters’ initial spike, although the number of web server hits remains consistent.

### 6.2 Client Latency

Figure 3 shows the end-to-end latency for a client to fetch a file from Coral. This latency includes the time for the client to make the initial DNS request and subsequently connect to a CoralProxy, the CoralProxy looking up a reference to the data in Coral, the CoralProxy retrieving the data either from another proxy or from the remote web server, and the client subsequently retrieving the data from

---

1 The stabilization time could be made significantly shorter by reducing the clustering epoch (5 minutes). Additionally, in real applications, clustering is in fact a simpler task, as new nodes would immediately join the nearby large cluster. In our experimental setup, clusters need to develop from an initial network comprised entirely of singletons.
the CoralProxy. Recall that our HTTP server does not implement cut-through routing. This figure includes three results: (1) the distribution of latency for clients using only single global cluster, (2) the distribution of latencies for clients in the network with 60-msec threshold level-1 clusters and a 20-msec threshold level-2 clusters, and (3) the same hierarchical network, but using traceroute during the DNS phase of the lookup to map clients to nearby CoralProxies.

The hierarchical network using traceroute clearly provides lower latency at virtually every percentile. This result is slightly favorable to our experimental setup, as all clients were running on the same LAN as a CoralProxy. In a real deployment, we expect some combination of hosts sharing networks with CoralProxies—within the same IP prefix as registered with Coral—and hosts without.

Figure 4 shows the latency of Coral get operations as seen by CoralProxies when looking-up URLs in Coral. We plot the latency of requests on a single global cluster vs. the three-level hierarchy. While the two times converge towards the 80% percentile, for almost 50% of requests, the hierarchical operation is almost an order of magnitude faster, albeit for small values.

### 6.3 Clustering

Figures 5 and 6 illustrate a snapshot of the clusters formed near the conclusion (70 minutes out) of the experiment shown in previous figures. Recall that the system was parameterized to develop a three level hierarchy with thresholds at 60 msec for level 1 and 20 msec for level 2. On each map, each unique cluster within the network is assigned an alphanumeric character. We have plotted the location of our nodes by latitude/longitude coordinates. If two nodes belong to the same cluster, they are represented by the same character on the map. As each PlanetLab site usually colocates several servers, the size of the alphanumeric value expresses the number of nodes at that side that belong to the same cluster. This map is meant to provide a qualitative feel for the organic nature of cluster development, as opposed to any quantitative measurements.

The world map shows that, at 60 msec, Coral found natural divisions along clean geographically-spatial lines for deployment of PlanetLab servers used during our experiments. The map shows 7 distinct clusters: Eastern and Central U.S., Western U.S., Europe, Brazil, Australia, Hong Kong, and Taiwan. The close correlation between network distance and physical distance suggests that, for higher thresholds, speed-of-light delays dominate round-trip-times.

The close-up of the United States shows level-2 clusters again roughly separated by physical locality. The map shows 22 distinct clusters; obvious clusters include the Pacific Northwest, Southern California, Texas, the Chicago region, Massachusetts, etc. In the Northeast Corridor, Coral formed a cluster between nodes at NYU, Columbia, Stonybrook, Princeton, Penn, Johns Hopkins, UVA, and a NYC-area Internet2 node. One particularly interesting aspect of this map is the San Francisco Bay Area, which shows three separate clusters. Close examination of individual round-trip-times between various hosts shows widely varying latencies; Coral in fact clustered correctly given the network topology.

### 6.4 Load Balancing

Finally, Figure 7 shows the extent to which Coral’s DSHTs balance requests to the same key. In this experiment, we first brought up 100 PlanetLab hosts, well-distributed over the network, all within a single level-0 cluster. Each PlanetLab node, following a 5 minute initialization, begins issuing back-to-back put/get requests at its maximal rate. All operations were to the same key, the values during put requests were randomized. On av-
Figure 5: World view of level-1 clusters for a 60 msec threshold

Figure 6: United States view of level-2 clusters for a 20 msec threshold
average, each node issued 400 put/get operation pairs per second. Interestingly, the target node in this experiment, given the random key, was distantly located at Bologna, Italy.

The graphs show the number of RPCs that have hit each node within the first two minutes of activity, as a function of the node’s distance from the key according to the XOR metric. During later time periods, the number of put RPCs seen at each node stays relatively stable (given no system membership changes). After the first minute, no further get RPCs are seen, given that the keys get pushed back to all nodes in the system given such high demand, and no values expire within the time period of the experiment. Recall that put RPCs include RPCs in both of the two put phases. Careful analysis of the put structure shows that intermediate nodes to the key receive a number of RPCs that are low multiples of $\beta$ (12), where $\beta$ was defined as the load parameter in Section 3.2: Each node only allows $\beta$ RPCs “through” it per minute. Thus, we get a total of 1200 put RPCs per minute from all nodes.

Note that no nodes on the right side of either graph, i.e., those greater than half the distance from the target, receive any RPCs. This condition is expected, as these nodes begin their routing operations by flipping their most-significant bit to match that of key. The regular structure of the graph is also attributable to the determinism of this algorithm, modulo any incomplete information in the network or misbalanced nodeid allocation.

Most DHTs would enjoy similar performance for get operations by result-caching on the lookup path. However, all put operations would hit the same node.

7 Conclusions

Coral is a peer-to-peer content distribution network that harness people’s apparent willingness to redistribute data they themselves find useful. It indexes cached web content with a new distributed storage abstraction called a DSHT. DSHTs store (key, value) pairs, and can scale to large numbers of read and write accesses of the same key. Using the DSHT abstraction for network hints, Coral forms successively larger clusters of nearby machines, ensuring nearby replicas of data can be located and retrieved without querying more distant nodes. Finally, a peer-to-peer DNS layer redirects clients to nearby Coral HTTP servers, ensuring that even unmodified web browsers benefit from Coral’s content distribution, and more importantly avoid overloading source servers.

Measurements of Coral demonstrate that it allows under-provisioned web sites to achieve dramatically higher capacity. A web server behind a slow cable modem experiences hardly any load when hit by a flash crowd with a sustained aggregate transfer rate of over 13 MBytes/sec. Moreover, Coral’s clustering mechanism appears to form qualitatively sensible geographic clusters, which provide quantitatively better performance than un-clustered systems.

We hope to make Coral freely available in the near future, so that even people with slow network connections can publish web sites whose capacity grows automatically with popularity.

References


[23] 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, October 2001.


