Democratizing content publication with Coral

Michael J. Freedman, Eric Freudenthal, David Mazières
New York University
http://www.scs.cs.nyu.edu/coral/

Abstract

CoralCDN is a 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 cheap broadband Internet connection. Volunteer sites that run CoralCDN automatically replicate content as a side effect of users accessing it. Publishing through CoralCDN is as simple as making a small change to the hostname in an object's URL; a peer-to-peer DNS layer transparently redirects browsers to nearby participating cache nodes, which in turn cooperate to minimize load on the origin web server. One of the system's key goals is to avoid creating hot spots that might dissuade volunteers and hurt performance. It achieves this through Coral, a latency-optimized hierarchical indexing infrastructure based on a novel abstraction called a *distributed sloppy hash table*, or DSHT.

1 Introduction

The availability of content on the Internet is to a large degree a function of the cost shouldered by the publisher. A well-funded web site can reach huge numbers of people through some combination of load-balanced servers, fast network connections, and commercial content distribution networks (CDNs). Publishers who cannot afford such amenities are limited in the size of audience and type of content they can serve. Moreover, their sites risk sudden overload following publicity, a phenomenon nicknamed the "Slashdot" effect, after a popular web site that periodically links to under-provisioned servers, driving unsustainable levels of traffic to them. Thus, 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 publishers can afford to serve themselves—volunteers can mirror the data on their own servers and networks. Indeed, the Internet has a long history of organizations with good network connectivity mirroring data they consider to be of value. More recently, peer-to-peer file sharing has demonstrated the willingness of even individual broadband users to dedicate upstream bandwidth to redistribute content the users themselves enjoy. Additionally, organizations that mirror popular content reduce their downstream bandwidth utilization and improve the latency for local users accessing the mirror.

This paper describes CoralCDN, a decentralized, self-organizing, peer-to-peer web-content distribution network. CoralCDN leverages the aggregate bandwidth of volunteers running the software to absorb and dissipate most of the traffic for web sites using the system. In so doing, CoralCDN replicates content in proportion to the content's popularity, regardless of the publisher's resources—in effect democratizing content publication.

To use CoralCDN, a content publisher—or someone posting a link to a high-traffic portal—simply appends "`.nyud.net:8090`" to the hostname in a URL. Through DNS redirection, oblivious clients with unmodified web browsers are transparently redirected to nearby Coral web caches. These caches cooperate to transfer data from nearby peers whenever possible, minimizing both the load on the origin web server and the end-to-end latency experienced by browsers.

CoralCDN is built on top of a novel key/value indexing infrastructure called Coral. Two properties make Coral ideal for CDNs. First, Coral allows nodes to locate nearby cached copies of web objects without querying more distant nodes. Second, Coral prevents hot spots in the infrastructure, even under degenerate loads. For instance, if every node repeatedly stores the same key, the rate of requests to the most heavily-loaded machine is still only logarithmic in the total number of nodes.

Coral exploits overlay routing techniques recently popularized by a number of peer-to-peer distributed hash tables (DHTs). However, Coral differs from DHTs in several ways. First, Coral's locality and hot-spot prevention properties are not possible for DHTs. Second, Coral's architecture is based on clusters of well-connected machines. Clusters are exposed in the interface to higher-level software, and in fact form a crucial part of the DNS redirection mechanism. Finally, to achieve its goals, Coral provides weaker consistency than traditional DHTs. For that reason, we call its indexing abstraction a *distributed sloppy hash table*, or DSHT.

CoralCDN makes a number of contributions. It enables people to publish content that they previously could not or would not because of distribution costs. It is the first completely decentralized and self-organizing web-content distribution network. Coral, the indexing infrastructure, provides a new abstraction potentially of use to any application that needs to locate nearby instances of resources on the network. Coral also introduces an epidemic clustering algorithm that exploits distributed network measurements. Furthermore, Coral is the first peer-to-peer key/value index that can scale to many stores of the same key without hot-spot congestion, thanks to a new rate-limiting technique. Finally, CoralCDN contains the first peer-to-peer DNS redirection infrastructure, allowing the system to inter-operate with unmodified web browsers.

Measurements of CoralCDN demonstrate that it allows under-provisioned web sites to achieve dramatically higher capacity, and its clustering provides quantitatively better performance than locality-unaware systems.

The remainder of this paper is structured as follows. Section 2 provides a high-level description of CoralCDN, and Section 3 describes its DNS system and web caching components. In Section 4, we describe the Coral indexing infrastructure, its underlying DSHT layers, and the clustering algorithms. Section 5 includes an implementation overview and Section 6 presents experimental results. Section 7 describes related work, Section 8 discusses future work, and Section 9 concludes.

2 The Coral Content Distribution Network

The Coral Content Distribution Network (CoralCDN) is composed of three main parts: (1) a network of cooperative HTTP proxies that handle users' requests,[1] (2) a network of DNS nameservers for `nyucd.net` that map clients to nearby Coral HTTP proxies, and (3) the underlying Coral indexing infrastructure and clustering machinery on which the first two applications are built.

2.1 Usage Models

To enable immediate and incremental deployment, CoralCDN is transparent to clients and requires no software or plug-in installation. CoralCDN can be used in a variety of ways, including:

- **Publishers.** A web site publisher for `x.com` can change selected URLs in their web pages to "Coralized" URLs, such as `http://www.x.com.nyud.net:8090/y.jpg`.
- **Third-parties.** An interested third-party—*e.g.*, a poster to a web portal or a Usenet group—can Coralize a URL before publishing it, causing all embedded relative links to use CoralCDN as well.
- **Users.** Coral-aware users can manually construct Coralized URLs when surfing slow or overloaded web sites. All relative links and HTTP redirects are automatically Coralized.

[1]While Coral's HTTP proxy definitely provides proxy functionality, it is not an HTTP proxy in the strict RFC2616 sense; it serves requests that are syntactically formatted for an ordinary HTTP server.

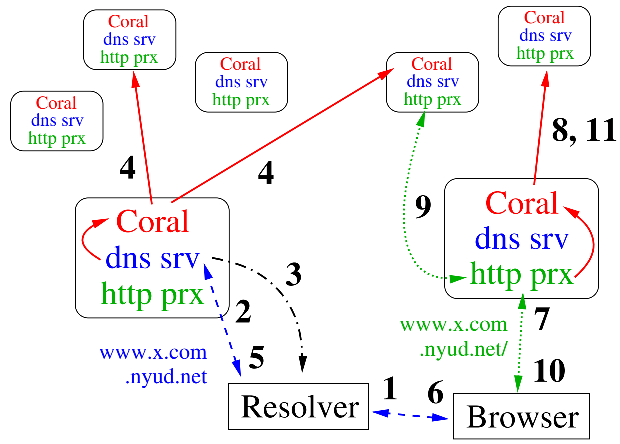


Figure 1: Using CoralCDN, the steps involved in resolving a Coralized URL and returning the corresponding file, per Section 2.2. Rounded boxes represent CoralCDN nodes running Coral, DNS, and HTTP servers. Solid arrows correspond to Coral RPCs, dashed arrows to DNS traffic, dotted-dashed arrows to network probes, and dotted arrows to HTTP traffic.

2.2 System Overview

Figure 1 shows the steps that occur when a client accesses a Coralized URL, such as `http://www.x.com.nyud.net:8090/`, using a standard web browser. The two main stages—DNS redirection and HTTP request handling—both use the Coral indexing infrastructure.

1. A client sends a DNS request for `www.x.com.nyud.net` to its local resolver.
2. The client's resolver attempts to resolve the hostname using some Coral DNS server(s), possibly starting at one of the few registered under the `.net` domain.
3. Upon receiving a query, a Coral DNS server probes the client to determines its round-trip-time and last few network hops.
4. Based on the probe results, the DNS server checks Coral to see if there are any known nameservers and/or HTTP proxies near the client's resolver.
5. The DNS server replies, returning any servers found through Coral in the previous step; if none were found, it returns a random set of nameservers and proxies. In either case, if the DNS server is close to the client, it only returns nodes that are close to itself (see Section 3.1).
6. The client's resolver returns the address of a Coral HTTP proxy for `www.x.com.nyud.net`.

7. The client sends the HTTP request `http://www.x.com.nyud.net:8090/` to the specified proxy. If the proxy is caching the file locally, it returns the file and stops. Otherwise, this process continues.
8. The proxy looks up the web object's URL in Coral.
9. If Coral returns the address of a node caching the object, the proxy fetches the object from this node. Otherwise, the proxy downloads the object from the origin server, `www.x.com` (not shown).
10. The proxy stores the web object and returns it to the client browser.
11. The proxy stores a reference to itself in Coral, recording the fact that is now caching the URL.

2.3 The Coral Indexing Abstraction

This section introduces the Coral indexing infrastructure as used by CoralCDN. Coral provides a *distributed sloppy hash table* (DSHT) abstraction. DSHTs are designed for applications storing soft-state key/value pairs, where multiple values may be stored under the same key. CoralCDN uses this mechanism to map a variety of types of key onto addresses of CoralCDN nodes. In particular, it uses DSHTs to find Coral nameservers topologically close clients' networks, to find HTTP proxies caching particular web objects, and to locate nearby Coral nodes for the purposes of minimizing internal request latency.

Instead of one global overlay as in [5, 14, 27], each Coral node belongs to several distinct DSHTs called *clusters*. Each cluster is characterized by a maximum desired network round-trip-time (RTT) we call the *diameter*. The system is parameterized by a fixed hierarchy of diameters known as *levels*. Every node is a member of one DSHT at each level. A group of nodes can form a level-i cluster if a high-enough fraction their pair-wise RTTs are below the level-i diameter threshold. Although Coral's implementation allows for an arbitrarily-deep DSHT hierarchy, this paper describes a three-level hierarchy with thresholds of ∞, 60 msec, and 20 msec for level-0, -1, and -2 clusters respectively. Coral queries nodes in higher-level, fast clusters before those in lower-level, slower clusters. This both reduces the latency of lookups and increases the chances of returning values stored by nearby nodes.

Coral provides the following interface to higher-level applications:

- $put(key, val, ttl, [levels])$: Inserts a mapping from the key to some arbitrary value, specifying the time-to-live of the reference. The caller may optionally specify a subset of the cluster hierarchy to restrict the operation to certain levels.
- $get(key, [levels])$: Retrieves some subset of the values stored under a key. Again, one can optionally specify a subset of the cluster hierarchy.
- $nodes(level, count, [target], [services])$: Returns *count* neighbors belonging to the node's cluster as specified by $level$. *target*, if supplied, specifies the IP address of a machine to which the returned nodes would ideally be near. Coral can probe *target* and exploit network topology hints stored in the DSHT to satisfy the request. If *services* is specified, Coral will only return nodes running the particular service, *e.g.*, an HTTP proxy or DNS server.
- $levels()$: Returns the number of levels in Coral's hierarchy and their corresponding RTT thresholds.

The next section describes the design of CoralCDN's DNS redirector and HTTP proxy—especially with regard to their use of Coral's DSHT abstraction and clustering hierarchy—before returning to Coral in Section 4.

3 Application-Layer Components

The Coral DNS server directs browsers fetching Coralized URLs to Coral HTTP proxies, attempting to find ones near the requesting client. These HTTP proxies exploit each others' caches in such a way as to minimize both transfer latency and the load on origin web servers.

3.1 The Coral DNS server

The Coral DNS server, *dnssrv*, returns IP addresses of Coral HTTP proxies when browsers look up the hostnames in Coralized URLs. To improve locality, it attempts to return proxies near requesting clients. In particular, whenever a DNS resolver (client) contacts a nearby *dnssrv* instance, *dnssrv* both returns proxies within an appropriate cluster, and ensures that future DNS requests from that client will not need to leave the cluster. Using the *nodes* function, *dnssrv* also exploits Coral's on-the-fly network measurement capabilities and stored topology hints to increase the chances of clients discovering nearby DNS servers.

More specifically, every instance of *dnssrv* is an authoritative nameserver for the domain `nyucd.net`. Assuming a 3-level hierarchy, as Coral is generally configured, *dnssrv* maps any domain name ending `http.L2.L1.L0.nyucd.net` to one or more Coral HTTP proxies. (For an $(n+1)$-level hierarchy, the domain name is extended out to `L`n in the obvious way.) Because such names are somewhat unwieldy, we established a DNS DNAME alias [4], `nyud.net`, with target `http.L2.L1.L0.nyucd.net`. Any domain name ending `nyud.net` is therefore equivalent to the same name with suffix `http.L2.L1.L0.nyucd.net`, allowing Coralized URLs to have the more concise form `http://www.x.com.nyud.net:8090/`.

dnssrv assumes that web browsers are generally close to their resolvers on the network, so that the source address of a DNS query reflects the browser's network location. This assumption holds to varying degrees, but is good enough that Akamai [12], Digital Island [6], and Mirror Image [21] have all successfully deployed commercial CDNs based on DNS redirection. The locality problem therefore is reduced to returning proxies that are near the source of a DNS request. In order to achieve locality, *dnssrv* measures its round-trip-time to the resolver and categorizes it by level. For a 3-level hierarchy, the resolver will correspond to a level 2, level 1, or level 0 client, depending on how its RTT compares to Coral's cluster-level thresholds.

When asked for the address of a hostname ending `http.L2.L1.L0.nyucd.net`, *dnssrv*'s reply contains two sections of interest: A set of addresses for the name—*answers* to the query—and a set of nameservers for that name's domain—known as the *authority* section of a DNS reply. *dnssrv* returns addresses of *CoralProxies* in the cluster whose level corresponds to the client's level categorization. In other words, if the RTT between the DNS client and *dnssrv* is below the level-i threshold (for the best i), *dnssrv* will only return addresses of Coral nodes in its level-i cluster. *dnssrv* obtains a list of such nodes with the *nodes* function. Note that *dnssrv* always returns *CoralProxy* addresses with short time-to-live fields (30 seconds for levels 0 and 1, 60 for level 2).

To achieve better locality, *dnssrv* also specifies the client's IP address as a *target* argument to *nodes*. This causes Coral to probe the addresses of the last five network hops to the client and use the results to look for clustering hints in the DSHTs. To avoid significantly delaying clients, Coral maps these network hops using a fast, built-in traceroute-like mechanism that combines concurrent probes and aggressive time-outs to minimize latency. The entire mapping process generally requires around 2 RTTs and 350 bytes of bandwidth. A Coral node caches results to avoid repeatedly probing the same client.

The closer *dnssrv* is to a client, the better its selection of *CoralProxy* addresses will likely be for the client. *dnssrv* therefore exploits the authority section of DNS replies to lock a DNS client into a good cluster whenever it happens upon a nearby *dnssrv*. As with the answer section, *dnssrv* selects the nameservers it returns from the appropriate cluster level and uses the *target* argument to exploit measurement and network hints. Unlike addresses in the answer section, however, it gives nameservers in the authority section a long TTL (one hour). A nearby *dnssrv* must therefore override any inferior nameservers a DNS client may be caching from previous queries. *dnssrv* does so by manipulating the domain for which returned nameservers are servers. To clients more distant than the level-1 timing threshold, *dnssrv* claims to return nameservers for domain `L0.nyucd.net`. For clients closer than that threshold, it returns nameservers for `L1.L0.nyucd.net`. For clients closer than the level-2 threshold, it returns nameservers for domain `L2.L1.L0.nyucd.net`. Because DNS resolvers query the servers for the most specific known domain, this scheme allows closer *dnssrv* instances to override the results of more distant ones.

Unfortunately, although resolvers can tolerate a fraction of unavailable DNS servers, browsers do not handle bad HTTP servers gracefully. (This is one reason for returning *CoralProxy* addresses with short TTL fields.) As an added precaution, *dnssrv* only returns *CoralProxy* addresses which it has recently verified first-hand. This sometimes means synchronously checking a proxy's status (via a UDP RPC) prior replying to a DNS query. We note further that people who wish to contribute only upstream bandwidth can flag their proxy as "non-recursive," in which case *dnssrv* will only return that proxy to clients on local networks.

3.2 The Coral HTTP proxy

The Coral HTTP proxy, *CoralProxy*, satisfies HTTP requests for Coralized URLs. It seeks to provide reasonable request latency and high system throughput, even while serving data from origin servers behind comparatively slow network links such as home broadband connections. This design space requires particular care in minimizing load on origin servers compared to traditional CDNs, for two reasons. First, many of Coral's origin servers are likely to have slower network connections than typical customers of commercial CDNs. Second, commercial CDNs often collocate a number of machines at each deployment site and then select proxies based in part on the URL requested—effectively distributing URLs across proxies. Coral, in contrast, selects proxies only based on client locality. Thus, in CoralCDN, it is much easier for every single proxy to end up fetching a particular URL.

To aggressively minimize load on origin servers, a *CoralProxy* must fetch web pages from other proxies whenever possible. Each proxy keeps a local cache from which it can immediately fulfill requests. When a client requests a non-resident URL, *CoralProxy* first attempts to locate a cached copy of the referenced resource using Coral (a *get*), with the resource indexed by a SHA-1 hash of its URL [22]. If *CoralProxy* discovers that one or more other proxies have the data, it attempts to fetch the data from the proxy to which it first connects. If Coral provides no referrals or if no referrals return the data, *CoralProxy* must fetch the resource directly from the origin.

While *CoralProxy* is fetching a web object—either from the origin or from another *CoralProxy*—it inserts a reference to itself in its DSHTs with a time-to-live of 20 seconds. (It will renew this short-lived reference until it completes the download.) Thus, if a flash crowd suddenly fetches a web page, all *CoralProxies*, other than the first simultaneous requests, 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 they can be stored on well-selected nodes even under high insertion load, as later described in Section 4.2.

CoralProxies periodically renew referrals to resources in their caches. A proxy should not evict a web object from its cache while a reference to it may persist in the DSHT. Ideally, proxies would adaptively set TTLs based on cache capacity, though this is not yet implemented.

4 Coral: A Hierarchical Indexing System

This section describes the Coral indexing infrastructure, which CoralCDN leverages to achieve scalability, self-organization, and efficient data retrieval. We describe how Coral implements the put and get operations that form the basis of its *distributed sloppy hash table* (DSHT) abstraction: the underlying key-based routing layer (4.1), the DSHT algorithms that balance load (4.2), and the changes that enable latency and data-placement optimizations within a hierarchical set of DSHTs (4.3). Finally, we describe the clustering mechanisms that manage this hierarchical structure (4.4).

4.1 Coral's Key-Based Routing Layer

Coral's keys are opaque 160-bit ID values; nodes are assigned IDs in the same 160-bit identifier space. A node's ID is the SHA-1 hash of its IP address. Coral defines a distance metric on IDs. Henceforth, we describe a node as being *close* to a key if the distance between the key and the node's ID is small. A Coral put operation stores a key/value pair at a node close to the key. A get operation searches for stored key/value pairs at nodes successively closer to the key. To support these operations, a node requires some mechanism to discover other nodes close to any arbitrary key.

Every DSHT contains a routing table. For any key k, a node R's routing table allows it to find a node closer to k, unless R is already the closest node. These routing tables are based on Kademlia [17], which defines the distance between two values in the ID-space to be their bitwise exclusive or (XOR), interpreted as an unsigned integer. Using the XOR metric, IDs with longer matching prefixes (of most significant bits) are numerically *closer*.

The size of a node's routing table in a DSHT is logarithmic in the total number of nodes comprising the DSHT. If a node R is not the closest node to some key k, then R's routing table almost always contains either the closest node to k, or some node whose distance to k is at least one bit shorter than R's. This permits R to visit a sequence of nodes with monotonically decreasing distances $[d_1, d_2, \ldots]$ to k, such that the encoding of d_{i+1} as a binary number has one fewer bit than d_i. As a result, the expected number of iterations for R to discover the closest node to k is logarithmic in the number of nodes.

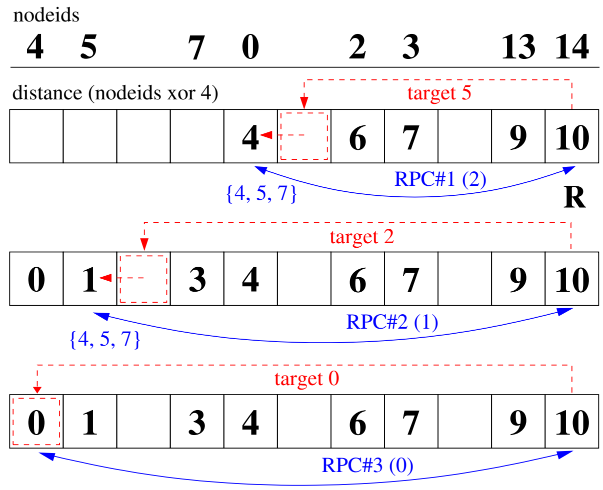


Figure 2: Example of routing operations in a system containing eight nodes with IDs $\{4, 5, 7, 0, 2, 3, 13, 14\}$. In this illustration, node R with $id = 14$ is looking up the node closest to key $k = 4$, and we have sorted the nodes by their distance to k. The top boxed row illustrates XOR distances for the nodes $\{0, 2, 3, 13, 14\}$ that are initially known by R. R first contacts a known peer whose distance to k is closest to half of R's distance ($10/2 = 5$); in this illustration, this peer is node zero, whose distance to k is $0 \oplus 4 = 4$. Data in RPC requests and responses are shown in parentheses and braces, respectively: R asks node zero for its peers that are half-way closer to k, *i.e.*, those at distance $\frac{4}{2} = 2$. R inserts these new references into its routing table (middle row). R now repeats this process, contacting node five, whose distance 1 is closest to $\frac{4}{2}$. Finally, R contacts node four, whose distance is 0, and completes its search (bottom row).

Figure 2 illustrates the Coral routing algorithm, which successively visits nodes whose distances to the key are approximately halved each iteration. Traditional key-based routing layers attempt to route directly to the node closest to the key whenever possible [25, 26, 31, 35], resorting to several intermediate hops only when faced with incomplete routing information. By caching additional routing state—beyond the necessary $\log(n)$ references—these systems in practice manage to achieve routing in a *constant* number of hops. We observe that frequent references to the same key can generate high levels of traffic in nodes close to the key. This congestion, called *tree saturation*, was first identified in shared-memory interconnection networks [24].

To minimize tree saturation, each iteration of a Coral search prefers to correct only b bits at a time.[2] More specifically, let $\text{splice}(k, r, i)$ designate the most significant bi bits of k followed by the least significant $160 - bi$ bits of r. If node R with ID r wishes to search for key k, R first initializes a variable $t \leftarrow r$. At each iteration, R updates $t \leftarrow \text{splice}(k, t, i)$, using the smallest value of i that yields a new value of t. The next hop in the lookup path is the closest node to t that already exists in R's routing table. As described below, by limiting the use of potentially closer known hops in this way, Coral can avoid overloading any node, even in the presence of very heavily accessed keys.

The potential downside of longer lookup paths is higher lookup latency in the presence of slow or stale nodes. In order to mitigate these effects, Coral keeps a window of multiple outstanding RPCs during a lookup, possibly contacting the closest few nodes to intermediary target t.

4.2 Sloppy Storage

Coral uses a sloppy storage technique that caches key/value pairs at nodes whose IDs are close to the key being referenced. These cached values reduce hot-spot congestion and tree saturation throughout the indexing infrastructure: They frequently satisfy *put* and *get* requests at nodes other than those closest to the key. This characteristic differs from DHTs, whose *put* operations all proceed to nodes closest to the key.

The Insertion Algorithm. Coral performs a two-phase operation to insert a key/value pair. In the first, or "forward," phase, Coral routes to nodes that are successively closer to the key, as previously described. However, to avoid tree saturation, an insertion operation may terminate prior to locating the closest node to the key, in which case the key/value pair will be stored at a more distant node. More specifically, the forward phase terminates whenever the storing node happens upon another node that is both *full* and *loaded* for the key:

1. A node is *full* with respect to some key k when it stores l values for k whose TTLs are all at least one-half of the new value.
2. A node is *loaded* with respect to k when it has received more than the maximum *leakage rate* β requests for k within the past minute.

In our experiments, $l = 4$ and $\beta = 12$, meaning that under high load, a node claims to be loaded for all but one store attempt every 5 seconds. This prevents excessive numbers of requests from hitting the key's closest nodes, yet still allows enough requests to propagate to keep values at these nodes fresh.

In the forward phase, Coral's routing layer makes repeated RPCs to contact nodes successively closer to the key. Each of these remote nodes returns (1) whether the key is loaded and (2) the number of values it stores under the key, along with the minimum expiry time of any such values. The client node uses this information to determine if the remote node can accept the store, potentially evicting a value with a shorter TTL. This forward phase terminates when the client node finds either the node closest to the key, or a node that is full and loaded with respect to the key. The client node places all contacted nodes that are not both full and loaded on a stack, ordered by XOR distance from the key.

During the reverse phase, the client node attempts to insert the value at the remote node referenced by the top stack element, *i.e.*, the node closest to the key. If this operation does not succeed—perhaps due to others' insertions—the client node pops the stack and tries to insert on the new stack top. This process is repeated until a store succeeds or the stack is empty.

This two-phase algorithm avoids tree saturation by storing values progressively further from the key. Still, eviction and the leakage rate β ensure that nodes close to the key retain long-lived values, so that live keys remain reachable: β nodes per minute that contact an intermediate node (including itself) will go on to contact nodes closer to the key. For a perfectly-balanced tree, the key's closest node receives only $\left(\beta \cdot (2^b - 1) \cdot \lceil \frac{\log n}{b} \rceil\right)$ store requests per minute, when fixing b bits per iteration.

Proof sketch. Each node in a system of n nodes can be uniquely identified by a string S of $\log n$ bits. Consider S to be a string of b-bit digits. A node will contact the closest node to the key before it contacts any other node if and only if its ID differs from the key in exactly one digit. There are $\lceil (\log n)/b \rceil$ digits in S. Each digit can take on $2^b - 1$ values that differ from the key. Every node that differs in one digit will throttle all but β requests per minute. Therefore, the closest node receives a maximum rate of $\left(\beta \cdot (2^b - 1) \cdot \lceil \frac{\log n}{b} \rceil\right)$ RPCs per minute.

Irregularities in the node ID distribution may increase this rate slightly, but the overall rate of traffic is still logarithmic, while in traditional DHTs it is linear. Section 6.4 provides supporting experimental evidence.

The Retrieval Algorithm. To retrieve the value associated with a key k, a node simply traverses the ID space with RPCs. When it finds a peer storing k, the remote peer returns k's corresponding list of values. The node terminates its search and *get* returns. The requesting client application handles these redundant references in some application-specific way, *e.g.*, *CoralProxy* contacts multiple sources in parallel to download cached content.

Multiple stores of the same key will be spread over multiple nodes. The pointers retrieved by the application are

[2] Experiments in this paper use $b = 1$.

thus distributed among those stored, providing load balancing both *within* Coral and between servers using Coral.

4.3 Hierarchical Operations

For locality-optimized routing and data placement, Coral uses several *levels* of DSHTs called clusters. Each level-i cluster is named by a randomly-chosen 160-bit cluster identifier; the level-0 cluster ID is predefined as 0^{160}. Recall that a set of nodes should form a cluster if their average, pair-wise RTTs are below some threshold. As mentioned earlier, we describe a three-level hierarchy with thresholds of ∞, 60 msec, and 20 msec for level-0, -1, and -2 clusters respectively. In Section 6, we present experimental evidence to the client-side benefit of clustering.

Figure 3 illustrates Coral's hierarchical routing operations. Each Coral node has the same node ID 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. This structure must be reflected in Coral's basic routing infrastructure, in particular to support switching between a node's distinct DSHTs midway through a lookup.[3]

The Hierarchical Retrieval Algorithm. A requesting node R specifies the starting and stopping levels at which Coral should search. By default, it initiates the *get* query on its highest (level-2) cluster to try to take advantage of network locality. If routing RPCs on this cluster hit some node storing the key k (RPC 1 in Fig. 3), the lookup halts and returns the corresponding stored value(s)—a *hit*—without ever searching lower-level clusters.

If a key is not found, the lookup will reach k's closest node C_2 in this cluster (RPC 2), signifying failure at this level. So, node R continues the search in its level-1 cluster. As these clusters are very often concentric, C_2 likely exists at the identical location in the identifier space in all clusters, as shown. R begins searching onward from C_2 in its level-1 cluster (RPC 3), having already traversed the ID-space up to C_2's prefix.

Even if the search eventually switches to the global cluster (RPC 4), the total number of RPCs required is about the same as a single-level lookup service, as a lookup continues from the point at which it left off in the identifier space of the previous cluster. Thus, (1) all lookups at the beginning are fast, (2) the system can tightly bound RPC timeouts, and (3) all pointers in higher-level clusters reference data *within* that local cluster.

The Hierarchical Insertion Algorithm. A node starts by performing a *put* on its level-2 cluster as in Section 4.2, so that other nearby nodes can take advantage of locality.

[3]We initially built Coral using the Chord [31] routing layer as a block-box; difficulties in maintaining distinct clusters and the complexity of the subsequent system caused us to scrap the implementation.

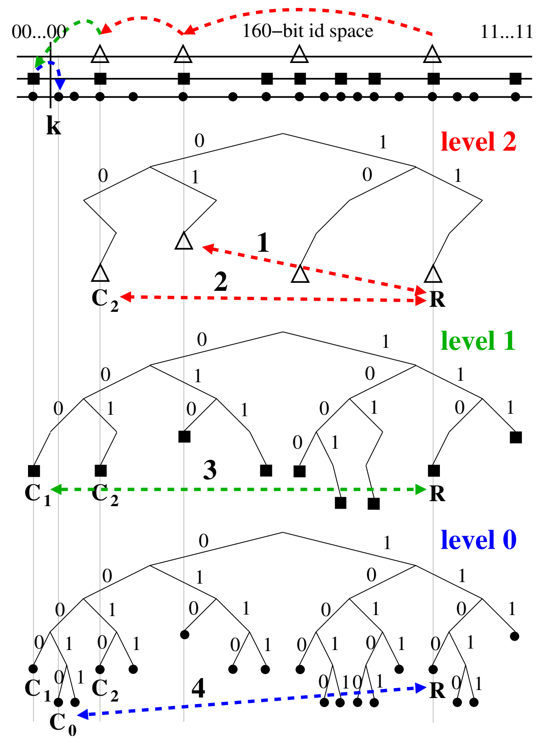


Figure 3: Coral's hierarchical routing structure. Nodes use the same IDs in each of their clusters; higher-level clusters are naturally sparser. Note that a node can be identified in a cluster by its shortest unique ID prefix, *e.g.*, "11" for R in its level-2 cluster; nodes sharing ID prefixes are located on common subtrees and are closer in the XOR metric. While higher-level neighbors usually share lower-level clusters as shown, this is not necessarily so. RPCs for a retrieval on key k are sequentially numbered.

However, this placement is only "correct" within the context of the local level-2 cluster. Thus, provided that the key is not already loaded, the node continues its insertion in the level-1 cluster from the point at which the key was inserted in level 2, much as in the retrieval case. Again, Coral traverses the ID-space only once. As illustrated in Figure 3, this practice results in a loose hierarchical cache, whereby a lower-level cluster contains nearly all data stored in the higher-level clusters to which its members also belong.

To enable such cluster-aware behavior, the headers of every Coral RPC include the sender's cluster information: the identifier, age, and a size estimate 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, as described next.

4.4 Joining and Managing Clusters

As in any peer-to-peer system, a peer contacts an existing node to join the system. Next, a new node makes several queries to seed its routing tables. However, for non-global clusters, Coral adds one important requirement: A node will only join an *acceptable* cluster, where acceptability requires that the latency to 80% of the nodes be below the cluster's threshold. A node can easily determine whether this condition holds by recording minimum round-trip-times (RTTs) to some subset of nodes belonging to the cluster.

While nodes learn about clusters as a side effect of normal lookups, Coral also exploits its DSHTs to store hints. When Coral starts up, it uses its built-in fast traceroute mechanism (described in Section 3.1) to determine the addresses of routers up to five hops out. Excluding any private ("RFC1918") IP addresses, Coral uses these router addresses as keys under which to index clustering hints in its DSHTs. More specifically, a node R stores mappings from each router address to its own IP address and UDP port number. When a new node S, sharing a gateway with R, joins the network, it will find one or more of R's hints and quickly cluster with it, assuming R is, in fact, near S.

In addition, nodes store mappings to themselves using as keys any IP subnets they directly connect to and the 24-bit prefixes of gateway router addresses. These prefix hints are of use to Coral's *level* function, which traceroutes clients in the other direction; addresses on forward and reverse traceroute paths often share 24-bit prefixes.

Nodes continuously collect clustering information from peers: All RPCs include round-trip-times, cluster membership, and estimates of cluster size. Every five minutes, each node considers changing its cluster membership based on this collected data. If this collected data indicates that an alternative candidate cluster is desirable, the node first validates the collected data by contacting several nodes within the candidate cluster by routing to selected keys. A node can also form a new singleton cluster when 50% of its accesses to members of its present cluster do not meet the RTT constraints.

If probes indicate that 80% of a cluster's nodes are within acceptable TTLs and the cluster is larger, it replaces a node's current cluster. If multiple clusters are acceptable, then Coral chooses the largest cluster.

Unfortunately, Coral has only rough *approximations* of cluster size, based on its routing-table size. If nearby clusters A and B are of similar sizes, inaccurate estimations could lead to oscillation as nodes flow back-and-forth (although we have not observed such behavior). To perturb an oscillating system 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:

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

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 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. Otherwise, δ's transition perturbs clusters to a steady state.[4]

In either case, a node that switches clusters 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. This merging of clusters is very beneficial. While a small cluster diameter provides fast lookup, a large cluster capacity increases the hit rate.

5 Implementation

The Coral indexing system is composed of a client library and stand-alone daemon. The simple client library allows applications, such as our DNS server and HTTP proxy, to connect to and interface with the Coral daemon. Coral is 14,000 lines of C++, the DNS server, *dnssrv*, is 2,000 lines of C++, and the HTTP proxy is an additional 4,000 lines. All three components use the asynchronous I/O library provided by the SFS toolkit [19] and are structured by asynchronous events and callbacks. Coral network communication is via RPC over UDP. We have successfully run Coral on Linux, OpenBSD, FreeBSD, and Mac OS X.

6 Evaluation

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

1. CoralCDN 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 prevents hot spots within its indexing system.

[4]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 cluster id. Should the clusters oscillate when $\delta = 2$ (as the estimations "hit" with one around 2^2-times larger), the nodes will all flow to the larger one when δ returns to zero.

To examine all claims, we present wide-area measurements of a synthetic work-load on CoralCDN nodes running on PlanetLab, an internationally-deployed test bed. We use such an experimental setup because traditional tests for CDNs or web servers are not interesting in evaluating CoralCDN: (1) Client-side traces generally measure the cacheability of data and client latencies. However, we are mainly interested in how well the system handles load spikes. (2) Benchmark tests such as SPECweb99 measure the web server's throughput on disk-bound access patterns, while CoralCDN is designed to reduce load on off-the-shelf web servers that are *network-bound*.

The basic structure of the experiments were is follows. First, on 166 PlanetLab machines geographically distributed mainly over North America and Europe, we launch a Coral daemon, as well as a *dnssrv* and *CoralProxy*. For experiments referred to as *multi-level*, we configure a three-level hierarchy by setting the clustering RTT threshold of level 1 to 60 msec and level 2 to 20 msec. Experiments referred to as *single-level* use only the level-0 global cluster. No objects are evicted from *CoralProxy* caches during these experiments. For simplicity, all nodes are seeded with the same well-known host. The network is allowed to stabilize for 30 minutes.[5]

Second, we run an unmodified Apache web server sitting behind a DSL line with 384 Kbit/sec upstream bandwidth, serving 12 different 41KB 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 multiple, large, embedded images (*i.e.*, the Slashdot effect). With 166 clients, we are generating 99.6 requests/sec, resulting in a cumulative download rate of approximately $32,800$ Kb/sec. This rate is almost two orders of magnitude greater than the origin web server could handle. Note that this rate was chosen synthetically and in no way suggests a maximum system throughput.

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

[5]The stabilization time could be made shorter by reducing the clustering period (5 minutes). Additionally, in real applications, clustering is in fact a simpler task, as new nodes would immediately join nearby large clusters as they join the pre-established system. In our setup, clusters develop from an initial network comprised entirely of singletons.

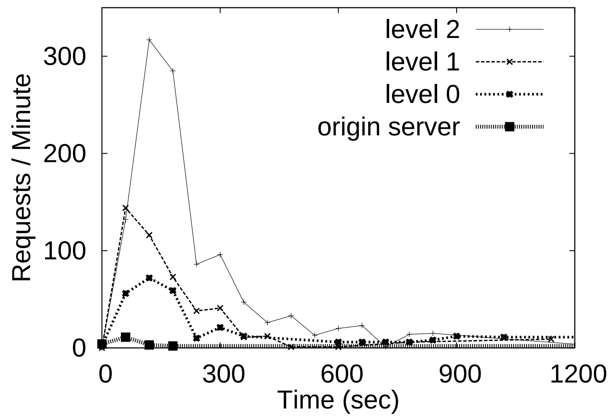


Figure 4: The number of client accesses to *CoralProxies* and the origin HTTP server. *CoralProxy* accesses are reported relative to the cluster level from which data was fetched, and do not include requests handled through local caches.

6.1 Server Load

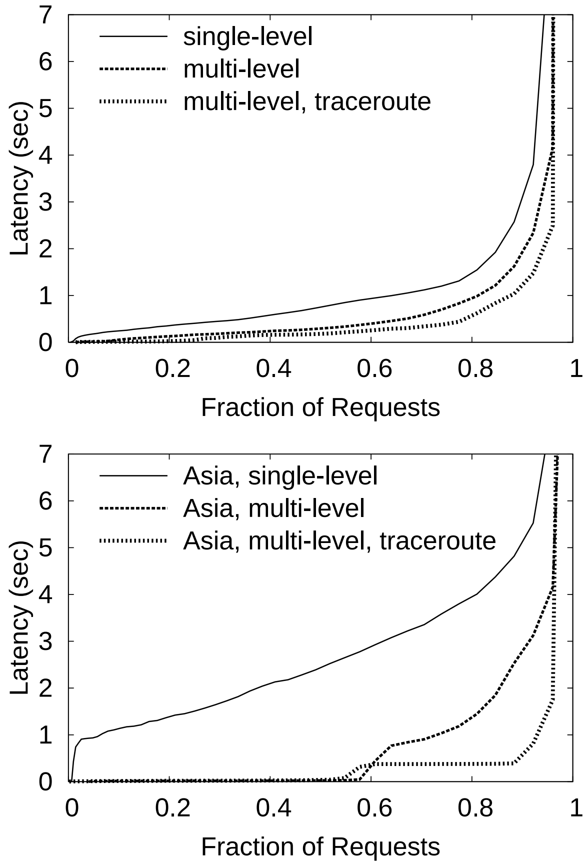
Figure 4 plots the number of requests per minute that could not be handled by a *CoralProxy*'s local cache. During the initial minute, 15 requests hit the origin web server (for 12 unique files). The 3 redundant lookups are due to the simultaneity at which requests are generated; subsequently, requests are handled either through CoralCDN's wide-area cooperative cache or through a proxy's local cache, supporting our hypothesis that CoralCDN can migrate load off of a web server.

During this first minute, equal numbers of requests were handled by the level-1 and level-2 cluster caches. However, as the files propagated into *CoralProxy* caches, requests quickly were resolved within faster level-2 clusters. Within 8-10 minutes, the files became replicated at nearly every server, so few client requests went further than the proxies' local caches. Repeated runs of this experiment yielded some variance in the relative magnitudes of the initial spikes in requests to different levels, although the number of origin server hits remained consistent.

6.2 Client Latency

Figure 5 shows the end-to-end latency for a client to fetch a file from CoralCDN, following the steps given in Section 2.2. The top graph shows the latency across all PlanetLab nodes used in the experiment, the bottom graph only includes data from the clients located on 5 nodes in Asia (Hong Kong (2), Taiwan, Japan, and the Philippines). Because most nodes are located in the U.S. or Europe, the performance benefit of clustering is much more pronounced on the graph of Asian nodes.

Recall that this end-to-end latency includes the time for the client to make a DNS request and to connect to the



Request latency (sec)	All nodes		Asian nodes	
	50%	96%	50%	96%
single-level	0.79	9.54	2.52	8.01
multi-level	0.31	4.17	0.04	4.16
multi-level, traceroute	0.19	2.50	0.03	1.75

Figure 5: End-to-End client latency for requests for Coralized URLs, comparing the effect of single-level vs. multi-level clusters and of using traceroute during DNS redirection. The top graph includes all nodes; the bottom only nodes in Asia.

discovered *CoralProxy*. The proxy attempts to fulfill the client request first through its local cache, then through Coral, and finally through the origin web server. We note that *CoralProxy* implements cut-through routing by forwarding data to the client prior to receiving the entire file.

These figures report three results: (1) the distribution of latency of clients using only a single level-0 cluster (the solid line), (2) the distribution of latencies of clients using multi-level clusters (dashed), and (3) the same hierarchical network, but using traceroute during DNS resolution to map clients to nearby proxies (dotted).

All clients ran on the same subnet (and host, in fact) as a *CoralProxy* in our experimental setup. This would not be the case in the real deployment: We would expect a combination of hosts sharing networks with *CoralProxies*—within the same IP prefix as registered with Coral—and hosts without. Although the multi-level network using traceroute provides the lowest latency at most percentiles, the multi-level system without traceroute also performs better than the single-level system. Clustering has a clear performance benefit for clients, and this benefit is particularly apparent for poorly-connected hosts.

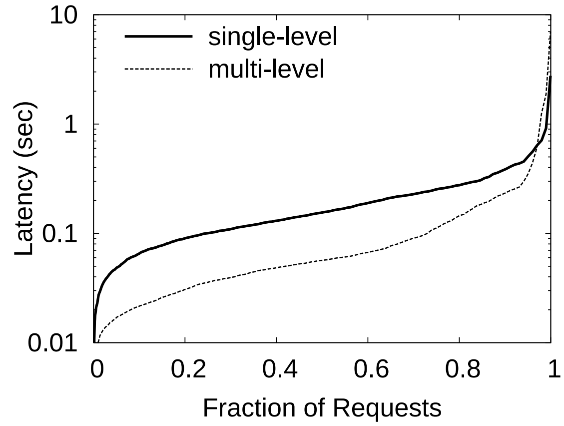


Figure 6: Latencies for proxy to get keys from Coral.

Figure 6 shows the latency of get operations, as seen by *CoralProxies* when they lookup URLs in Coral (Step 8 of Section 2.2). We plot the get latency on the single level-0 system vs. the multi-level systems. The multi-level system is 2-5 times faster up to the 80% percentile. After the 98% percentile, the single-level system is actually faster: Under heavy packet loss, the multi-system requires a few more timeouts as it traverses its hierarchy levels.

6.3 Clustering

Figure 7 illustrates a snapshot of the clusters from the previous experiments, at the time when clients began fetching URLs (30 minutes out). This map is meant to provide a qualitative feel for the organic nature of cluster development, as opposed to offering any quantitative measurements. On both maps, each unique, non-singleton cluster within the network is assigned a letter. 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 letter. As each PlanetLab site usually collocates several servers, the size of the letter expresses the number of nodes at that site that belong to the same cluster. For example, the very large “H” (world map) and “A” (U.S. map) correspond to nodes collocated at U.C. Berkeley. We did not include singleton clusters on the maps to improve readability; post-run analysis showed that such nodes’ RTTs to others (surprisingly, sometimes even at the same site) were above the Coral thresholds.

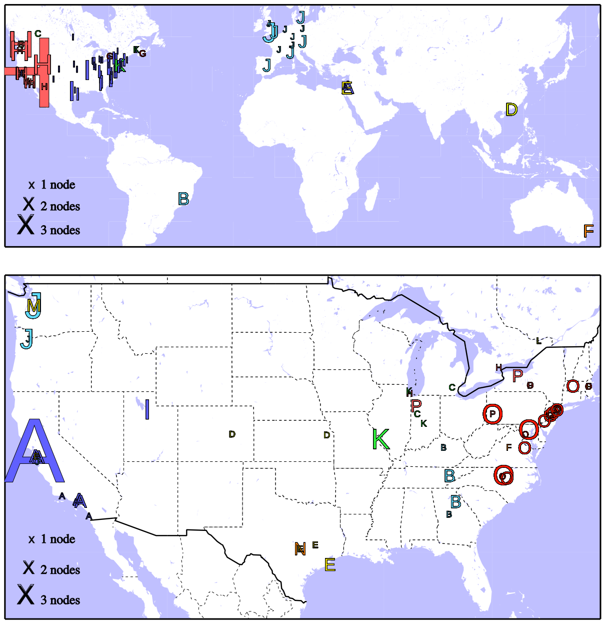


Figure 7: World view of level-1 clusters (60 msec threshold), and United States view of level-2 clusters (20 msec threshold). Each unique, non-singleton cluster is assigned a letter; the size of the letter corresponds to collocated nodes in the same cluster.

The world map shows that Coral found natural divisions between sets of nodes along geospatial lines at a 60 msec threshold. The map shows several distinct regions, the most dramatic being the Eastern U.S. (70 nodes), the Western U.S. (37 nodes), and Europe (19 nodes). The close correlation between network and physical distance suggests that speed-of-light delays dominate round-trip-times. Note that, as we did not plot singleton clusters, the map does not include three Asian nodes (in Japan, Taiwan, and the Philippines, respectively).

The United States map shows level-2 clusters again roughly separated by physical locality. The map shows 16 distinct clusters; obvious clusters include California (22 nodes), the Pacific Northwest (9 nodes), the South, the Midwest, etc. The Northeast Corridor cluster contains 29 nodes, stretching from North Carolina to Massachusetts. One interesting aspect of this map is the three separate, non-singleton clusters in the San Francisco Bay Area. Close examination of individual RTTs between these sites shows widely varying latencies; Coral clustered correctly given the underlying network topology.

6.4 Load Balancing

Finally, Figure 8 shows the extent to which a DSHT balances requests to the same key ID. In this experiment, we ran 3 nodes on each of the earlier hosts for a total of 494 nodes. We configured the system as a single level-0 cluster. At the same time, all PlanetLab nodes began to issue back-to-back put/get requests at their maximum (non-concurrent) rates. All operations referenced the same key; the values stored during put requests were randomized. On average, each node issued 400 put/get operation pairs per second, for a total of approximately 12 million put/get requests per minute, although only a fraction hit the network. Once a node is storing a key, get requests are satisfied locally. Once it is *loaded*, each node only allows the leakage rate β RPCs "through" it per minute.

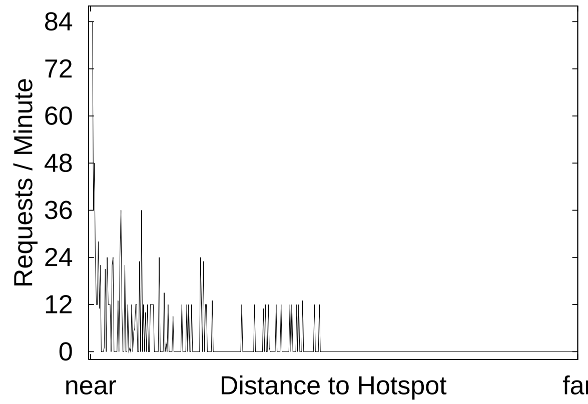


Figure 8: The total number of put RPCs hitting each Coral node per minute, sorted by distance from node ID to target key.

The graphs show the number of put RPCs that hit each node in steady-state, sorted by the XOR distance of the node's ID to the key. During the first minute, the closest node received 106 put RPCs. In the second minute, as shown in Figure 8, the system reached steady-state with the closest node receiving 83 put RPCs per minute. Recall that our equation in Section 4.2 predicts that it should receive $(\beta \cdot \log n) = 108$ RPCs per minute. The plot strongly emphasizes the efficacy of the leakage rate $\beta = 12$, as the number of RPCs received by the majority of nodes is a low multiple of 12.

No nodes on the far side of the graph received any RPCs. Coral's routing algorithm explains this condition: these nodes begin routing by flipping their ID's most-significant bit to match the key's, and they subsequently contact a node on the near side. We have omitted the graph of get RPCs: During the first minute, the most-loaded node received 27 RPCs; subsequently, the key was widely distributed and the system quiesced.

7 Related work

CoralCDN builds on previous work in peer-to-peer systems and web-based content delivery.

7.1 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 key ID; $get(key)$ returns this stored value, just as in a normal hash table. Most DHTs use a key-based routing layer—such as CAN [25], Chord [31], Kademlia [17], Pastry [26], or Tapestry [35]—and store keys on the node whose ID is closest to the key. Keys must be well distributed to balance load among nodes. DHTs often replicate multiply-fetched key/value pairs for scalability, *e.g.*, by having peers replicate the pair onto the second-to-last peer they contacted as part of a *get* request.

DHTs can act either as actual data stores or merely as directory services storing pointers. CFS [5] and PAST [27] take the former approach to build a distributed file system: They require true read/write consistency among operations, where writes should atomically replace previously-stored values, not modify them.

Using the network as a directory service, Tapestry [35] and Coral relax the consistency of operations in the network. To put a key, Tapestry routes along fast hops between peers, placing at each peer 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. Coral's flexible clustering provides similar latency-optimized lookup and data placement, and its algorithms prevent multiple stores from forming hot spots. SkipNet also builds a hierarchy of lookup groups, although it explicitly groups nodes by domain name to support organizational disconnect [9].

7.2 Web caching and content distribution

Web caching systems fit within a large class of CDNs that handle high demand through diverse replication.

Prior to the recent interest in peer-to-peer systems, several projects proposed cooperative Web caching [2, 7, 8, 16]. These systems either multicast queries or require that caches know some or all other servers, which worsens their scalability, fault-tolerance, and susceptibility to hot spots. Although the cache hit rate of cooperative web caching increases only to a certain level, corresponding to a moderate population size [34], highly-scalable cooperative systems can still increase the total system throughput by reducing server-side load.

Several projects have considered peer-to-peer overlays for web caching, although all such systems only benefit participating clients and thus require widespread adoption to reduce server load. Stading *et al.* use a DHT to cache replicas [29], and PROOFS uses a randomized overlay to distribute popular content [30]. Both systems focus solely on mitigating flash crowds and suffer from high request latency. Squirrel proposes web caching on a traditional DHT, although only for organization-wide networks [10]. Squirrel reported poor load-balancing when the system stored pointers in the DHT. We attribute this to the DHT's inability to handle too many values for the same key—Squirrel only stored 4 pointers per object—while CoralCDN references many more proxies by storing different sets of pointers on different nodes. SCAN examined replication policies for data disseminated through a multicast tree from a DHT deployed at ISPs [3].

Akamai [1] and other commercial CDNs use DNS redirection to reroute client requests to local clusters of machines, having built detailed maps of the Internet through a combination of BGP feeds and their own measurements, such as traceroutes from numerous vantage points [28]. Then, upon reaching a cluster of collocated machines, hashing schemes [11, 32] 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.

Such centrally-managed CDNs appear to offer two benefits over CoralCDN. (1) CoralCDN's network measurements, via traceroute-like probing of DNS clients, are somewhat constrained in comparison. CoralCDN nodes do not have BGP feeds and are under tight latency constraints to avoid delaying DNS replies while probing. Additionally, Coral's design assumes that no single node even knows the identity of all other nodes in the system, let alone their precise network location. Yet, if many people adopt the system, it will build up a rich database of neighboring networks. (2) CoralCDN offers less aggregate storage capacity, as cache management is completely localized. But, it is designed for a much larger number of machines and vantage points: CoralCDN may provide better performance for small organizations hosting nodes, as it is not economically efficient for commercial CDNs to deploy machines behind most bottleneck links.

More recently, CoDeeN has provided users with a set of open web proxies [23]. Users can reconfigure their browsers to use a CoDeeN proxy and subsequently enjoy better performance. The system has been deployed, and anecdotal evidence suggests it is very successful at distributing content efficiently. Earlier simulation results show that certain policies should achieve high system throughput and low request latency [33]. (Specific details of the deployed system have not yet been published, including an Akamai-like service also in development.)

Although CoDeeN gives *participating* users better performance to *most* web sites, CoralCDN's goal is to gives *most* users better performance to *participating* web sites—namely those whose publishers have "Coralized" the URLs. The two design points pose somewhat different challenges. For instance, CoralCDN takes pains to greatly minimize the load on under-provisioned origin servers, while CoDeeN has tighter latency requirements as it is on the critical path for *all* web requests. Finally, while CoDeeN has suffered a number of administrative headaches, many of these problems do not apply to CoralCDN, as, *e.g.*, CoralCDN does not allow POST operations or SSL tunneling, and it can be barred from accessing particular sites without affecting users' browsing experience.

8 Future Work

Security. This paper does not address CoralCDN's security issues. Probably the most important issue is ensuring the integrity of cached data. Given our experience with spam on the Internet, we should expect that adversaries will attempt to replace cached data with advertisements for pornography or prescription drugs. A solution is future work, but breaks down into three components.

First, honest Coral nodes should not cache invalid data. A possible solution might include embedding self-certifying pathnames [20] in Coralized URLs, although this solution requires server buy-in. Second, Coral nodes should be able to trace the path that cached data has taken and exclude data from known bad systems. Third, we should try to prevent clients from using malicious proxies. This requires client buy-in, but offers additional incentives for organizations to run Coral: Recall that a client will access a local proxy when one is available, or administrators can configure a local DNS resolver to always return a *specific* Coral instance. Alternatively, "SSL splitting" [15] provides end-to-end security between clients and servers, albeit at a higher overhead for the origin servers.

CoralCDN may require some additional abuse-prevention mechanisms, such as throttling bandwidth hogs and restricting access to address-authenticated content [23]. To leverage our redundant resources, we are considering efficient erasure coding for large-file transfers [18]. For such, we have developed on-the-fly verification mechanisms to limit malicious proxies' abilities to waste a node's downstream bandwidth [13].

Leveraging the Clustering Abstraction. This paper presents clustering mainly as a performance optimization for lookup operations and DNS redirection. However, the clustering algorithms we use are driven by *generic* policies that 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, for a simple mechanism that reflects administrative control and performs well under network partition. Or, Coral's clusters could be used to explicitly encode a web-of-trust security model in the system, especially useful given its standard open-admissions policy. Then, clusters could easily represent trust relationships, allowing lookups to resolve at the most trustworthy hosts. Clustering may prove to be a very useful abstraction for building interesting applications.

Multi-cast Tree Formation. CoralCDN may transmit multiple requests to an origin HTTP server at the beginning of a flash crowd. This is caused by a race condition at the key's closest node, which we could eliminate by extending store transactions to provide return status information (like test-and-set in shared-memory systems). Similar extensions to store semantics may be useful for balancing its dynamically-formed dissemination trees.

Handling Heterogeneous Proxies. We should consider the heterogeneity of proxies when performing DNS redirection and intra-Coral HTTP fetches. We might use some type of feedback-based allocation policy, as proxies can return their current load and bandwidth availability, given that they are already probed to determine liveness.

Deployment and Scalability Studies. We are planning an initial deployment of CoralCDN as a long-lived PlanetLab port 53 (DNS) service. In doing so, we hope to gather measurements from a large, active client population, to better quantify CoralCDN's scalability and effectiveness: Given our client-transparency, achieving wide-spread use is much easier than with most peer-to-peer systems.

9 Conclusions

CoralCDN is a peer-to-peer web-content distribution network that harnesses people's willingness to redistribute data they themselves find useful. It indexes cached web content with a new distributed storage abstraction called a DSHT. DSHTs map a key to multiple values and can scale to many stores of the same key without hot-spot congestion. Coral successfully clusters nodes by network diameter, ensuring that 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 *CoralProxies*, allowing unmodified web browsers to benefit from CoralCDN, and more importantly, to avoid overloading origin servers.

Measurements of CoralCDN demonstrate that it allows under-provisioned web sites to achieve dramatically higher capacity. A web server behind a DSL line experiences hardly any load when hit by a flash crowd with a sustained aggregate transfer rate that is two orders of magnitude greater than its bandwidth. Moreover, Coral's clustering mechanism forms qualitatively sensible geographic clusters and provides quantitatively better performance than locality-unaware systems.

We have made CoralCDN freely available, so that even people with slow connections can publish web sites whose capacity grows automatically with popularity. Please visit `http://www.scs.cs.nyu.edu/coral/`.

Acknowledgments. We are grateful to Vijay Karamcheti for early conversations that helped shape this work. We thank David Andersen, Nick Feamster, Daniel Giffin, Robert Grimm, and our shepherd, Marvin Theimer, for their helpful feedback on drafts of this paper. Petar Maymounkov and Max Krohn provided access to Kademlia data structure and HTTP parsing code, respectively. We thank the PlanetLab support team for allowing us the use of UDP port 53 (DNS), despite the additional hassle this caused them. Coral is part of project IRIS (`http://project-iris.net/`), supported by the NSF under Cooperative Agreement No. ANI-0225660. David Mazières is supported by an Alfred P. Sloan Research Fellowship. Michael Freedman is supported by an NDSEG Fellowship.

References

[1] Akamai Technologies, Inc. http://www.akamai.com/, 2004.

[2] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical internet object cache. In *USENIX*, Jan 1996.

[3] Y. Chen, R. Katz, and J. Kubiatowicz. SCAN: A dynamic, scalable, and efficient content distribution network. In *Proceedings of the International Conference on Pervasive Computing*, Zurich, Switzerland, Aug 2002.

[4] M. Crawford. RFC 2672: Non-terminal DNS name redirection, Aug 1999.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *SOSP*, Banff, Canada, Oct 2001.

[6] Digital Island, Inc. http://www.digitalisland.com/, 2004.

[7] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web-cache sharing protocol. Technical Report 1361, CS Dept, U. Wisconson, Madison, Feb 1998.

[8] S. Gadde, J. Chase, and M. Rabinovich. A taste of crispy squid. In *Workshop on Internet Server Perf.*, Madison, WI, Jun 1998.

[9] N. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *USITS*, Seattle, WA, Mar 2003.

[10] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *PODC*, Monterey, CA, Jul 2002.

[11] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, May 1997.

[12] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. *WWW8 / Computer Networks*, 31(11–16):1203–1213, 1999.

[13] M. Krohn, M. J. Freedman, and D. Mazières. On-the-fly verification of rateless erasure codes for efficient content distribution. In *IEEE Symp. on Security and Privacy*, Oakland, CA, May 2004.

[14] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *ASPLOS*, Cambridge, MA, Nov 2000.

[15] C. Lesniewski-Laas and M. F. Kaashoek. SSL splitting: Securely serving data from untrusted caches. In *USENIX Security*, Washington, D.C., Aug 2003.

[16] R. Malpani, J. Lorch, and D. Berger. Making world wide web caching servers cooperate. In *WWW*, Apr 1995.

[17] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS*, Cambridge, MA, Mar 2002.

[18] P. Maymounkov and D. Mazières. Rateless codes and big downloads. In *IPTPS*, Berkeley, CA, Feb 2003.

[19] D. Mazières. A toolkit for user-level file systems. In *USENIX*, Boston, MA, Jun 2001.

[20] D. Mazières and M. F. Kaashoek. Escaping the evils of centralized control with self-certifying pathnames. In *ACM SIGOPS European Workshop*, Sep 1998.

[21] Mirror Image Internet. http://www.mirror-image.com/, 2004.

[22] *FIPS Publication 180-1: Secure Hash Standard*. National Institute of Standards and Technology (NIST), Apr 1995.

[23] V. Pai, L. Wang, K. Park, R. Pang, and L. Peterson. The dark side of the web: An open proxy's view. In *HotNets*, Cambridge, MA, Nov 2003.

[24] G. Pfister and V. A. Norton. "hot spot" contention and combining in multistage interconnection networks. *IEEE Trans. on Computers*, 34(10), Oct 1985.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, San Diego, CA, Aug 2001.

[26] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, Nov 2001.

[27] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, Banff, Canada, Oct 2001.

[28] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *SIGCOMM*, Pittsburgh, PA, Aug 2002.

[29] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *IPTPS*, Cambridge, MA, Mar 2002.

[30] A. Stavrou, D. Rubenstein, and S. Sahu. A lightweight, robust p2p system to handle flash crowds. In *IEEE ICNP*, Paris, France, Nov 2002.

[31] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *IEEE/ACM Trans. on Networking*, 2002.

[32] D. Thaler and C. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Trans. on Networking*, 6(1):1–14, 1998.

[33] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on cdn robustness. In *OSDI*, Boston, MA, Dec 2002.

[34] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative web proxy caching. In *SOSP*, Kiawah Island, SC, Dec 1999.

[35] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE J. Selected Areas in Communications*, 2003.