Appeared in ***7th USENIX Symposium on Network Design and Implementation (NSDI '10)***

Experiences with CoralCDN: A Five-Year Operational View

Michael J. Freedman
Princeton University

Abstract

CoralCDN is a self-organizing web content distribution network (CDN). Publishing through CoralCDN is as simple as making a small change to a URL's hostname; a decentralized DNS layer transparently directs browsers to nearby participating cache nodes, which in turn cooperate to minimize load on the origin webserver. CoralCDN has been publicly available on PlanetLab since March 2004, accounting for the majority of its bandwidth and serving requests for several million users (client IPs) per day. This paper describes CoralCDN's usage scenarios and a number of experiences drawn from its multi-year deployment. These lessons range from the specific to the general, touching on the Web (APIs, naming, and security), CDNs (robustness and resource management), and virtualized hosting (visibility and control). We identify design aspects and changes that helped CoralCDN succeed, yet also those that proved wrong for its current environment.

1 Introduction

The goal of CoralCDN was to make desired web content available to everybody, regardless of the publisher's own resources or dedicated hosting services. To do so, CoralCDN provides an open, self-organizing web content distribution network (CDN) that any publisher is free to use, without any prior registration, authorization, or special configuration. Publishing through CoralCDN is as simple as appending a suffix to a URL's hostname, *e.g.*, `http://example.com.`*`nyud.net`*`/`. This URL modification may be done by clients, origin servers, or third parties that link to these domains. Clients accessing such *Coralized* URLs are transparently directed by CoralCDN's network of DNS servers to nearby participating proxies. These proxies, in turn, coordinate to serve content and thus minimize load on origin servers.

CoralCDN was designed to automatically and scalably handle sudden spikes in traffic for new content [14]. It can efficiently discover cached content anywhere in its network, and it dynamically replicates content in proportion to its popularity. Both techniques help minimize origin requests and satisfy changing traffic demands.

While originally designed for decentralized and unmanaged settings, CoralCDN was deployed on the PlanetLab research network [27] in March 2004, given PlanetLab's convenience and availability. CoralCDN has since remained publicly available for more than five years at hundreds of PlanetLab sites world-wide. Accounting for a majority of public PlanetLab traffic and users, CoralCDN typically serves several terabytes of data per day, in response to tens of millions of HTTP requests from around two million users (unique client IP addresses).

Over the course of its deployment, we have come to acknowledge several realities. On a positive note, CoralCDN's notably simple interface led to widespread and innovative uses. Sites began using CoralCDN as an elastic infrastructure, dynamically redirecting traffic to CoralCDN at times of high resource contention and pulling back as traffic levels abated. On the flip side, fundamental parts of CoralCDN's design were ill-suited for its deployment and the majority of its use. If one were to consider the various reasons for its use—for resurrecting long-unavailable sites, supporting random surfing, distributing popular content, and mitigating flash crowds—CoralCDN's design is insufficient for the first, unnecessary for the second, and overkill for the third, at least given its current deployment. But diverse and unanticipated use is unavoidable for an open system, yet openness is a necessary design choice for handling the final flash-crowd scenario.

This paper provides a retrospective of our experience building and operating CoralCDN over the past five years. Our purpose is threefold. First, after summarizing CoralCDN's published design [14] in Section §2, we present data collected over the system's production deployment and consider its implications. Second, we discuss various deployment challenges we encountered and describe our preferred solutions. Some of these changes we have implemented and incorporated into CoralCDN; others require adoption by third-parties. Third, given these insights, we revisit the problem of building a secure, open, and scalable content distribution network. More specifically, this paper addresses the following topics:

- *The success of CoralCDN's design given observed usage patterns (§3).* Our verdict is mixed: A large majority of its traffic does not require any cooperative caching at all, yet its handling of flash crowds relies on such cooperation.
- *Web security implications of CoralCDN's open API (§4).* Through its open API, sites began leveraging CoralCDN as an elastic resource for content distri-

bution. Yet this very openness exposed a number of web security challenges. Many can be attributed to a lack of explicitness for specifying appropriate protection domains, and they arise due to violations of traditional security principles (such as least privilege, complete mediation, and fail-safe defaults [33]).

- *Resource management in CDNs (§5).* CoralCDN commonly faced the challenge of interacting with oversubscribed and ill-behaved resources, both remote origin servers and its own deployment platform. Various aspects of its design react conservatively to change and perform admission control for resources.
- *Desired properties for deployment platforms (§6).* Application deployments could benefit from greater visibility into and control over lower layers of their platforms. Some challenges are again confounded when information and policies cannot be expressed explicitly between layers.
- *Directions for building large-scale, cooperative CDNs (§7).* While using decentralized algorithms, CoralCDN currently operates on a centrally-administered, smaller-scale testbed of trusted servers. We revisit the challenge of escaping this setting.

Rather than focus on CoralCDN's self-organizing algorithms, the majority of this paper analyzes CoralCDN as an example of an open web service on a virtualized platform. As such, the experiences we detail may have implications to a wider audience, including those developing distributed hash tables (DHTs) for key-value storage, CDNs or web services for elastic provisioning, virtualized network facilities for programmable networks, or cloud computing platforms for virtualized hosting. While many of the observations we report are neither new nor surprising in hindsight, many relate to mistakes, oversights, or limitations of CoralCDN's original design that only became apparent to us from its deployment.

We next review CoralCDN's architecture and protocols; a more complete description can be found in [14]. All system details presented after §2 were developed subsequent to that publication. We discuss related work throughout the paper as we touch on different aspects of CoralCDN.

2 Original CoralCDN Design

The Coral Content Distribution Network is composed of three main parts: (1) a network of cooperative HTTP proxies that handle client requests from users, (2) a network of DNS nameservers for `nyud.net` that map clients to nearby CoralCDN HTTP proxies, and (3) the underlying Coral indexing infrastructure and clustering machinery on which the first two applications are built. This paper consistently refers to the system's *indexing* layer as Coral, and the entire content distribution system as CoralCDN.

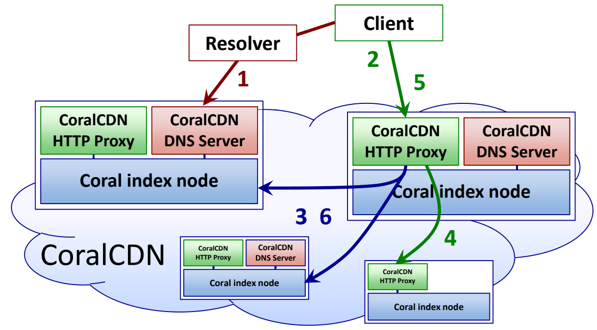

Figure 1: **The steps involved in serving a Coralized URL.**

2.1 System overview

At a high level, the following steps occur when a client issues a request to CoralCDN, as shown in Figure 1.

1. ***Resolving DNS.*** A client resolves a "Coralized" domain name (*e.g.*, of the form `example.com.nyud.net`) using CoralCDN nameservers. A CoralCDN nameserver probes the client to determine its round-trip-time and uses this information to determine appropriate nameservers and proxies to return.
2. ***Processing HTTP client requests.*** The client sends an HTTP request for a Coralized URL to one of the returned proxies. If the proxy is caching the web object locally, it returns the object and the client is finished. Otherwise, the proxy attempts to find the object on another CoralCDN proxy.
3. ***Discovering cooperative-cached content.*** The proxy looks up the object's URL in the Coral indexing layer.
4. ***Retrieving content.*** 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 `example.com`.
5. ***Serving content to clients.*** The proxy stores the web object to disk and returns it to the client browser.
6. ***Announcing cached content.*** The proxy stores a reference to itself in Coral, recording the fact that is now caching the URL.

This section reviews the design of the Coral indexing layer and the CDN's proxies, as proposed in [14].

2.2 Coral indexing layer

The Coral indexing layer is closely related to the structure and organization of distributed hash tables like Chord [34] and Kademlia [23], with the latter serving as the basis for its underlying algorithm. The system maps opaque keys onto nodes by hashing their value onto a flat, semantic-free identifier (ID) space; nodes are assigned identifiers in the same ID space. It allows scalable key lookup (in $O(\log(n))$ overlay hops for n-node systems), reorganizes itself upon network membership changes, and provides robust behavior against failure.

Compared to "traditional" DHTs, Coral introduced a few novel techniques that were well-suited for its particular application [13]. Its key-value indexing layer was designed with weaker consistency requirements in mind, and its lookup structure self-organized into a locality-optimized hierarchy of clusters of peers. After all, a client need not discover all proxies caching a particular file, it only needs to find several such proxies, preferably ones nearby. Like most DHTs, Coral exposes *put* and *get* operations, to announce one's address as caching a web object, and to discover other proxies caching the object associated with a particular URL, respectively. Inserted addresses are soft-state mappings with a time-to-live (TTL) value.

Coral's *put* and *get* operations are designed to spread load, both within the DHT and across CoralCDN proxies. To *get* the proxy addresses associated with a key k, a node traverses the ID space with iterative RPCs, and it stops upon finding any remote peer storing values for k. This peer need not be the one closest to k (in terms of DHT identifier space distance). To *put* a key/value pair, Coral routes to nodes successively closer to k and stops when finding either (1) the nodes closest to k or (2) one that is experiencing high request rates for k and already is caching several corresponding values (with longer-lived TTLs). It stores the pair at the node closest to k that it managed to reach. These processes prevent tree saturation in the DHT.

To improve locality, these routing operations are not initially performed across the entire global overlay. Instead, each Coral node belongs to several distinct routing structures called *clusters*. Each cluster is characterized by a maximum desired network round-trip-time (RTT). The system is parameterized by a fixed hierarchy of clusters with different expected RTT thresholds. Coral's deployment uses a three-level hierarchy, with level-0 denoting the global cluster and level-2 the most local one. Coral employs distributed algorithms to form localized, stable clusters, which we briefly return to in §5.3.

Every node belongs to one cluster at each level, as in Figure 2. Coral queries nodes in fast clusters before those in slower clusters. This both reduces lookup latency and increases the chance of returning values stored at nearby nodes, which correspond to addresses of nearby proxies.

2.3 The CoralCDN HTTP proxy

CoralCDN seeks to aggressively minimize load on origin servers. This section summarizes how its proxies use Coral for inter-proxy cooperation and adaptation to flash crowds.

2.3.1 Locality-optimized inter-proxy transfers

Each CoralCDN proxy keeps a local cache from which it can immediately fulfill client requests. When a client requests a non-resident URL, CoralCDN proxies attempt to fetch web content from each other, using the Coral indexing layer for discovery. A proxy only contacts a URL's origin server after the Coral indexing layer provides no referrals or none of its referrals return the data.

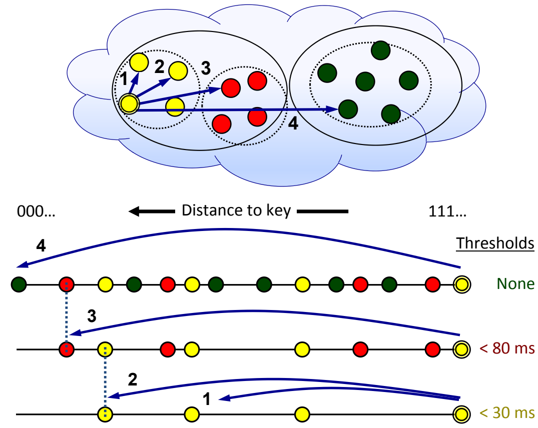

Figure 2: ***Coral's three-level hierarchical overlay structure.*** **A node first queries others in its level-2 cluster (the dotted rings), where pointers reference other caching proxies within the same cluster. If a node finds a mapping in its local cluster (after step 2), its *get* finishes. Otherwise, it continues among its level-1 cluster (the solid rings), and finally, if needed, to any node within the global level-0 system.**

CoralCDN's inter-proxy transfers are optimized for locality, both from their use of parallel connections to other proxies and by the order in which neighboring proxies are contacted. The properties of Coral's hierarchical indexing ensures that the list of proxies returned by *get* will be sorted based on their cluster distance to the request initiator. Thus, proxies will attempt to contact level-2 neighbors before level-1 and level-0 proxies, respectively.

2.3.2 Rapid adaptation to flash crowds

Unlike many web proxies, CoralCDN is explicitly designed for flash-crowd scenarios. If a flash crowd suddenly arrives for a web object, proxies self-organize into a form of multicast tree for retrieving the object. Data streams from the proxies that started to fetch the object from the origin server to those arriving later. This limits concurrent object requests to the origin server upon a flash crowd.

CoralCDN provides such behavior by cut-through routing and optimistic references. First, CoralCDN's use of *cut-through* routing at each proxy helps reduce transmission time for larger files. That is, a proxy will upload portions of a object as soon as they are downloaded, not waiting until it receives the entire object. Second, proxies optimistically announce themselves as sources of content. As soon as a CoralCDN proxy begins receiving the first bytes of a web object—either from the origin or another proxy—it inserts a reference to itself into Coral with a short TTL (30 seconds). It continually renews this short-lived reference until either it completes the download (at which time it inserts a longer-lived reference[1]) or the download fails.

[1]The deployed system uses 2-hour TTLs for successful results (status codes of 200, 301, 302, etc.), and 15-minute TTLs for 403, 404, and other unsuccessful, non-transient results.

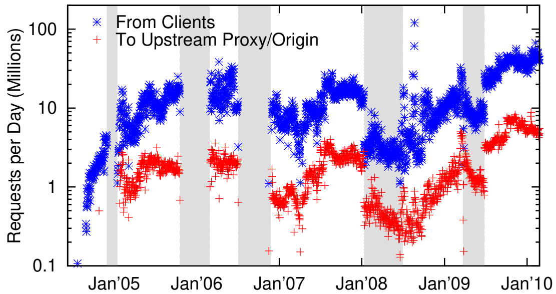

Figure 3: **Total HTTP requests per day during CoralCDN's deployment. Grayed regions correspond to missing or incomplete data.**

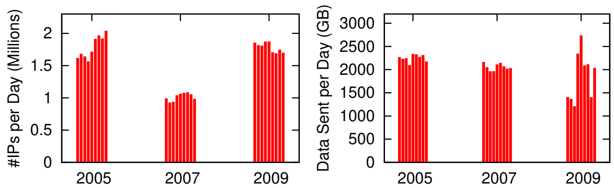

Figure 4: **CoralCDN usage: number of unique clients (left) and upload volume (right) for each day during August 9–18.**

2.4 Implementation and deployment

CoralCDN is composed of three stand-alone applications. The Coral daemon provides the distributed indexing layer, accessed over UNIX domain sockets from a simple client library linked into applications such as CoralCDN's HTTP proxy and DNS server. All three are written from scratch. Coral network communication uses Sun RPC over UDP, while CoralCDN proxies transfer content via standard HTTP connections. At initial publication [14], the Coral daemon was about 14,000 lines of C++, the DNS server 2,000 LOC, and the proxy 4,000 LOC. CoralCDN's implementation has since grown to around 50,000 LOC. The changes we later discuss help account for this increase.

CoralCDN typically runs on 300–400 PlanetLab servers (about 70–100 of which run its DNS server), spread over 100-200 sites worldwide. It avoids Internet2-only and commercial sites, the latter due to policy decisions that restrict their use for open services. CoralCDN uses no special knowledge of these machines' locations or connectivity (*e.g.*, GPS coordinates, routing information, etc.). Even though CoralCDN runs on a centrally-managed testbed, its mechanisms remain decentralized and self-organizing. The only use of centralization is for managing software and configuration updates and for controlling run status.

3 Analyzing CoralCDN's Usage

This section presents some HTTP-level data from CoralCDN's deployment and considers its implications.

3.1 System traces and traffic patterns

To understand some of the HTTP traffic patterns that CoralCDN sees, we analyzed several datasets in increasing depth. Figure 3 plots the total number of HTTP requests that the system received each day from mid-2004 through early 2010, showing both the number of HTTP requests from clients, as well as the number of requests issued to upstream CoralCDN peers or origin sites. The traces show common request rates for much of CoralCDN's deployment between 5 and 20 million HTTP requests per day, with more recent rates of 40–50 million daily requests.[2]

Year	Unique domains	Unique URLs	% URLs with 1 req	Reqs to most popular URL
2005	7881	577K	54%	697K
2007	21555	588K	59%	410K
2009	20680	1787K	77%	1578K

Figure 5: **CoralCDN traffic statistics for an arbitrary day (Aug 9).**

We examined three time periods from these logs in more depth, each consisting of HTTP traffic over the same nine-day period (August 9–18) in 2005, 2007, and 2009. CoralCDN received 15–25M requests during each day of these periods. Figure 4 plots the total number of unique client IP addresses from which these requests originated (left) and the aggregate amount of bandwidth uploaded (right). The traces showed 1–2 million clients per day, resulting in a few terabytes of content transferred. We will primarily use the 2009 trace, consisting of 209M requests, in later analysis. Figure 5 provides more information about the traffic patterns, focusing on the first day of each trace.

Figure 6 plots the distribution of requests per unique URL. We see that the number of requests per URL follows a Zipf-like distribution, as common among web caching and proxy networks [5]. Certain URLs are very popular—the so-called "head" of the distribution—such as the most popular one in the Aug-9-2009 trace, which received almost 1.6M requests itself. A large number of URLs—the distribution's "heavy tail"—receive only a single request.

The datasets also show stability in the most popular URLs and domains over time. In all three datasets, the most popular URL retained that ranking across all nine days. In fact, this URL in the 2007 and 2009 traces belonged to the same domain: a site that uses CoralCDN to distribute rule-set updates for the popular Firefox AdBlock browser extension. Exploring this further, Figure 7 uses the 2009 trace to plot the request rate per day for the most popular domains (taking the union of each day's most popular five domains resulted in nine unique domains). We see that six of the nine domains had stable traffic patterns—they were long-term CoralCDN "customers"—while three varied between two and six orders of magnitude per day. The traffic patterns that we see in these two figures have design implications, which we discuss next.

[2]The peak of 120M requests on August 21, 2008 corresponds to a short-lived experiment of an academic research project using CoralCDN as a key-value store [15].

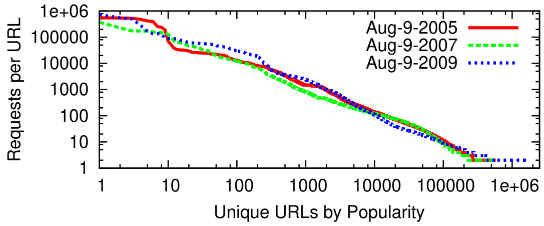

Figure 6: **Total requests per unique URL.**

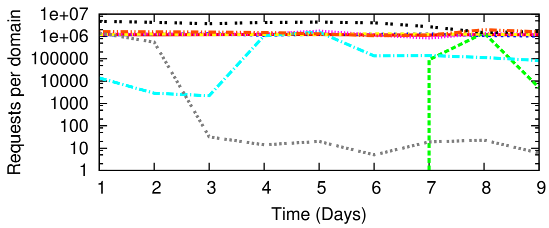

Figure 7: **Requests per top-5 domain over time (Aug 9-18, 2009).**

3.2 Implications of usage scenarios

For CoralCDN to help under-provisioned websites survive unexpected traffic spikes, it does not require any prior registration or authorization. Yet while such openness is necessary to enable even unmanaged websites to survive flash crowds, it comes at a cost: CoralCDN is used in a variety of ways that differ from this more narrow goal. This section considers how well CoralCDN's design is suited for its four main usage scenarios:

1. ***Resurrecting old content:*** Anecdotally, some clients attempt to use CoralCDN for long-term durability. One can download browser plugins that link to both CoralCDN and `archive.org` as potential sources of content when origin servers are unavailable.
2. ***Accessing unpopular content:*** CoralCDN's request distribution shows a heavy tail of unpopular URLs. Servers may Coralize URLs that few visit. And some clients use CoralCDN as a more traditional proxy, for (presumed) anonymity, censorship or filtering circumvention [32], or automated crawling.
3. ***Serving long-term popular content:*** Most requests are for a small set of popular objects. These objects, already widely cached across the network, belong to the stable set of customer domains that effectively use CoralCDN as a free, long-term CDN provider.
4. ***Surviving flash crowds to content:*** Finally, CoralCDN is used for its stated goal of enabling underprovisioned websites to withstand transient load spikes. Popular portals regularly link to Coralized URLs, and users post links in comments. Some sites even adopt dynamic and programmatic mechanisms to redirect requests to CoralCDN, based on observed load and request referrers. We discuss this further in §4.1.

Unfortunately, CoralCDN's design is not well-suited for the first three use cases.

Top URLs	Total Size (MB)	% of Total Reqs
0.01%	14	49.1%
0.1%	157	71.8%
1%	3744	84.8%
10%	28734	92.2%

Figure 8: **CoralCDN's working set size for its most popular URLs on Aug 9, 2009: A small percentage of URLs account for a large fraction of requests, yet they require relatively little storage to cache.**

Insufficient for resurrecting old content. CoralCDN is not designed for archival storage. Proxies do not proactively replicate content for durability, and unpopular content is evicted from proxy caches over time. Further, if content has an expiry time (default is 12 hours), a proxy will serve expired content for at most 24 hours after the origin fails. Still, some clients attempt to use CoralCDN for this purpose. This underscores a design tradeoff: In stressing support for flash crowds rather than long-term durability, CoralCDN devotes its resources to provide availability for content being actively requested. On the other hand, by serving expired content for a limited duration, CoralCDN can mask the *temporary* unavailability of an origin, at least for content already cached in its network.

Unnecessary for unpopular content. While proxies can discover even rare cached content, CoralCDN does not provide any benefit by serving such unpopular content: It does not reduce servers' load meaningfully, and it often results in higher client latency. As such, clients that use CoralCDN to avoid local filtering, circumvent geographic restrictions, or provide (minimal) anonymity may be better served by standard open proxies (that vanilla browsers can be configured to use) or through specialized tools such as Tor [12]. Yet, this type of usage persists—the long tail of Figure 6—and CoralCDN might then be better served with a different design for such traffic, *i.e.*, one that doesn't require a multi-hop, wide-area DHT lookup to complete before fetching content from the origin. For example, for its modest deployment on PlanetLab, each Coral node could maintain connectivity to all others and simply use consistent hashing for a global, one-hop DHT [17, 37]. Alternatively, Coral could only maintain connections with regional peers and eschew global lookups, a design which we evaluate further in §7.

Overkill for stably popular content, so far. For most of CoralCDN's traffic, cooperation is not needed: Figure 6 shows that a small number of URLs accounts for a large fraction of requests. We now measure their working set size in Figure 8, in order to determine how much storage is required to handle this traffic. We find that the most popular 0.01% of URLs account for more than 49% of the total requests to CoralCDN, yet require only 14 MB of storage. Each proxy has a 3.0 GB disk cache, managed using an LRU eviction policy. This is sufficient for serving nearly 85% of all requests from local cache.

70.4%	hit in local cache
12.6%	returned 4xx or 5xx error code
9.9%	fetched from origin site
7.1%	fetched from other CoralCDN proxy
↦ 1.7%	from level-0 cluster (global)
↦ 1.9%	from level-1 cluster (regional)
↦ 3.6%	from level-2 cluster (local)

Figure 9: **CoralCDN access ratios for content during Aug 9, 2009.**

These workload distributions support one aspect of CoralCDN's design: Content should be locally cached by the "forward" CoralCDN proxy directly serving end-clients, given that small to moderate size caches in these proxies can serve a very large fraction of requests. This differs from the traditional DHT approach of just storing data on a small number of globally-selected proxies, so-called "server surrogates" [8, 37].

If CoralCDN's working set can be fully cached by each node, we should understand how much cooperation is actually needed. Figure 9 summarizes the extent to which proxies cooperate when handling requests. 70% of requests to proxies are satisfied locally, while only 7% result in cooperative transfers. (The high rate of error messages is due to admission control as a means of bandwidth management, which we discuss in §5.2.) In short, at least for its current workload and environment, only a small fraction of CoralCDN's traffic uses its cooperation mechanisms.

A related result about the limits of cooperative caching had been observed earlier [38], but from the perspective of limited improvements in client-side hit rates. This is a significantly different goal from reducing server-side request rates, however: Non-cooperating groups of nodes would each individually request content from the origin.

This design trade-off comes down to the question of how much traffic is too much for origin servers. For moderately-provisioned origins, such as the customers of commercial CDNs, a caching system might only rely on local or regional cooperation. In fact, Akamai's network is designed precisely so: Nodes *within* each of its approximately 1000 clusters cooperate, but each cluster typically fetches content independently from origin sites [22]. To replicate such scenarios, Coral's clustering algorithms could be used to self-organize a network into local or regional clusters. It could thus avoid the manual configuration of Harvest [7] or colocated deployments of Akamai.

On the other hand, while cooperation is not needed for most traffic, CoralCDN's ability to react quickly to flash crowds—to offload traffic from a failing or oversubscribed origin—is precisely the scenario for which it was designed (and commercial CDNs are not). We consider these next.

Useful for mitigating flash crowds. CoralCDN's traces regularly show spikes in requests to different URLs. We find, however, that these flash crowds grow in popularity on the order of minutes, not seconds. There is a sufficiently long leading edge before traffic rises by several orders of magnitude, which has interesting implications.

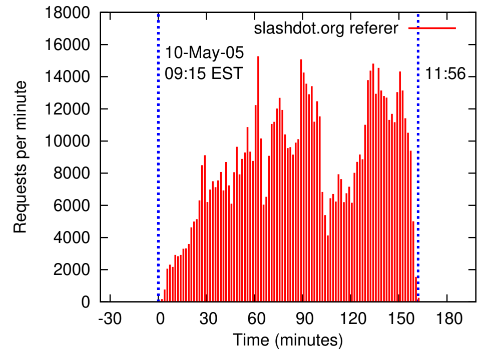

Figure 10: **Flash crowd to a Coralized URL linked to by Slashdot.**

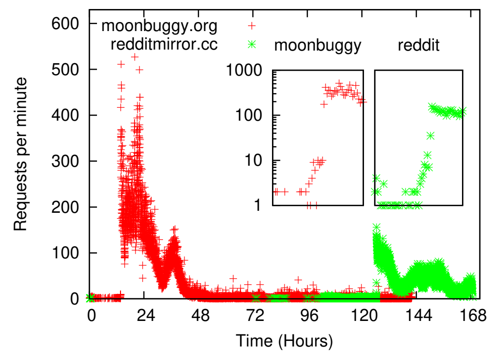

Figure 11: **Mini-flash crowds during August 2009 trace. Each datapoint represents a one-minute duration; embedded subfigures show request rates for the tens of minutes around the onset of flash crowds.**

Figures 10 and 11 show the request patterns of several flash crowds that CoralCDN experienced. The former was to a site linked to in a Slashdot article in May 2005. After rising, the Slashdot flash crowd lasted less than three hours in duration and came to an abrupt conclusion (perhaps as the story dropped off the website's main page). The latter, covering our August 2009 trace, shows spikes to the image cache of a less popular portal (`moonbuggy.org`), as well as to a well-publicized mirror for the collaboratively-filtered `reddit.com`, with another attenuated spike 24 hours later. The embedded graphs in Figure 11 depict the request rates around the onset of the traffic spike for a narrower range of time. All three flash crowds show that the initial rise took minutes.

For a more quantitative analysis of the frequency of flash crowds, we examined the prevalence of domains that experience a large increase in their request rates from one time period to the next. In particular, Figure 12 considers all five-second periods across the August 2009 ten-day trace. The left graph plots a complementary cumulative distribution function (CCDF) of the percentage of domains requested in each period that experience a 10- or 100-fold rate increase. The right graph plots the percentage of requests accounted for by these domains that experience orders-of-magnitude (OOM) increases. Sudden

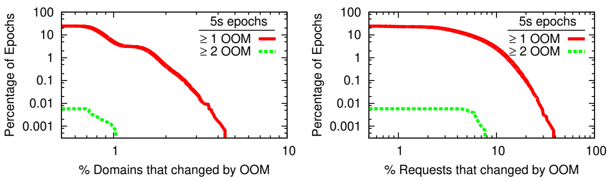

Figure 12: **CCDF of extent of flash-crowd dynamics in August 2009 trace. *Left* graph shows percentage of domains experiencing orders of magnitude (OOM) changes in request rates across five-second epochs. *Right* shows % requests for which these domains account.**

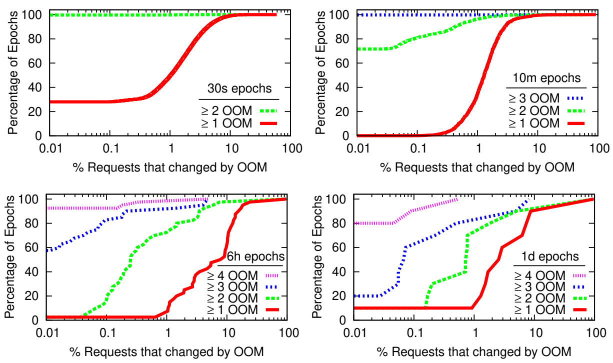

Figure 13: **CDFs of percentage of requests accounted for by domains experiencing order(s)-of-magnitude rate increases. Rate increases computed across epochs of 30 seconds (*top left*), 10 minutes (*top right*), six hours (*bottom left*), and one day (*bottom right*). Plots start on the y-axis with zero domains having such an increase, *e.g.*, 28% of 30s epochs have no domains with a ≥ 1 OOM rate increase.**

increases do exist, but they are rare. In 76% of 5s epochs, no domains experienced any 10-fold increase, while in 1% of epochs, 1.7% of domains (accounting for 12.9% of requests) increased by one order-of-magnitude. Larger dynamism was even more rare: only in 0.006% of epochs did there exist a domain that experienced a 100-fold increase in request rate. No three OOM increase occurred.

To further understand the precipitousness of "flash" crowds, Figure 13 extends this analysis across longer durations.[3] Among 30s epochs, 50% of epochs have at most 0.4% of domains experience a 10-fold increase in their rates (not shown), which account for a total of 1.0% of requests (top left). Only 0.29% of 30s epochs have *any* domains with more than a 100-fold rate increase. At 10-minute epochs, 28% of epochs have at least one domain that experiences a two OOM rate increase, while 0.21% have a domain with a three OOM increase. Still, these flash crowds account for a small fraction of total requests: Domains experiencing 100-fold increases accounted for at least 1% of all requests in only 3.8% of 10m epochs, and 10% of requests in 0.05% of epochs.

[3] To avoid overcounting unpopular domains, we do not count changes when the absolute number of requests to a domain in a given time period is less than some minimum amount, *i.e.*, 10 requests for 5s, 30s, and 10m periods, and 100 requests for 6h and 1d periods.

In short, this data shows that (1) only a small fraction of CoralCDN's domains experience large rate increases within short time periods, (2) those domains' traffic accounts for a small fraction of the total requests, and (3) any rate increases very rarely occur on the order of seconds.

This moderate adoption rate avoids the need to introduce even more aggressive content discovery algorithms. Simulated workloads in early experiments (Figure 4 of [14]) showed that under high concurrency, CoralCDN might issue several redundant fetches to an origin server due to a race-like condition in its lookup protocol. If multiple nodes concurrently *get* the same key which does not yet exist in the index, all concurrent lookups can fail and multiple nodes can contact the origin. This race condition is shared by most applications which use a distributed hash table (both peer-to-peer and datacenter services). But because these traces show that the arrival of user requests happens over a much longer time-scale than a DHT lookup, this race condition does not pose a significant problem.

Note that it is possible to mitigate this condition. While designing a network file system for PlanetLab that supported cooperative caching [2]—meant to quickly distribute a file in preparation for a new experiment—we sought to minimize redundant fetches to the file server. We extended Coral's insert operation to provide return status information, like test-and-set in shared-memory systems. A single *put+get* both returns the first values it encountered in the DHT, as well as inserts its own values at an appropriate location (for a new key, this would be at its closest node). This optimization comes at a subtle cost, however, as it now optimistically inserts a node's identity even before that proxy begins downloading the file! If the origin fetch fails—a greater possibility in CoralCDN's environment than with a managed file server—then the use of these index entries degrades performance. Thus, after using this *put+get* protocol in CoralCDN for several months during 2005, we discontinued its use.

CoralCDN's openness permits users to quickly leverage its resources under load, and its more complex coordination helps mitigate these flash crowds and mask temporary server unavailability. Yet this very openness led to varied usage, the majority of which does not require CoralCDN's more complex design. As we will see, this openness also introduces other problems.

4 Lessons for the Web

CoralCDN's naming technique provides an open API for CDN services that can transparently work for almost any website. Over the course of its deployment, clients and servers have used this API to adopt CoralCDN as an *elastic resource for content distribution.* Through completely automated means, work can be dynamically expanded out to use CoralCDN when websites require additional bandwidth resources, and it can be contracted back when flash crowds abate. In doing so, its use presaged the notion of "surge computing" with public cloud platforms. But these naming techniques and CoralCDN's open design introduce a number of web security problems, many of which are engendered by a *lack of explicitness for specifying protection domains*. We discuss these issues here.

4.1 An API for elastic CDN services

We believe that the central reason for CoralCDN's adoption has been its simple user interface and open design.

Interface design. While superficially obvious, CoralCDN's interface design achieves several important goals:

- ***Transparency:*** Work with *unmodified*, *unconfigured*, and *unaware* web clients and webservers.
- ***Deep caching:*** Retrieve embedded images or links automatically through CoralCDN when appropriate.
- ***Server control:*** Not interfere with sites' ability to perform usage logging or otherwise control how their content is served (*e.g.*, via CoralCDN or directly).
- ***Ad-friendly:*** Not interfere with third-party advertising, analytics, or other tools incorporated into a site.
- ***Forward compatible:*** Be amenable to future end-to-end security mechanisms for content integrity or other end-host deployed mechanisms.

Consider an alternative and even simpler interface design [11, 25, 29], in which one embeds origin URLs into the HTTP path, *e.g.*, `http://nyud.net/example.com/`. Not only is HTTP parsing simpler, but nameservers would not need to synthesize DNS records on the fly (unlike our DNS servers for `*.nyud.net`). Unfortunately, while this interface can be used to distribute individual objects, it fails on entire webpages. Any relative links would lack the `example.com` prefix that a proxy needs to identify its origin. One alternative might be to try to rewrite pages to add such links, although active content such as javascript makes this notoriously difficult. Further, such active rewriting impedes a site's control over its content, and it can interfere with analytics and advertisements.

CoralCDN's approach, however, interprets relative links with respect to a page's Coralized hostname, and thus transparently requests these objects through it as well. But all absolute URLs continue to point to their origin sites, and third-party advertisements and analytics remain largely unaffected. Further, as CoralCDN does not modify content, content also may be amenable to verification through end-to-end content signatures [30, 35].

In short, it was important for adoption that *site owners retain sufficient control over how their content is displayed and accessed*. In fact, our predicted usage scenario of sites publishing Coralized URLs proved to be less popular than that of dynamic redirection (which we did not foresee).

An API for dynamic adoption. CoralCDN was envisioned with manual URL manipulation in mind, whether by publishers editing HTML, users typing Coralized URLs, or third-parties posting links. After deployment, however, users soon began treating CoralCDN's interface as an API for accessing CDN services.

On the client side, these techniques included simple browser extensions that offer "right-click" options to Coralize links or that provide a link when a page appears unavailable. They ranged to more complex integration into frameworks like Firefox's Greasemonkey [21]. Greasemonkey allows third-party developers to write site-specific javascript code that, once installed by users, manipulates a site's HTML content (usually through the DOM interface) whenever the user accesses it. Greasemonkey scripts for CoralCDN include those that automatically rewrite links on popular portals, or modify articles to include tooltips or additional links to Coralized URLs. CoralCDN also has been integrated directly into a number of client-side software packages for podcasting.

The more interesting cases of CoralCDN integration are on the server-side. One common strategy is for the origin to receive the initial request, but respond with a 302 redirect to a Coralized URL. This can work well even for flash crowds, as the overhead of generating redirects is modest compared to that of actually serving the content.

Generating such redirects can be done by installing a server plugin and writing a few lines of configuration code. For example, the complete dynamic redirection rule using Apache's `mod_rewrite` plugin is as follows.

```
RewriteEngine on
RewriteCond %{HTTP_USER_AGENT} !^CoralWebPrx
RewriteCond %{QUERY_STRING} !(^|&)coral-no-serve$
RewriteRule ^(.*)$ http://%{HTTP_HOST}.nyud.net
                           %{REQUEST_URI} [R,L]
```

Still, redirection rules must be crafted carefully. In this example, the second line checks whether the client is a CoralCDN proxy and thus should be served directly. Otherwise, a redirection loop potentially could be formed (although proxies prevent this from happening by checking for potential loops and returning errors if one is found).

Amusingly, some early users during CoralCDN's deployment caused recursion in a different way—and a form of amplification attack—by submitting URLs with a long string of `nyud.net`'s appended to a domain. Before proxies checked for such conditions, this single request caused a proxy to issue a number of requests, stripping the last instance of `nyud.net` off in each iteration.

While the above rewriting rule applies for all requests, other sites incorporate redirection in more inventive ways, such as only redirecting clients arriving from particular high-traffic referrers:

```
RewriteCond %{HTTP_REFERER} slashdot\.org [NC,OR]
RewriteCond %{HTTP_REFERER} digg\.com [NC,OR]
RewriteCond %{HTTP_REFERER} blogspot\.com [NC]
```

And most interestingly, some sites have even combined such tools with server plugins that monitor server load and bandwidth use, so that their servers only start rewriting requests *under high load conditions*.

Websites therefore used CoralCDN's naming technique to leverage its CDN resources in an elastic fashion. Based on feedback from users, we expanded this "API" to give sites some simple control over how CoralCDN should handle their requests. For example, webservers can include `X-Coral-Control` response headers, which are saved as cache meta-data, to specify whether CoralCDN proxies should "redirect home" domains that exceed their bandwidth limits (per §5.2) or just return an error as is standard.

4.2 Security and resource protection

A number of security mechanisms curtailed the misuse of CoralCDN. We highlight the design principle for each.

4.2.1 Limiting functionality

CoralCDN proxies have only ever supported GET and HEAD requests. Many of the attacks for which "open" proxies are infamous [24] are simply not feasible. For example, clients cannot use CoralCDN to POST passwords for brute-force cracking. Proxies do not support CONNECT requests, and thus they cannot be used to send spam as SMTP relays or to forge "From" addresses in web mail. Proxies do not support HTTPS and they delete all HTTP cookies sent in headers. These proxies thus provide *minimal application functionality* needed to achieve their goals, which is cooperatively serving cacheable content.

CoralCDN's design had several unexpected consequences. Perhaps most interestingly, given CoralCDN's multi-layer caching architecture, attempting to crawl or brute-force attack a website via CoralCDN is quite slow. New or randomly-selected URLs first require a DHT lookup to fail, which serves to delay requests against an origin website, in much the same way that `ssh` "tarpits" delay responses to failed login attempts. In addition, because CoralCDN only handles explicit Coralized URLs, it cannot be used by simply configuring a vanilla browser's proxy settings. Further, CoralCDN cannot be used to anonymously launch attacks, as it eschews anonymity. Proxies use unique `User-Agent` strings ("`CoralWebPrx`") and include their identity in `Via` headers, and they report an instigating client's IP address to the origin server (in an `X-Forwarded-For` request header). We can only surmise whether the combination of these properties played some role, but CoralCDN has seen little abuse as a platform for proxying server attacks.

4.2.2 Curtailing excessive resource use

CoralCDN's major limiting resource is aggregate bandwidth. The system employs fair-sharing mechanisms to balance bandwidth consumption between origin domains, which we discuss further in §5.2. In addition to monitoring server-side consumption, proxies keep a sliding window of client-side usage. Not only do we seek to prevent excessive bandwidth consumption by clients, but also an excessive number of (even small) requests. These are caused typically by server misconfigurations that result in HTTP redirection loops (per §4.1) or by "bot" misuse as part of a brute-force attack. While CoralCDN's limited functionality mitigates such attacks, one notable brute-force login attempt took advantage of poor security at a top-5 website, which used cleartext passwords over GET requests.

Given both its storage and bandwidth limitations, CoralCDN enforces a maximum file size of 50 MB. This has generally prevented clients from using CoralCDN for video distribution, a pragmatic goal when deploying proxies on university-hosted PlanetLab servers. We found that sites attempted to circumvent these limits by omitting `Content-Length` headers (on connections marked as persistent and without chunked encoding). To ensure compliance, proxies now monitor ongoing transfers and halt (and blacklist) any ones that exceed their limits. This skepticism is needed as proxies interact with potentially untrusted servers, and thus must enforce *complete mediation* [33] to their resources (in this case, bandwidth).

4.2.3 Blacklisting domains and offloading security

We maintain a global blacklist for blocking access to specified origin domain names. Each proxy regularly fetches and reloads the blacklist. This is a practical, but not fundamental, necessity, employed to prevent CoralCDN's deployment sites from restricting its use. Parties that request blacklisting typically cite one of the following reasons.

Suspected phishing. Websites have been concerned that CoralCDN is—or will be confused with—a phishing site. After all, both appear to be "scraping" content and publish a simulacrum under an alternate domain. The difference, of course, is that CoralCDN is serving the site's content unmodified, yet the web lacks any protocol to authenticate the integrity of content (as in S-HTTP [30]) in order to verify this. As SSL only authenticates identity, *websites must typically include CDNs in their trusted computing base.*

Potential copyright violation. Typically following a DMCA take-down notice, third-parties report that copyrighted material may be found on a Coralized domain and want it blocked. This scenario is mitigated by CoralCDN's explicit naming—which preserves the name of the actual origin in question—and by its caching design. Once content is removed from an origin server, it is evicted automatically from CoralCDN in at most 24 hours. This is a natural implication of its goal of handling flash crowds, rather than providing long-term availability.

Circumventing access-control restrictions. Some domains mediate access to their website via IP-based authentication, whereby requests from particular IP prefixes are granted access. This practice is especially common for online academic journals, in order to provide easy access for university subscribers. But open proxies within whitelisted prefixes would enable external clients to circumvent these access-control restrictions.

By offloading policing to their customers, sites *unnecessarily enlarge their security perimeter to include their customer's networks*. This scenario is common yet unnecessary. Recall that CoralCDN proxies do not hide their identities, and they include the originating client's IP address in standard request headers. Thus, origin sites can retain IP-based authentication while verifying that a request does not originate from outside allowed prefixes.[4] Sites are just not making use of this information, and thus fail to properly mediate access to their protected resources.[5]

We did encounter some interesting attacks on our *domain*-based blacklists, akin to fast-flux networks. An adversary created dynamic DNS records for a random domain that pointed to the IP address of a target domain (an online academic journal). The random domain naturally was not blacklisted by CoralCDN, and the content was successfully downloaded from the origin target. Such a circumvention technique would not have worked if the origin site checked either proxy headers (as above) or even just the `Host` field of the HTTP request. The `Host` corresponded to the fast-flux attack domain, not that of the journal. Again, this security hole demonstrates a lack of explicit verification and fail-safe defaults [33].

4.3 Security and naming conflation

We argued that CoralCDN's naming provided a powerful API for accessing CDN services. Unfortunately, its technique has serious implications as the Web's Same Origin Policy (SOP) *conflates naming with security*.

Browsers use domain names for three purposes. (1) Domains specify *where* to retrieve content after they are resolved to IP addresses, precisely how CoralCDN enacts its layer of indirection. (2) Domains provide a human-readable name for *what administrative entity* a client is interacting with (*e.g.*, the "common name" identified in SSL server certificates). (3) Domains specify *what security policies* to enforce on web objects and their interactions.

The Same Origin Policy specifies how scripts and instructions from an origin domain can access and modify browser state. This policy applies to manipulating cookies, browser windows, frames, and documents, as well as to accessing other URLs via an XmlHttpRequest. At its simplest level, all of these behaviors are only allowed between resources that belong to an identical origin domain. This provides security against sites accessing each others' private information kept in cookies, for example. It also prevents websites that run advertisements (such as Google's AdSense) from easily performing click fraud to pay themselves advertising dollars by programmatically "clicking" on their site's advertisements.[6]

One caveat to the strict definition of an identical origin [18] is that it provides an exception for domains that share the same `domain.tld` suffix, in that `www.example.com` can read and set cookies for `example.com`. This has bad implications for CoralCDN's naming strategy. When `example.com` is accessed via CoralCDN, it can manipulate all `nyud.net` cookies, not just those restricted to `example.com.nyud.net`.[7] Concerned with the potential privacy violations from this scenario, CoralCDN deletes all cookies from headers.

Unfortunately, many websites now manage cookies via javascript, so cookie information can still "leak" between Coralized domains on the browser. This happens often without a site's knowledge, as sites commonly use a URL's `domain.tld` without verifying its name. Thus, if the Coralized `example.com` writes `nyud.net` cookies, these will be sent to `evil.com.nyud.net` if the client visits that webpage. Honest CoralCDN proxies will delete these cookies in transit, but attackers can still circumvent this problem. For example, when a client visits `evil.com.nyud.net`, javascript from that page can access `nyud.net` cookies, then issue a XmlHttpRequest back to `evil.com.nyud.net` with cookie information embedded in the URL. Similar attacks are possible against other uses of the SOP, especially as it relates to the ability to access and manipulate the DOM. Note that these attack vectors exist even while CoralCDN operates on fully-trusted nodes, let alone more peer-to-peer environments!

Rather than conclude that CoralCDN's domain manipulation is fundamentally flawed, we argue that better adherence to security principles is needed. Websites are partially at fault because they default access to `domain.tld` suffixes too readily, as opposed to stripping the minimal number of domain prefixes: a violation of the principle of least information. An alternative solution that embraces least

[4] This does not address the corner case in which the original request comes from an IP address within that prefix, while subsequent ones that access the then-cached content do not. This can be handled typically by marking content as not cacheable, or by having a proxy include headers that explicitly specify its client population (*i.e.*, as "open" or by IP prefix).

[5] One might argue that sites use a pure IP-based filtering approach given its ability to be implemented in layer-3 front-end load balancers. But this is not a simple firewall problem, as sites also permit access for individual users that login with the appropriate credentials. The sites with which we communicated implemented such authorization logic either directly in webservers or in complex, layer-7 front-end appliances.

[6] This is prevented because advertisements like AdSense load in an `iframe` that the parent document—the third-party website that stands to gain revenue—cannot access, as the frame belongs to a different domain.

[7] Commercial CDNs like Akamai are typically not susceptible to such attacks, as they generally use a separate top-level domains for each customer, as opposed to CoralCDN's suffix-based approach. Unlike CoralCDN's zero configuration, however, such designs require that origins preestablish an operational relationship with their CDN provider and point their domain to the CDN service (*e.g.*, by aliasing their domain to the CDN through CNAME records in DNS).

privilege (and has much better incremental deployability) would be to *allow sources of content to explicitly constrain default security policies*. As one simple example, when serving content for some `origin.tld`, proxies could use HTTP response headers to specify that the most permissive domain should be `origin.tld.domain.tld`, not their own `domain.tld`. Interestingly, HTML 5, Flash, and various javascript hacks [6] are all exploring methods to *expand* explicit cross-domain communication.[8] Both proposals avow that the SOP is insufficient and should be adapted to support more flexible control through explicit rules; ours just views its corner cases as too permissive, while the other views its implications as too restrictive.

5 Lessons for CDNs

Unlike most commercial counterparts, CoralCDN is designed to interact with overloaded or poorly-behaving origin servers. Further, while commercial systems will grow their networks based on expected use (and hence revenue), the CoralCDN deployment is comprised of volunteer sites with fixed, limited bandwidth. This section describes how we adapted CoralCDN to satisfy these realities.

5.1 Designing for faulty origins

Given its design goals, CoralCDN needs to react to non-crash failures at origin servers as the rule, not the exception. Thus, one design philosophy that has come to govern CoralCDN's behavior is that *proxies should accept content conservatively and serve results liberally*.

Consider the following, fairly common, situation. A portal runs a story that links to a third-party website, driving a sudden influx of readers to this previously unpopular site. A user then posts a Coralized link to the third-party site as a "comment" to the portal's story, providing an alternate means to fetch the content.

Several scenarios are possible. (1) The website's origin server becomes unavailable before any proxy downloads its content. (2) CoralCDN already has a copy of the content, but requests arrive to it after the content's *expiry* time has passed. Unfortunately, subsequent HTTP requests to the origin webserver result in failures or errors. (3) Or, CoralCDN's content is again expired, but subsequent requests to the origin yield only partial transfers. CoralCDN employs different mechanisms to handle these failures.

Cache negative service results (#1). CoralCDN may be hit with a flood of requests for an inaccessible URL, *e.g.*, DNS resolution fails, TCP connections timeout, etc. For these situations, proxies maintain a local negative result cache about repeated failures. Otherwise, both proxies and their local DNS resolvers have experienced resource exhaustion, given flash crowds to apparently dead sites. (While negative result caching has also long been part of some DNS implementations [19], it is not universal and does not extend to TCP or application-level failures.) While more a usability issue, CoralCDN still receives requests for some Coralized URLs several years after their origins became unavailable.

Serve stale content if origin faulty (#2). CoralCDN seeks to avoid replacing good content with bad. As its proxies mostly obey content expiry times specified in HTTP headers,[9] if cached content expires, proxies perform a conditional request (`If-Modified-Since`) to revalidate or update expired content. Overloaded origin servers might fail to respond or might return some temporary error condition (data in §7 shows this to occur in about 0.5% of origin requests). Rather than retransmit this error, CoralCDN proxies return the stale content and continue to retain it for future use (for up to 24 hours after it expires).

Prevent truncations through whole-file overwrites (#3). Rather than not responding or returning an error, what if a revalidation yields a truncated transfer? This is not uncommon during a flash crowd, as a CoralCDN proxy will be competing for a webserver's resources. Rather than have proxies lose stale yet complete versions of objects, proxies implement *whole-file overwrites* in the spirit of AFS [16]. Namely, if a valid web object is already cached, the new version is written to a temporary file. Only after the new version completes downloading and appears valid (based on `Content-Length`) will a proxy replace the old one.

These approaches are not fail-proof, limited by both semantic ambiguity in status directives and inaccuracies with their use. In terms of ambiguity, does a 403 (Forbidden) response code signify that a publisher seeks to make the content unavailable (permanent), or is it caused by a website surpassing its daily bandwidth limits and having requests rejected (temporary)? Does a 404 (File Not Found) code indicate whether the condition is permanent (due to a DMCA take-down notice) or temporary (from a PHP or database error)? On the other hand, the application of status directives can be flawed. We often found websites to report human-readable errors in HTML body content, but with an HTTP status code of 200 (Success). This scenario leads CoralCDN to replace valid content with less useful information. We hypothesize that bad defaults in scripting languages such as PHP are partially to blame. Instead of being fail-safe, the response code defaults to success.

Even if transient errors were properly identified, for how long should CoralCDN serve expired content? HTTP lacks

[8] This is in reaction to the common practice of inserting third-party objects into a document's namespace via `<script>`—and thus sacrificing security protections—as the SOP does not permit a middle ground.

[9] Proxies in our deployment are configured with a *minimum* expiry time of some duration (five minutes), and thus do not recognize `No-Cache` directives as such. Because CoralCDN does not support cookies, SSL bridging, or POSTs, however, many of the privacy concerns associated with caching such content are alleviated.

the ability to specify explicit policy for handling expired content. Akamai defaults to a fail-safe scenario by not returning stale content [22], while CoralCDN seeks to balance this goal with availability under server failures. As opposed to only using the system-wide default of 24 hours, CoralCDN recently enabled its users to explicitly specify their policy through `max-stale` response headers.[10]

These examples all point to another lesson that governs CoralCDN's proxy design: *Maintain the status quo unless improvements are possible.*

Decoupling service dependencies. A similar theme of only improving the status quo governs CoralCDN's management system. CoralCDN servers query a centralized management point for a number of tasks: to update their overall run status, to start or stop individual service components (HTTP, DNS, DHT), to reinstall or update to a new software version, or to learn shared secrets that provide admission control to its DHT. Although designed for intermittent connectivity, one of CoralCDN's significant outages came when the management server began misbehaving and returning unexpected information. In response, we adopted what one might call *fail-same behavior* that accepts updates conservatively, an application of decoupling techniques from fault-tolerant systems. Management information is stored durably on servers, maintaining their status-quo operation (even across local crashes) until well-formed new instructions are received.

5.2 Managing oversubscribed bandwidth

While commercial CDNs and computing platforms often respond to oversubscription by acquiring more capacity, CoralCDN's deployment on PlanetLab does not have that luxury. Instead, the service must manage its bandwidth consumption within prescribed limits. This adoption of bandwidth limits was spurred on by administrative demands from its deployment sites. Following the Asian tsunami of December 2004, and with YouTube yet to be created, CoralCDN distributed large quantities of amateur videos of the natural disaster. With no bandwidth restrictions on PlanetLab at the time, CoralCDN's network traffic to the public Internet quickly spiked. PlanetLab sites threatened to pull their servers off the network if such use could not be curtailed. It was agreed that CoralCDN should restrict its usage to approximately 10 GB per day per server (*i.e.*, per PlanetLab sliver).

Several design options exist for limiting bandwidth consumption. A proxy could simply shut down after exceeding a configured daily capacity (as supported by Tor [12]). Or it could rate-limit its traffic to prevent transient congestion (as done by BitTorrent and Tor). But as CoralCDN

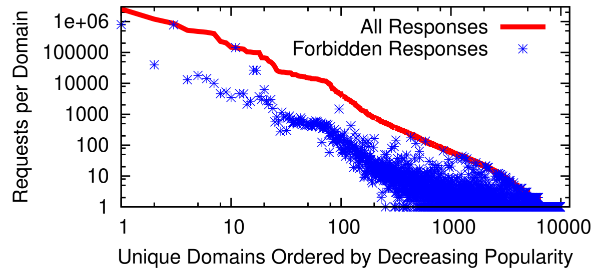

Figure 14: **Requests per domain and number of 403 rejections.**

primarily provides a service for websites, as opposed to clients, we chose to allocate its limited bandwidth in a way that both preserves some notion of *fairness* across its customer domains and maintains its central goal of handling flash crowds. The technique we developed is more broadly applicable than just PlanetLab and federated testbeds: to P2P deployments where users run peers within resource containers, to multi-tenant datacenters sharing resources between their own services, or to commercial hosting environments using billing models such as 95th-%ile usage.

Providing per-domain fairness might be resource intensive or difficult in the general case, given that CoralCDN interacts with 10,000s of domains each day, but our highly-skewed workloads greatly simplify the necessary accounting. Figure 14 shows the total number of requests per domain that CoralCDN received over one day (the solid top line). The distribution clearly has some very popular domains—the most popular one (a Tamil clone of YouTube) received 2.6M requests—while the remaining distribution fell off in a Zipf-like manner. (Note that Figure 6 was in terms of unique URLs, not unique domains.) Given that CoralCDN's traffic is dominated by a limited number of domains, its mechanisms can serve mainly to reject requests for (*i.e.*, perform admission control on) these bandwidth hogs. Still, CoralCDN should differentiate between peak limits and steady-state behavior to allow for flash crowds or changing traffic patterns.

To achieve these aims, each CoralCDN proxy implements an algorithm that attempts to simultaneously (1) provide a hard-upper limit on peak traffic per hour (configured to 1000 MB per hour per proxy), (2) bound the expected total traffic per epoch in steady state (400 MB per hour per proxy), and (3) bound the steady-state limit per domain. As setting this last limit statically—such as $1/k$-th of the total traffic if there are k popular domains—would lead to good fairness but poor utilization (given the non-uniform distribution across domains), we dynamically adjust this last traffic limit to balance this trade-off.

During each hour-long epoch, a proxy records the total number of bytes transmitted for each domain. It also calculates domains' average bandwidth as an exponentially-weighted moving average (attenuated over one week), as well as the total average consumption across all domains. This long attenuation period provides long-term fairness—

[10] HTTP/1.1 supports max-stale *request* headers, although we are not aware of their use by any HTTP clients. Further, as proxies often evict expired content from their caches, it is unclear whether such request directives can be typically satisfied.

and most consumption is long-term, as shown in Figure 7—but also emphasizes support for short-term flash crowds. Across epochs, bandwidth usage is only tracked, and durably stored, for the top-100 domains. If a domain is not currently one of the top-100 bandwidth consumers, its historical average bandwidth is set to zero (providing additional leeway to sites experiencing flash crowds).

When a requested domain is over its hourly budget (case 3 above), CoralCDN proxies respond with 403 (Forbidden) messages. If instead the proxy is over its peak or steady-state limit calculated over all domains (cases 1 or 2 above), then the proxy redirects the client back to the origin site, and the proxy temporarily makes itself unavailable for new client requests, which would be rejected anyway.[11]

By applying these mechanisms, CoralCDN reduces its bandwidth consumption to manageable levels. While its demand sometimes exceeds 10 TBs per day (aggregate across all proxies), its actual HTTP traffic remains steady at about 2 TB per day after rejecting a significant number of requests. The scatter plot in Figure 14 shows the number of requests resulting in 403 responses per domain, most due to these admission control mechanisms. We see how variances in domains' object sizes yield different rejection rates. The second-most popular domain serves mostly images smaller than 10 KB and experiences a rejection rate of 3.3%. Yet the videos of the third-most popular domain—user-contributed screensavers of fractal flames—are typically 5 MB in size, leading to an 89% rejection rate.

Note that we could significantly curtail the use of CoralCDN as a long-term CDN provider (see §3.2) through simple changes to these configuration settings. A low steady-state limit per domain, coupled with a greater weight on a domain's historic averages, devotes resources to flash-crowd relief at the exclusion of long-term consumption.

Admittedly, CoralCDN's approach penalizes an origin site with more regional access patterns. Bandwidth accounting and admission control is performed independently on each node, reflecting CoralCDN's lack of centralization. By not sharing information between nodes (provided that DNS resolution preserves locality), a site with regional interest can be throttled before it reaches its fair share of global capacity. While this does not pose an operational problem for CoralCDN, it is an interesting research problem to perform (approximate) accounting across the network that is both decentralized and scalable. Distributed Rate Limiting [28] considered a related problem, but focused on instantaneous limits (*e.g.*, Mbps) instead of long-term aggregate volumes and gossiped state that is linear in both the number of domains and nodes.

[11] If clients are redirected back to the origin, a proxy appends the query-string `coral-no-serve` on the location URL returned to the client. Origins that use redirection scripts with CoralCDN check for this string to prevent loops, per §4.1. Although not the default, operators of some sites preferred this redirection home even if their domain was to blame (a policy they can specify through a `X-Coral-Control` response header).

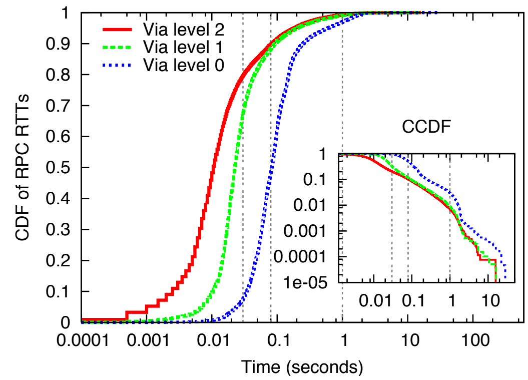

Figure 15: **RPC RTTs to various levels of Coral's DHT hierarchy.**

5.3 Managing performance jitter

Running on an oversubscribed deployment platform, CoralCDN developed several techniques to better handle latency variations. With PlanetLab services facing high disk, memory, and CPU contention, and sometimes additional traffic shaping in the kernel, applications can face both performance jitter and prolonged delays. These performance variations are not unique to PlanetLab, and they have been well documented across a variety of settings. For example, Google's MapReduce [10] took runtime adaption of cluster query processing [3] to the large-scale, where performance variations even among homogeneous components required speculative re-execution of work. More recently, studies of a MapReduce clone on Amazon's EC2 underscored how shared and virtualized platforms provide new performance challenges [39].

CoralCDN saw the implications of performance variations most strikingly with its latency-sensitive self-organization. For example, Coral's DHT hierarchy was based on nodes clustering by network RTTs. A node would join a cluster provided some minimum fraction (85%) of its members were below the specified threshold (30 ms for level 2, 80 ms for level 1). Figure 15 shows the RTTs for RPC between Coral nodes, broken down by levels (with vertical lines added at 30ms, 80ms, and 1s). While the clustering algorithms achieve their goals and local clusters have lower RTTs, the heavy tail in all CDFs is rather striking. Fully 1% of RPCs took longer than 1 second, even within local clusters. Coral's use of concurrent RPCs during DHT operations helped mask this effect.

Another lesson from CoralCDN's deployment was the need for *stability in the face of performance variations*. This translated to the following rule in Coral. A node would switch to a smaller (and hence less attractive) cluster only if fewer than 70% of a cluster's members now satisfy its threshold, and form a singleton only if fewer than 50% of neighbors are satisfactory. In other words, the barrier to enter a cluster is high (85%), but once a member, it's easier to remain. Before leveraging this form of hysteresis, cluster oscillations were much more common, which led

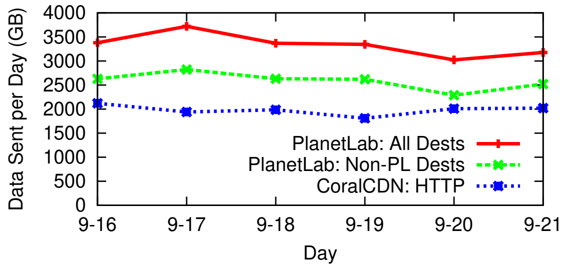

Figure 16: **Comparison of PlanetLab's accounting of all upstream traffic, PlanetLab's count to non-PlanetLab destinations, and CoralCDN's accounting through HTTP logs.**

to many stale DHT references. A related use of hysteresis within self-organizing systems helped improve virtual network coordinate systems for both PlanetLab [26] and Azureus [20], as well as failure recovery in Bamboo [31].

6 Lessons for Platforms

With the growth of virtualized hosting and cloud deployments, Internet services are increasingly running on third-party infrastructure. Motivated by CoralCDN's deployment on PlanetLab, we discuss some benefits from improving an application's visibility into and control over its lower layers. We first revisit CoralCDN's bandwidth management from the perspective of fine-grained service differentiation, then describe tackling its fault-tolerance challenge with adequate network support.

6.1 Exposing information and expressing preferences across layers

We described CoralCDN's bandwidth management as self-regulating, which works well in trusted environments. But many resource providers would rather *enforce* restrictions than assume applications behave well. Indeed, in 2006, PlanetLab began enforcing average daily bandwidth limits per node per service (*i.e.*, per PlanetLab "sliver"). When a sliver hits 80% of its limit—17.2 GB/day from each server to the public Internet—the kernel begins enforcing bandwidth caps (using Linux's Hierarchical Token Bucket scheduler) as calculated over five-minute epochs.

We now have the possibility of two levels of bandwidth management: admission control by CoralCDN proxies and rate limiting by the underlying hosting platform. Interestingly, even though CoralCDN uses a relatively conservative limit for itself (10 GB/day per sliver), it still surpasses the 80% mark (13.8 GB) on 5–10 servers per day (out of its 300-400 servers). The main cause of this overage is that, while CoralCDN counts only successful HTTP responses, its hosting platform accounts for all traffic—HTTP, DNS, DHT RPCs, log transfers, packet headers, retransmissions, etc.—generated by its sliver. Figure 16 shows the difference in these recorded values for the week of Sept 16, 2009. We see that kernel statistics were 50%-90% higher than CoralCDN's accounting. This problem of accurate accounting is a general one, as it is difficult or expensive to collect such data in user-space.[12] And even accurate information does not prevent CoralCDN's managed HTTP traffic from competing for network resources with the rest of its sliver's unmanaged traffic.

We argue that hosting platforms should provide better visibility and control. First, these platforms should export greater information to higher levels, such as their current measured resource consumption in a machine-readable format and in real time. Second, these platforms should allow applications to push policies into lower levels, *i.e.*, an application's explicit preferences for handling different classes of resources. For the specific case of network resources, the platform kernel could apply priorities on a granularity finer that just per-sliver, akin to a form of end-host DiffServ; CoralCDN would prioritize DNS and DHT traffic over HTTP traffic, in turn over log maintenance.

Note that we are concerned with a different type of resource management than that provided by VM hypervisors or kernel resource containers [4]. Those systems focus on *short-term* resource isolation or prioritized scheduling between applications, and typically reason about *coarse-grain* VM-level resources. Our focus instead is on *long-term* resource accounting. PlanetLab is not unique here; commercial cloud-computing providers such as Amazon and Rackspace use long-term resource accounting for billing purposes. (In fact, Amazon just launched its CloudWatch service in June 2009 to expose real-time resource monitoring on a coarser-grain per-VM basis [1].) Thus, providing greater visibility and control would be useful not only for deploying applications on platforms with hard constraints (*e.g.*, PlanetLab), but also for managing applications on commercial platforms so as to minimize costs (*e.g.*, in both metered and 95th-%ile billing scenarios).

6.2 Providing support for fault-tolerance

A central reliability issue in CoralCDN is due to its bootstrapping problem: To initially resolve a Coralized URL with no prior knowledge of system participants, a client's resolver must contact one of only 10–12 CoralCDN nameservers registered with the `.net` gTLD servers. If one of these nameservers fails—each IP address represents a static PlanetLab server—clients experience long DNS timeouts. Thus, while CoralCDN *internally* detects and reacts quickly to failure, the same rapid recovery is not enjoyed by its primary nameservers registered *externally*. And once legacy clients bind to a particular proxy's IP address—*e.g.*, web browsers cache name-to-IP mapping to prevent certain types of "rebinding" attacks on the

[12] In fact, even Akamai servers only use an estimate of bandwidth consumption (their so-called "fully-weighted bits") when calculating server load [22]. Only more recently did PlanetLab expose kernel accounting.

Same Origin Policy [9]—CoralCDN cannot recover for this client if that proxy fails.

While certainly observed before, CoralCDN's reliability challenge underscores the limits of purely application-layer recovery, especially as it relates to bootstrapping. In the context of DNS-based bootstrapping, several possibilities exist, including (1) dynamically updating root nameservers to reflect changes, *e.g.*, via the rarely-supported RFC2136 [36], (2) announcing IP anycast addresses via BGP or OSPF, or (3) using transparent network-layer failover between colocated nameservers (*e.g.*, ARP spoofing or VIP/DIP load balancers). IP-level recovery between proxies has its own solutions, but most commonly rely on colocated servers in LAN environments. None of these suggestions are new ones, but they still present a higher barrier to entry; PlanetLab did not have any available to it.

Deployment platforms should strive to provide or expose such network functionality to their services. Amazon EC2's launch of Elastic IP Addresses in March 2008, for example, hid the complexity of ARP spoofing for VM environments. The further development of such support should be an explicit goal for future deployment platforms.

7 Conclusions and Looking Forward

Our retrospective on CoralCDN's deployment has a rather mixed message. We view the adoption of CoralCDN as a successful proof-of-concept of how users can and will leverage open APIs for CDN services. But many of its architectural features were over-designed for its current environment and with its current workload: A much simpler design could have sufficed with probably better performance to boot.

That said, it is a entirely different question as to whether CoralCDN provides a good basis for designing an Internet-scale cooperative CDN. The service remained tied to PlanetLab because we desired a solution that was backwards compatible with both unmodified clients and servers. Running on untrusted nodes seemed imprudent at best given our inability to provide end-to-end security checks. We have shown, however, that even running CoralCDN on fully trusted nodes introduces some security concerns. So, if we dropped the goal of full backwards compatibility, what minimal changes could better support more open, flexible infrastructure?

Naming. CoralCDN's naming provided a layer of indirection for composing two loosely-coupled Internet services. In fact, one could compose longer series of services that each offer different functionality by simply chaining together their domain names. While this technique would not be safe under today's Same Origin Policy, we showed in §4.3 how a trusted proxy could constrain the default security policy. For a participating origin server with an untrusted CDN, the origin should specify (and sign) its minimally required domain suffix of `origin.tld.*`.

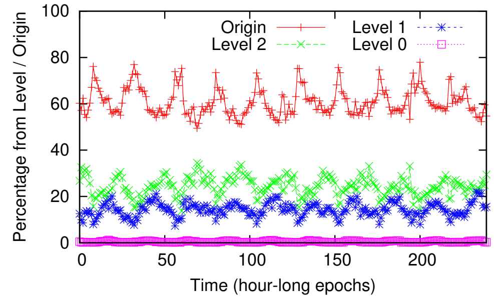

Figure 17: **Percentage of a proxy's upstream requests satisfied by origin and by peers at various clustering levels when *regional cooperation* is used, *i.e.*, level-0 peers only serve as a failover from a faulty origin. Dataset covers 10-day period from December 9–19, 2009.**

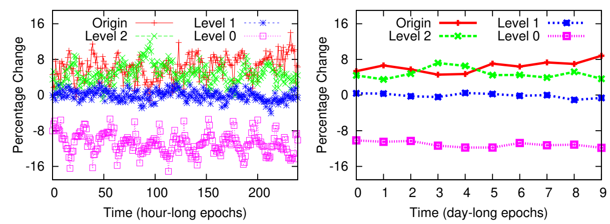

Figure 18: **Change in percentage between regional cooperation policy (Figure 17) and CoralCDN's traditional global peering. Positive values correspond to increased hit rates in regional peering.**

Content Integrity. Today's CDNs are full-fledged members of a website's trusted computing base. They have free reign to return modified content. Often, they can even programmatically read and modify any content served *directly* from a customer website to its clients (either by serving embedded `<script>`'s or by playing SOP tricks while masquerading as their customer behind a DNS alias). To provide content delivery via untrusted nodes, the natural solution is an HTTP protocol that supports end-to-end signatures for content integrity [30]. In fact, even a browser extension would suffice to deploy such security [35].

Fine-Grain Origin Control. A tension in this paper is between client latency and server load, underscored by our varied usage scenarios. An appropriate strategy for interacting with a well-provisioned server is a minimal attempt at cooperation before contacting the origin. Yet, an oversubscribed server wants its clients to make a maximal effort at cooperation. So far, proxies have used a "one-size-fits-all" approach, treating all origins as if they were oversubscribed. Instead, much as they have adopted dynamic URL rewriting, origin domains can signal a CoralCDN proxy about their desired policy in-band. At a high-level, this argues for a richer API for elastic CDN services.

To explore the effect of *regional cooperation*, we changed the default lookup policy on about half the deployed CoralCDN proxies since September 2009. If re-

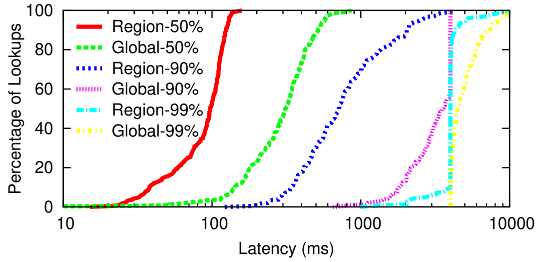

Figure 19: **CDF of median, 90th percentile, and 99th percentile lookup latency (over all hour-long epochs of Dec 9–19, 2009), comparing regional and global cooperation policies. Individual lookups were configured with a five-second timeout.**

quested content is not already cached locally, these proxies only perform lookups within local and regional clusters (level 2 and 1) before contacting the origin. For proxies operating under such a policy, Figure 17 shows the percentage of upstream requests that were satisfied by the origin and at different levels of clusters. Figure 18 depicts the *change* in behavior compared to the traditional global lookup strategy, showing that the 10–12% of requests that had been satisfied by level-0 proxies shifted to higher hit rates at both the origin and local proxies.[13] This change was associated with an order-of-magnitude latency improvement for the Coral lookup, shown in Figure 19. The global index still provides some benefit to the system, however, as per Figure 17, it satisfies an average of 0.56% of requests (stddev 0.51%) that failed over from origin servers. In summary, system architectures like CoralCDN can support different policies that trade-off server load for latency, yet still mask temporary failures at origins.

While perhaps imperfectly suited for a smaller-scale platform like PlanetLab, CoralCDN's architecture provides interesting self-organizational and hierarchical properties. This paper discussed many of the challenges—in security, availability, fault-tolerance, robustness, and, perhaps most significantly, resource management—that we needed to address during its five-year deployment. We believe that its lessons may have wider and more lasting implications for other systems as well.

Acknowledgments. We are grateful to David Mazières for his significant contributions and support during the design and operation of CoralCDN. We also thank Larry Peterson and the entire PlanetLab team for providing a deployment platform for CoralCDN. CoralCDN was originally funded as part of Project IRIS (supported by the NSF under Coop. Agreement #ANI-0225660) and recently under NSF Award #0904860. Freedman was also supported by an NDSEG Fellowship. More information about CoralCDN can be found at `www.coralcdn.org`.

[13]These graphs also show interesting diurnal patterns, related to a default expiry time of 12 hours for content.

References

[1] Amazon CloudWatch. http://aws.amazon.com/cloudwatch/, 2009.

[2] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: Scaling file servers via cooperative caching. In *NSDI*, 2005.

[3] R. H. Arpaci-Dusseau. Run-time adaptation in river. *ACM Trans. Computer Systems*, 21(1), 2003.

[4] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, 1999.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM*, 1999.

[6] J. Burke. Cross domain frame communication with fragment identifiers. http://tagneto.blogspot.com/, June 6, 2006.

[7] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical Internet object cache. In *USENIX Annual*, 1996.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[9] D. Dean, E. W. Felten, and D. S. Wallach. Java security: from hotjava to netscape and beyond. In *Symp. Security and Privacy*, 1996.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[11] Dijjer. http://code.google.com/p/dijjer/, 2010.

[12] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *USENIX Security*, 2004.

[13] M. J. Freedman and D. Mazières. Sloppy hashing and self-organizing clusters. In *IPTPS*, 2003.

[14] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *NSDI*, 2004.

[15] E. Freudenthal, D. Herrera, S. Gutstein, R. Spring, and L. Longpre. Fern: An updatable authenticated dictionary suitable for distributed caching. In *MMM-ACNS*, 2007.

[16] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Computer Systems*, 6(1):51–81, 1988.

[17] 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*, 1997.

[18] D. Kristol and L. Montulli. RFC 2965: HTTP state management mechanism, 2000.

[19] A. Kumar, J. Postel, C. Neuman, P. Danzig, and S. Miller. RFC 1536: Common DNS errors and suggested fixes, 1993.

[20] J. Ledlie, P. Gardner, and M. Seltzer. Network coordinates in the wild. In *NSDI*, 2007.

[21] A. Lieuallen, A. Boodman, and J. Sundstrom. Greasemonkey. https://addons.mozilla.org/en-US/firefox/addon/748, 2010.

[22] B. Maggs. Personal communication, 2009.

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

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

[25] K. Park and V. S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *NSDI*, 2006.

[26] P. Pietzuch, J. Ledlie, and M. Seltzer. Supporting network coordinates on planetlab. In *WORLDS*, 2005.

[27] PlanetLab. http://www.planet-lab.org/, 2010.

[28] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. Snoeren. Cloud control with distributed rate limiting. In *SIGCOMM*, 2007.

[29] RedSwoosh. http://www.akamai.com/redswoosh, 2009.

[30] E. Rescorla and A. Schiffman. RFC 2660: The secure hypertext transfer protocol, 1999.

[31] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *USENIX Annual*, 2004.

[32] H. Roberts, E. Zuckerman, and J. Palfrey. 2007 circumvention landscape report: Methods, uses, and tools. Technical report, Berkman Center for Internet & Society, Harvard, 2009.

[33] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 93(9), 1975.

[34] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Trans. Network.*, 11(1):17–32, 2003.

[35] J. Terrace, H. Laidlaw, H. E. Liu, S. Stern, and M. J. Freedman. Bringing P2P to the Web: Security and privacy in the Firecoral network. In *IPTPS*, 2009.

[36] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. RFC 2136: Dynamic Updates in the Domain Name System, 1997.

[37] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on CDN robustness. In *OSDI*, Dec 2002.

[38] 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*, 1999.

[39] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving map-reduce performance in heterogeneous environments. In *OSDI*, 2008.