Democratizing Content Distribution

Michael Joseph Freedman

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

September 2007

Professor David Mazières

© Michael Joseph Freedman

All Rights Reserved, 2007

"Behold how good and how pleasant it is for

brethren to dwell together in unity."

— Psalms 133-1

To my future wife Jennifer

for her warmth and support

and

To my brother Daniel

for his courage of convictions

Acknowledgments

My advisor, David Mazières, always insisted that I pursue research that gives me joy. He certainly provides a model to emulate. David is tenacious with his questions, precise with his research, and unwavering in his support. I would not have had the same view of and success in research without his guidance over the years. I would also like to thank Robert Morris and Frans Kaashoek for helping to place my feet on the road to academia.

This thesis is based on research I have performed over the past five years with a number of great colleagues, including Siddhartha Annapureddy, Christina Aperjis, Eric Freudenthal, Ramesh Johari, Maxwell Krohn, Karthik Lakshminarayanan, David Mazières, Sean Rhea, and Ion Stoica. Graduate school was an exciting and stimulating experience due to their involvements. I would also like to thank the other two members of my dissertation committee, Dennis Shasha and Lakshminarayanan Subramanian, for their advice.

Over my time spent at MIT, NYU, and Stanford, I had the opportunity to collaborate with and learn from many excellent peers. The intellectual variety and milieu provided by each environment was invaluable to both the content and style of my research. Beyond those already mentioned above, I would also especially like to thank David Andersen, Martin Casado, Nick Feamster, Nick McKeown, Antonio Nicolosi, Benny Pinkas, and Scott Shenker for being research mentors, sounding boards, and (in one case) a partner in technology transfer to industry.

The research in this dissertation was funded through an NDSEG Graduate Research Fellowship and the IRIS project (`http://project-iris.net/`), supported by the National Science Foundation under Cooperative Agreement Number ANI-0225660.

Finally, I wish to give heartfelt thanks to my family, for their abiding support, dedication, and love: my grandparents Estelle, Gerald of blessed memory, Janice, and Saul, my parents Nancy and Louis, my brother Daniel, and my future wife Jennifer. May good countenance continue to shine upon them all.

Bibliographic Notes

This thesis is based on research I have performed with a number of excellent colleagues between 2002 and 2007. The consideration of non-transitive routing failures presented in §2.2.4 appears in a paper co-authored with Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica [60]. An initial design for the indexing and CDN system described in §2.3 and §3 appears in papers co-authored with David Mazières [58, 59] and Eric Freudenthal [59]. Material for the anycast system in §4 appears in [62], co-authored with Karthik Lakshminarayanan and David Mazières. The file-system design in §5 appears in a paper co-authored with Siddhartha Annapureddy and David Mazières [8], while the erasure-code authentication protocol in §6 is co-authored with Maxwell Krohn and David Mazières [104]. Finally, the ongoing work on market incentives discussed in §7 is with Christina Aperjis and Ramesh Johari [63].

More information about the systems we present in this thesis—including papers, source code, software plugins, and system usage and integration instructions—can be found at the website `http://www.coralcdn.org/`.

Abstract

In order to reach their large audiences, today's Internet publishers primarily use content distribution networks (CDNs) to deliver content. Yet the architectures of the prevalent commercial systems are tightly bound to centralized control, static deployments, and trusted infrastructure, inherently limiting their scope and scale to ensure cost recovery.

To move beyond such shortcomings, this thesis contributes a number of techniques that realize cooperative content distribution. By federating large numbers of unreliable or untrusted hosts, we can satisfy the demand for content by leveraging all available resources. We propose novel algorithms and architectures for three central mechanisms of CDNs: *content discovery* (where are nearby copies of the client's desired resource?), *server selection* (which node should a client use?), and *secure content transmission* (how should a client download content efficiently and securely from its multiple potential sources?).

These mechanisms have been implemented, deployed, and tested in production systems that have provided open content distribution services for more than three years. Every day, these systems answer tens of millions of client requests, serving terabytes of data to more than a million people.

This thesis presents five systems related to content distribution. First, Coral provides a distributed key-value index that enables content lookups to occur efficiently and returns references to nearby cached objects whenever possible, while still preventing any load imbalances from forming. Second, CoralCDN demonstrates how to construct a self-organizing CDN for web content out of unreliable nodes, providing robust behavior in the face of failures. Third, OASIS provides a general-purpose, flexible anycast infrastructure, with which clients can locate nearby or unloaded

instances of participating distributed systems. Fourth, as a more clean-slate design that can leverage untrusted participants, Shark offers a distributed file system that supports secure block-based file discovery and distribution. Finally, our authentication code protocol enables the integrity verification of large files on-the-fly when using erasure codes for efficient data dissemination.

Taken together, this thesis provides a novel set of tools for building highly-scalable, efficient, and secure content distribution systems. By enabling the automated replication of data based on its popularity, we can make desired content available and accessible to everybody. And in effect, *democratize content distribution*.

Contents

List of Figures

Chapter 1

Introduction

The rise of the Internet and the World Wide Web has led to a new potential of information access, both in terms of scope and immediacy. To realize this potential—and actually enable clients to download their desired content—publishers simply need to provide server and network resources sufficient to handle their clients' demands.

Yet, until now, assembling such resources has not been quite so simple—at least without significant monetary outlays. Publishers currently reach large audiences by leasing resources from commercial content distribution providers, which build centrally-managed, provisioned, and trusted networks for this task. Unfortunately, as providers' capital and operational costs need be passed back to their customers, publishers unable or unwilling to afford these costs will see their content remain unavailable due to a paucity of resources.

We face a situation, therefore, where the availability of content on the Internet is to a large degree a function of the cost shouldered by the publisher. Thus, while the Internet has always provided the *potential* for the free flow of information and data to all interested parties, this potential currently cannot be realized without significant expenditures. And yet this limitation is not fundamental: There exists a wealth of otherwise-untapped resources on the Internet that could help to distribute content, if only publishers had a method of leveraging them.

Our goal is to make desired content available to everybody, regardless of the publisher's own resources. By building mechanisms that can organize any available resources at hand—be they far-flung, unreliable, or untrusted—we can automatically replicate content in proportion to its popularity. Such systems would thus, in effect, *democratize content distribution*.

This thesis describes mechanisms and systems that help realize this goal. While we propose and evaluate novel designs towards this end, our approach is not merely academic: Several of the systems we describe have been publicly deployed and in widespread use for several years. Not only do they provide open functionality that other distributed services can leverage, but they are also directly used by end-users—to the tune of more than a million clients and tens of thousands of content publishers each day—and thus enable the dissemination of content that may be otherwise unavailable.

1.1 Wither the democratization of content?

The journalist Walter Winchell once said that "Today's gossip is tomorrow's headline." This is even more true since the Internet has helped break down communication barriers and enable the rapid dissemination of information. Messages propagate rapidly through e-mail and other messaging layers, websites boom and bust in popularity, videos and other multimedia content spread virally, and so forth. This ever-changing popularity of content leads to heterogeneous and dynamic traffic demands from users, sometimes spiking into a sudden and often unexpected burst of user requests, a so-called *flash crowd*.

At least on the World Wide Web, this dynamism is aided by the largely world-readable openness of content and the nature of hyperlinks. Acquaintances can quickly point others to content by relating a short Universal Resource Locator (URL). Websites, portals, and weblogs commonly include such URLs to external sites as well, causing their own users to access these sites through explicit or embedded links.

While this type of information dissemination enables some content to attain overnight celebrity, it poses a danger as well. Sites hosting such content risk sudden overload following publicity, as their resources are insufficient to satisfy the new demand from these flash crowds of users suddenly seeking to download their content. These jumps in traffic may be several orders of magnitude greater than a sites' normal levels of traffic, a development witnessed by websites following sporting events (such as the FIFO World Cup site [9]), natural disasters (such as the U.S. Geological Survey site following earthquakes [185]), and attacks (such as the CNN and MSNBC sites on September 11, 2001 [132]), to name only a few occurrences. In the technology world, this phenomenon is often nicknamed the "Slashdot" effect, after a popular website that periodically links to under-provisioned servers, driving unsustainable levels of traffic to them.

The content publisher's limiting factor to serve these client demands may be caused by any number of bottlenecks under his control: insufficient access-link network capacity to receive or send content, CPU cycles to process requests, or disk bandwidth to read or write data to storage. While various engineering techniques can increase the immediate capacity of any of these potential bottlenecks—*e.g.,* compressing data before transmitting it on the network, using a poll-based rather than an interrupt-driven network interface to prevent system livelock [124], and so forth—these incremental solutions are only stopgap measures. Fundamentally, the only real scalable and fault-tolerant solution is to replicate desired content across many servers and many independent networks.

Indeed, commercial *content distribution networks* (CDNs) have built a profitable business model around offering such a service, including service providers such as Akamai [2], Mirror Image [123], and LimeLight Networks [108], or technology providers such as Cisco [41]. CDNs deploy a network of servers around the Internet that collectively serve their customer's content: When a client makes a request for a particular piece of content, the client is directed to one of these CDN servers from which to download the content—usually by using a layer of naming indirection, such as the domain name service (DNS)—instead of accessing one of the content publisher's origin servers. Today's large Internet sites heavily rely on these CDNs to scale their capacity to their clients' demands.

Yet these commercial CDNs, in seeking to control who can use their services (*i.e.,* their paying customers) and to provide corresponding quality-of-service guarantees as specified in service-level agreements with these customers, have built systems characterized by static, centrally-provisioned, and centrally-managed deployments. These CDN companies own and operate their CDN servers, carefully deciding where to place clusters of such servers in the network. For example, the LimeLight network is currently comprised of fifteen data centers in the United States, Western Europe, Japan, and China, while Akamai has established many smaller clusters to leverage network peering arrangements in order to reduce their bandwidth costs. The advantage of this approach is that the network provides a trusted, known, and more predictable infrastructure with which to serve content. The downside is that CDN providers' costs often scale linearly with their capacity—*i.e.,* in terms of the cost of servers, bandwidth, and power—and these costs are thus passed back to customers in the form of mostly-linear pricing schedules.

So where does this leave a content publisher that experiences a sudden jump in traffic, yet does not have such a preexisting service contract with a commercial CDN, given that their normal traffic patterns are low? Or even when the demand for content is predictable or regular yet still high, and the content publisher cannot afford these CDN prices? They are left with their content, quite literally, "in the dark."

Fortunately, there is a way for popular data to reach many more people than publishers can afford to serve themselves: other parties can *cooperatively* mirror the data on their own servers and networks. There are many reasons why parties will choose to contribute resources for content distribution. Indeed, the Internet has a long history of altruistic volunteer 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, in order to improve the global good.

There are less altruistic reasons for participating in such a system as well. By collectively performing content distribution, content publishers can perform the equivalent of time-sharing on their network and server resources: they can provision their sites for steady-state traffic patterns, yet weather any traffic spikes by spreading load among other participating resources. Furthermore, organizations that mirror popular content reduce their own downstream bandwidth utilization and improve the latency for local users accessing the mirror. Finally, one can build in mechanisms that incentivize users to contribute upstream resources, by providing better quality-of-service to contributors when resources are otherwise scarce.

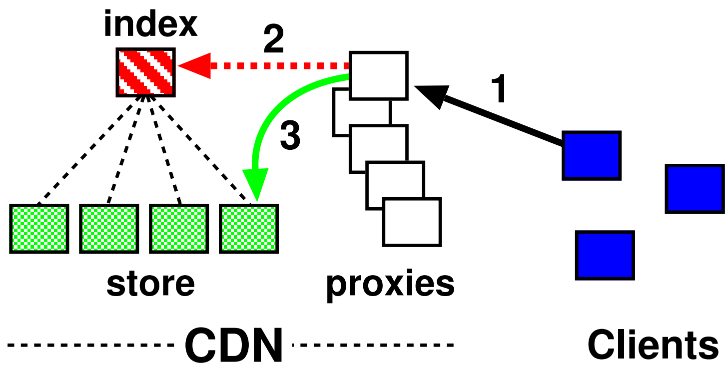


Figure 1.1: The main logical participants in a content distribution system.

1.2 Current CDN designs are insufficient

So why are the current architectures of CDNs insufficient to this task of cooperative content distribution? Let us first consider the necessary logical components and mechanisms of content distribution services.

At an abstract level, a content distribution system can be logically separated into several types of participants, illustrated in Figure 1.1: clients, proxies, content stores, and an index. Some system designs may physically combine some of these entities (*e.g.,* caching proxies that provide temporary content storage for other proxies), but we consider these logically distinct for now.

In order to serve a client with a particular piece of requested content, a CDN needs to address three mechanism design questions, as numbered in the figure. (1) *Server selection* answers with which proxy a client should directly interact. (2) *Content discovery* via some index finds where nearby instances of the client's desired resource are? And finally, (3) *secure content transmission* describes how this content should be fetched from the system and returned to the client. The accuracy of these answers are important, as the performance and cost of CDN systems depend highly on the nodes involved with handling a client's request. For example, file download times can vary greatly based on the locality and load of the chosen proxies and content stores, and a service provider's costs may depend on the load spikes that the server-selection mechanism produces.

As a simple example, consider how these three mechanisms are implemented within a typical

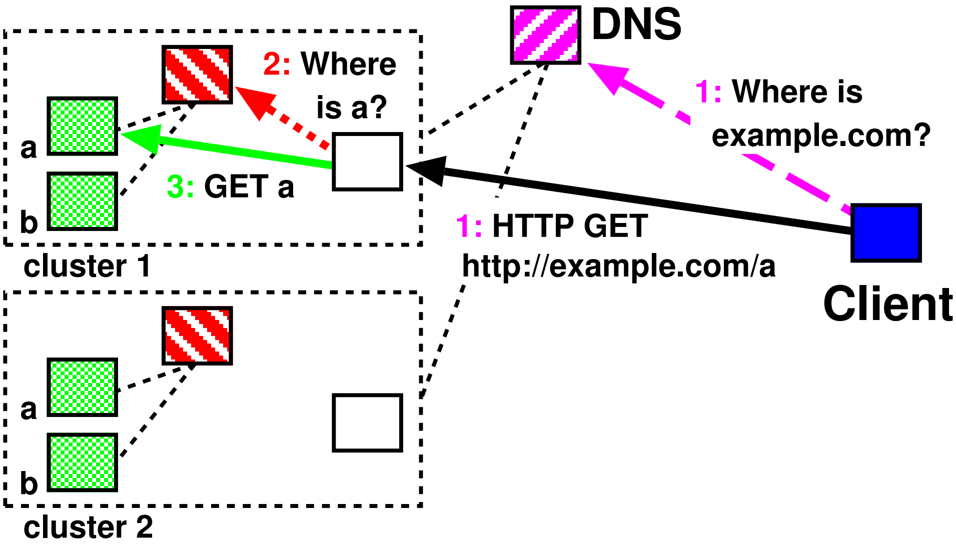


Figure 1.2: Getting content from a content distribution network comprised of two clusters via DNS redirection and HTTP requests.

commercial CDN architecture for web content, which is depicted in Figure 1.2 as comprised of two clusters deployed at different data centers, with each cluster containing the logical participants named in Figure 1.1. Let us assume that the publisher of a website (say, `example.com`) has delegated responsibility for serving `example.com` to the CDN. Consider the steps taken when a client fetches the content named by the URL `http://example.com/a`.

1. ***Server selection:*** The CDN first needs to determine from which data center the client should be served. This selection process, which we interchangeably refer to as *anycast*,[1] is commonly accomplished via DNS redirection: The DNS server(s) for the name `example.com`

[1]Some use the term anycast specifically to refer to IP anycast [137], which is characterized by late binding: Multiple hosts announce themselves as having the same anycast address through inter-domain routing protocols (BGP), and the IP routing layer causes a packet addressed to the anycast address to be sent to one of these hosts, based on the shortest-distance path in terms of network hops. (There are many arguments [10] why IP anycast never grew to widespread use, including scalability concerns, deployment challenges, and a lack of granularity and flexibility as compared to application-level mechanisms.) That said, we use a more encompassing definition of anycast: any mechanism used to select a host from some set of suitable destinations and communicate with it, either through early binding via naming indirection (*e.g.,* DNS) or through late binding via routing (*e.g.,* IP anycast).

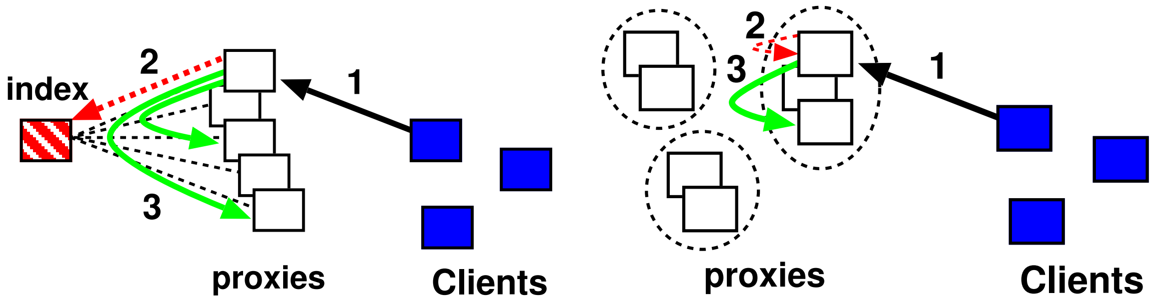
are aware of both data centers and, upon a name resolution query from the client's DNS resolver, will return an IP address for one of them, based on the client's location and the data centers' load. The client subsequently sends a HTTP GET request to a proxy located at the specified data center (either because the proxy is publicly addressable via IP or, more likely, because a pool of physical proxies are behind a hardware load balancer that demultiplexes client requests to the same virtual IP address).

2. ***Content discovery:*** Upon receiving a client request for content, the proxy queries a central "master index" within each cluster that keeps a mapping from all data items to the storage servers on which they reside, updated whenever stored content changes. The figure shows that files `a` and `b` reside on different stores, with their locations known by the index. Depending upon what type of functionality it provides—hosting versus caching content—if the stored content is not resident within the CDN, the CDN might either return an error message (in a hosted solution), or it might retrieve the new content from the publisher's origin servers and subsequently cache the content among its storage servers.

3. ***Content transmission:*** Finally, the proxy retrieves the data from the appropriate storage server and returns it to the client. If so-called dynamic content is involved, multiple pieces of content may be reassembled into a complete web object, either directly at the webserver (*e.g.,* via php, asp, EJB, or other programming languages) or within off-cluster proxies (*e.g.,* through Edge Side Includes [51]).

As alluded to already, virtually all commercial services use a centrally-managed and statically-provisioned architecture to provide these mechanisms. The use of a master index and secondary storage servers in both Figures 1.1 and 1.2, for example, illustrates the cluster-based storage and delivery approach taken by Google (via the Google File System [68]) and Yahoo! (via the Hadoop distributed file system [82]), in which a central "master chunk server" maps all data items to their current storage server locations.

Such a centralized approach is often taken to simplify system design and architecture, making it easier to administer, audit, and (arguably) deploy new services. If we consider these three



(a) BitTorrent-style architecture (b) Akamai-style cluster architecture

Figure 1.3: Potential content distribution system architectures

mechanisms in such a context: locality-driven server selection only needs to choose between the small set of stable data centers; content discovery only relies on the index at any cluster's master server; and whole files are transmitted from trusted servers. Unfortunately, this design comes at a significant cost as well: such deployments are expensive to build and operate, often do not make an efficient use of network resources (with content traversing the core multiple times), and ultimately do not scale to the increasing size and quantity of content that services seek to distribute.

There are certainly some alternatives in the design space, as shown in Figure 1.3, which may merge the role of proxies and content stores. A BitTorrent-style architecture [17], for example, has participating nodes all communicate their cached content to a central indexing server (called the tracker). Designed primarily for large files, chunks of content subsequently are downloaded in parallel from multiple peers. Unfortunately, this design also suffers from a central point-of-failure and limited scalability. (We should note that the BitTorrent protocol has a tracker manage information only about a single file, requiring some out-of-band, unspecified communication mechanism to find this tracker. Directly applying this approach to a cooperative CDN would require that the central tracker maintain information about *every* piece of content in the system, or it would require the introduction of a new content discovery layer for finding the tracker associated with a specific file.)

Akamai deploys known clusters of nodes, removing the need for a central meta-data index through its use of consistent hashing [94] (described further in §2.2.1) to assign responsibility for content storage among nodes *within* each cluster. These clusters are still statically-configured and mostly independent, provisioned with customer content through the aid of a central management system [168]. Our goal for a cooperative system can have no such provisioning, as the corpus that forms its content can span a huge domain, *e.g.,* the entire Web. Furthermore, this design controls data placement—the consistent-hashing algorithm decides where to place content within each cluster—as opposed to having proxies cache and share content in which they themselves (or their local users) are interested. Such a design is less amenable to cooperative environments. Finally, such an architecture assumes reliable and adequately-provisioned back-end storage for content, either centrally hosted by the commercial CDN or at the origin site, from which content is pushed out or pulled in by clusters. An underprovisioned origin site hit by an unexpected flash crowd, on the other hand, might quickly become unavailable.

A related design (not illustrated) is that taken by hierarchically-organized proxies [33], in which independent proxies can directly query their statically-configured neighbors (*e.g.,* via the Internet Cache Protocol [187]). This type of cooperation, while avoiding centralization, can still see many queries to an origin site, as each local group of configured proxies needs to independently fetch the content.

1.3 Design principles

This discussion of alternative designs leads to several design decisions that are important for a cooperative CDN:

Do not assume participants will delegate control over the contents of their cache. While users have clearly demonstrated their willingness to burn bandwidth by sharing files in which they themselves are interested, there is less incentive to dedicate bandwidth to sharing unknown data and, indeed, significant concerns in doing so. Such an approach confronts users with the questionable requirement that they must "donate" storage and network resources to the system for content *they have no control in choosing*, even though serving certain types of data can raise serious legal and moral concerns.

We posit that a cooperative CDN should not attempt to control the *placement* of data, but rather provide a content discovery layer that can locate content already cached by participants as a side-effect of their local access patterns: *i.e.,* index and manage location meta-data, not data itself. This approach allows different nodes to implement various caching and security policies. Some might provide unrestricted proxy service for the wider Internet population. Others, however, may be configured only to cache content requested by their local clients, but also provide upstream bandwidth to cooperatively serve other proxies (*i.e.,* Step 3 in Figures 1.1 and 1.3).

While content may have an authoritative source, do not assume that it is reliable or sufficiently provisioned. As our goal is to ensure that popular content remains available, content publishers with insufficient capacity at their disposal must be spared from a large demand for their resources. A cooperative CDN designed for this purpose should minimize load at the publisher's origin server whenever possible. To do such, the system's content discover index should enable cooperation between *all* participants, such that if a piece of content if available *anywhere* in the network, it can be efficiently discovered and transmitted.

The importance of global discovery also arises in cases when sudden load to a publisher's origin site would otherwise make it unavailable: Once any content—even if seemingly-unpopular, perhaps, because a flash crowd to the CDN has only begun—is cached within the CDN, it remains available to subsequent client requests.

Scale to Internet-size client populations and be robust to failures. No single or few nodes are sufficiently reliable or scalable to maintain the system's entire location meta-data index. Thus, both the content discovery index and the cooperative data cache should be distributed over large numbers of participants, in order to avoid any central point-of-failure or system bottlenecks.

Leverage participants wherever they may be found. While centrally-planned and -provisioned systems may be built around well-formed and well-known clusters—and grow by deploying additional clusters or adding new nodes to existing clusters—a cooperative system has only the peers that choose to participate. As such, the location of these participants may be unknown, they may differ greatly by network capacity, they may be quite remote from another other participants, they are not accompanied by any special hardware load-balancers or related infrastructure, and, of course, they may join or leave the system quite unexpectedly. Yet, in order to scale to true Internet-size populations, a cooperative CDN needs to encompass as many resources as it can find. Thus, the system should be designed to be self-organizing and self-managing to properly leverage these disparate resources effectively.

The popularity of different content can vary by orders of magnitude. Not only does the popularity of content normally differ greatly in content distribution systems—it has been demonstrated that web objects, file-sharing content, video-on-demand, DVD or music sales, etc. all follow a heavy-tailed distribution (*e.g.,* per [5, 19, 77, 194], among others)—but our cooperative CDN is designed specifically with flash crowds in mind. Thus, a naive approach to partitioning the index over multiple hosts—by statically assigning responsibility of different items to different hosts—could still have scalability limitations due to load imbalances. Thus, the system should be designed to avoid system bottlenecks caused by the heterogeneous popularity of content.

The system should provide low end-to-end latency. Finally, while the ability to minimize origin server load takes precedence over performance—hence the design choice of global content discovery, as opposed to purely local search—the CDN service should also strive towards low end-to-end latency to provide an acceptable client experience. To achieve such, all three CDN mechanisms—discovery, transmission, and server selection—should be designed for low latency and fast failover, with both asymptotic performance and practical engineering techniques in mind.

1.4 Contributions and organization

It is our thesis that one can democratize content distribution by enabling highly scalable and efficient resource cooperation through decentralized architectures. Our contributions are two-fold. First, we propose new algorithms and architectures for all three important components of content distribution systems. Second, we realize the goal of cooperative content distribution through the design, implementation, and deployment of several systems that, by leveraging these techniques, currently serve several terabytes of data to more than a million users every day. The view of the Internet gained from deploying these systems provided important insights that helped evolve our systems' designs.

The first half of this thesis describes content distribution systems that are backwards-compatible with today's servers and clients. As such, they can immediately benefit underprovisioned content publishers seeking to reach large audiences. On the other hand, these techniques are limited in their ability to leverage *untrusted* participants, given a lack of end-to-end security mechanisms for unmodified clients. Thus, in the second half, we consider a more *clean-slate* approach to building CDN services, tackling the problem of securing content transmission between distrustful peers.

More specifically, this thesis is organized as follows.

Chapter 2 – Content discovery and Coral. We start with the problem of content discovery: Given a request for a particular data object, how can one determine whether and where that object resides within the system? This chapter introduces a distributed key-value index well-suited for large, cooperative CDNs. This distributed index—which maps objects to the nodes caching them—is designed to replace the centralized functionality described in §1.1.

We start by reviewing the basic indexing approach provided by consistent hashing, and then discuss the more scalable algorithms of distributed hash tables (DHTs). After describing how these proposals break under realistic routing conditions and how to fix them [60], we argue why the traditional key-value *stores* as proposed in the literature are ill-suited for building cooperative CDNs and the like. In response, wee propose a new abstraction, called a *distributed sloppy hash table* (DSHT), that weakens their consistency model to yield a better-suited distributed index [58].

This chapter describes and experimentally validates the design of Coral, a distributed indexing layer that couples these weaker DSHT semantics with a hierarchical, locality-optimized structure. Coral enables lookups to occur efficiently and returns references to *nearby* cached objects whenever possible, while still preventing hot-spots in the indexing layer, even under degenerate loads. As such, Coral implements many of the design points we argued for in §1.3: it scales to Internet-size populations while not controlling data placement, it is robust to failure and self-organizing, it can handle great load imbalances to particular data items (named by keys in the index), and it yields low discovery and transmission latency.

Chapter 3 ***– CoralCDN.*** Leveraging the distribution indexing functionality of Coral, we next describe the design and implementation of CoralCDN [59], a decentralized, self-organizing web content distribution network. To use CoralCDN, some party simply appends "`.nyud.net`" to the hostname in *any* URL. Through DNS redirection, oblivious clients accessing this so-called "Coralized" URL are transparently redirected to nearby CoralCDN web caches. As such, CoralCDN is trivial to integrate, and thus can—and does—provide immediate relief for otherwise-overloaded sites.

Upon receiving client requests, CoralCDN caching proxies cooperate to transfer data from nearby peers whenever possible, dissipating most of the traffic to a URL's origin server and enabling its content to reach much larger audiences. CoralCDN's use of the Coral indexing layer both enables nodes to efficiently find content without querying distant nodes and avoids any load imbalances in its overlay and proxy infrastructures.

CoralCDN has been deployed on 300-400 servers for over three years. Each day, it serves several terabytes of web content to over one million unique client IPs, spread across an average of 20 to 25 million requests. This chapter concludes by describing robustness, fairness, and security mechanisms—learned through CoralCDN's deployment—introduced to improve its effectiveness in real operating environments. We also discuss the failures of CoralCDN's original DNS redirection service, which motivated the design of our next system.

Chapter 4 ***– Server selection and OASIS.*** We next turn to the problem of server selection: Given a set of CoralCDN proxies, which one should a browser access to maximize performance? But rather than construct another CoralCDN-specific service, we seek to build a general service that can support *many* distributed systems simultaneously, with marginal additional cost per system.

This chapter presents OASIS [62], a shared locality- and load-aware server-selection infrastructure. Since OASIS is shared across many application services, it amortizes deployment and network measurement costs to services that adopt it. Yet to facilitate such sharing, OASIS has to maintain network locality information in an application-independent way. OASIS achieves these goals by mapping different portions of the Internet in advance (based on IP prefixes) to the geographic coordinates of the nearest known landmark, using simple constraint satisfaction tests to ensure that measurement results are not (accidentally or maliciously) erroneous. OASIS is designed explicitly to incentivize cooperation: the more distributed services that adopt OASIS, the more vantage points OASIS has with which to perform network measurement, which in turn leads to improved anycast accuracy *and* decreased cost per participating service.

OASIS again demonstrates the facility of distributed key-value indexing techniques introduced in §2, which it uses to store information and assign tasks—which network prefixes to probe, which servers belong to which distributed systems, where prefixes are located—in a fully-decentralized manner. OASIS has been publicly deployed for two years, currently providing anycast service for about a dozen distributed systems.

CoralCDN and OASIS focus on building cooperative services for unmodified clients, with the goal of immediate deployability and widespread benefit.

What are the implications of this backwards compatibility? First, the systems are limited in their ability to explore more advanced methods of content transmission directly to end clients: from chunk-based (or swarming) file distribution (Chapter 5) to erasure-encoded data dissemination (Chapter 6). Second, and even more important, these unmodified clients cannot readily deal with malicious proxies. Web security *a la* SSL, for example, is designed to *authenticate* servers, not ensure the *integrity* of the data they return. A CDN proxy server could thus maliciously modify content en-route to the client without its detection. While this latter problem is easy to solve, at least technically provided that the fraction of malicious proxies is small—namely, origin servers can cryptographically sign content, and a client can failover to alternative proxies or back to the origin server in its content downloaded from a proxy does not contain a valid signature—no such functionality universally exists in today's web browsers.

Therefore, much like Akamai and Limelight have deployed their centrally-managed CDNs on trusted infrastructure, we have deployed the decentralized CoralCDN on trusted hosts (namely, the shared PlanetLab network [145], spread over about 300 university sites).[2] Thus, even through their architectural designs are decentralized and highly scalable, the systems' current *deployments* are not "peer-to-peer" in the sense that the resource beneficiaries are not necessarily the resource contributors, given our clients' limitations.

In the second half of this thesis, we consider some *clean-slate* content distribution systems for different settings, *i.e.,* read-write data and very large files. By designing new functionality for end-hosts, we should how even mutually-distrustful peers can use each others' resources to improve system performance and scalability.

Chapter 5 – Shark. Turning from immutable content to read-write data, the next system we present, Shark [8], considers the problem building a scalable yet secure distributed file system. A network file system can help support widely-distributed applications. Rather than manually replicate program files, users can place a distributed application and its entire run-time environment in an exported file system, and then simply execute the program directly from the file system on all nodes: The file system handles any necessary content distribution and synchronization transparently. Unfortunately, current network file systems like NFS have clients fetch data from a central file server, inherently limiting their scalability.

Shark's design, on the other hand, only requires the central server to initially authorize a client to read data and provide meta-data. Actual data transfers occur directly between caching peers, minimizing load on the central server, and security protocols ensure the integrity of downloaded chunks and reader's authorization to do so. In fact, given the way that Shark splits files into smaller chunks and names these chunks, Shark provides *cross-file-system sharing*, such that peers can cooperate to share identical content chunks, even

[2] OASIS has a mixed deployment, using the resources of third-party services benefiting from its functionality for network measurement, but using PlanetLab for information storage and front-end interfaces such as DNS and HTTP redirection.

when they belongs to different files in different file systems. By using a Coral index to communicate, Shark clients can locate nearby chunks of files across a global-scale system.

***Chapter 6* – Securing erasure-coded transmission.** As a final variant of content distribution, we turn to the problem of disseminating very large files on the Internet. A particularly attractive approach for this context, taken by multiple research systems, uses rateless erasure codes to avoid chunk negotiation among peers. That is, instead of the need to locate and download *specific* data chunks, a peer need only fetch some threshold number of randomly-chosen coded chunks in order to recover a file. Unfortunately, such codes cannot use the traditional method for verifying the integrity of data-blocks on-the-fly: As the domain of potential code blocks is exponentially large, using cryptographic signatures with hash trees (Merkle trees [120]) would require an exponentially-large hash tree. This limitation seemingly makes such schemes ill-suited for untrusted environments.

In response, we present a scheme based on homomorphic hashing that allows a downloader to perform on-the-fly verification of erasure-encoded blocks [104]. We prove the protocol's security (by tight reduction to standard discrete-log cryptographic assumptions), and our evaluation shows that, by incorporating optimizations such as probabilistic batch verification, it can be surprisingly efficient. This protocol, therefore, enables the application of coding techniques for efficient content dissemination to more open (and hence untrusted) settings.

***Chapter 7* – Summary and incentiving participation.** Finally, this chapter summarizes our contributions and discusses some ongoing work in incentivizing participation towards a more peer-to-peer design. Not only does our preliminary market-based approach encourage users to contribute upstream resources, but it also naturally leads to peers efficiently choosing which files are "most useful" to transmit and from which peers to download content in order to minimize network congestion.

Taken together, this thesis provides a novel suite of tools related to content distribution, ranging from content discovery, to server selection, and to secure content transmission. While we describe the design, implementation, and deployment of several concrete content distribution systems built using these mechanisms, we envision that they may be applicable and useful to a wider range of distributed services.

Chapter 2

A Distributed Indexing Architecture for Content Discovery

This chapter focuses on the problem of content discovery: Upon receiving a request that names a particular object, a CDN node must determine whether and where that object resides within its system. Only then may it fetch the desired content and serve it in response to the request.

At an abstract level, this content discovery step may be implemented via a key-value index supports two basic operations:

- $put(key, value)$: Inserts a mapping from the key to some arbitrary value.
- $get(key)$: Retrieves some or all of the values stored under a key.

In the context of content discovery, this index serves to map data objects—named by some type of key, such as a file handle in network file system, a URL in a web CDN, or a data checksum—to the node(s) currently storing them. If the index fails to return valid results for resident data, either clients will fail to retrieve desired data, or the data's origin servers may become overloaded following an excessive number of requests caused by client failover. Furthermore, if the indexing system either is slow to return results, does not properly spread requests between nodes, or insufficiently incorporates data locality, the client can experience poor end-to-end performance.

With these criteria in mind, this chapter describes the design and implementation of efficient, scalable, and fault-tolerant key-value indexes that are well-suited for content distribution systems.

2.1 Motivation

In most cluster-based architectures—as illustrated earlier in Figure 1.1—this key-value index is managed by a single server, providing strong semantic guarantees. Recent examples of such architectures include cluster-block servers (*e.g.,* GFS [68] and Hadoop [82]), network-attached storage device architectures [69], and many commercial cluster management solutions (*e.g.,* IBM Tivoli Storage Manager [86]).

Our goal, on the other hand, is a key-value index without any central point-of-failure or scalability bottleneck. While protocols such as primary-copy replication [109, 131] and consensus [106] have existed for some time as a means to remove a central point-of-failure in order to improve availability, they suffer from worse scalability bottlenecks due to the overhead of *data* replication.

This chapter describes several variants of distributed key-value indexes, all designed to operate at scale and across the wide-area Internet. Instead of one server storing the entire key-value map, this map is partitioned over a set of nodes, in which each node stores the key-value entries that belong to its partition. Such a distributed map requires that all nodes agree on the partitioning, *i.e.,* (1) each node knows the partition for which it is responsible, and (2) each node can determine and contact the node responsible for any key-value entry. The systems we present in Chapters 3, 4, and 5—for web content distribution, an anycast infrastructure, and a distributed file system, respectively—will all make use of various flavors of these distributed key-value indexes.

In Section §2.2, we review the basic design of a key-value index based on consistent hashing [94]. Such an index requires that each node maintain a global network view, *i.e.,* knows all other nodes in the network. Since Karger *et al.*'s seminal work [94], a number of papers (*e.g.,* [116, 150, 162, 177, 195]) have proposed more scalable algorithms, named *distributed hash tables* (DHTs), which we also review. DHTs extend this distributed partitioning to settings in which each node need only know some subset of other nodes in the system. Because each node only knows a smaller subset of peers, these algorithms also include some routing mechanism by which known intermediate nodes are used to contact the ultimate node responsible for any key.

In Section §2.3, we argue that the DHT-based storage abstraction that has largely captured the interest of the academic community—proposed as an building block for higher-level applications—is actually ill-suited for distributed services such as content distribution: They provide the wrong abstraction, have poor locality, and control data placement in the network (and thus ignore users' own preferences and policies).

In response, we present the design of a new key-value *indexing* layer, called *Coral*. Section §2.3.1 describes how Coral weakens the semantics of DHT's traditional get and put interfaces, in order to provide an indexing abstraction better-suited for the type of distributed applications we seek to build. Section §2.3.2 describes how we extend this basic design to a hierarchical structure, in order to provide better routing and data locality.

Coral makes several contributions. Its new abstraction with weaker semantics is potentially of use to any application that needs to locate nearby instances of resources on the network: Indeed, we later build a CDN for web content (Chapter §3) and a distributed file system (Chapter §5) on top of Coral. Coral introduces an epidemic clustering algorithm that exploits distributed network measurements to construct its locality-optimized hierarchical structure in a fully self-organizing manner. Finally, Coral is the first peer-to-peer key-value index that can scale to many stores of the same key without hot-spot congestion, based on a combination of new rate-limiting and constrained-routing techniques.

2.2 A distributed key-value index

This section reviews previous work in consistent hashing (§2.2.1) and distributed hash tables (§2.2.3) for building key-value indexes. While these mechanisms have been proposed by other researchers in the literature, our contribution lies in extending these approaches to be more scalable and fault-tolerant. Namely, §2.2.2 describes the addition of a gossip-based group membership protocol in order to minimize the amount of liveness probing necessary to maintain a global mem-

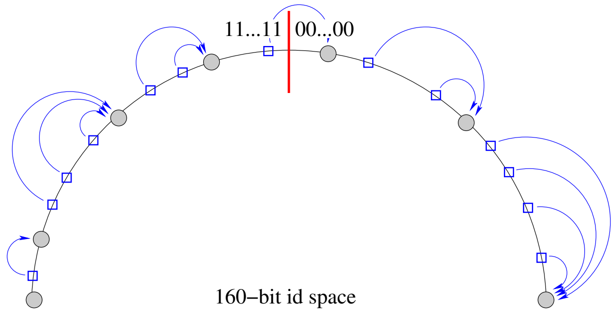


Figure 2.1: Keys (blue squares) are assigned to their successor nodes (circles) on a consistent-hashing ring. This assignment is illustrated by arrows in the figure.

bership view, while §2.2.4 describes an incorrect assumption of previous designs—specifically, a fully-connected communication graph between all peers—that leads to faulty behavior under routing failures, as well as our resulting system modifications to better handle real network conditions.

2.2.1 Consistent hashing with global membership views

Consistent hashing [94] partitions the identifier space of a key-value map over participating nodes as follows. Each participating node is assigned one or more random identifiers from the space of integers modulo 2^K, where identifiers (IDs) are K-bit numbers. Each key, also drawn from the same domain of identifiers, is mapped to the node whose node-ID most immediately follows it in this ID-space, as shown in Figure 2.1. We thus say that the system implements the *successor* relation, and so long as each node in the network knows its predecessor in the ID-space, any node can compute which keys are mapped onto it. For concreteness, we use the set of integers $[0, 2^{160}]$ as the ID-space in our descriptions, which is well-suited for generating pseudorandomly-distributed keys by taking the SHA-1 hash [55] of some input (*e.g.,* a node's IP address).[1]

The main benefit offered by consistent hashing over static partitioning—*e.g.,* using some hash function modulo N where N is the number of nodes—is that consistent hashing minimizes the disruption caused by nodes joining or leaving the system. If κ is the number of keys in the system, membership changes only disrupt $O(\kappa/N)$ keys with high probability when using consistent hash-

[1]One could certainly use 256-bit identifiers and SHA-256 as an alternative hash function instead, if the continued security of SHA-1's collision resistance is a concern.

ing: A new node takes responsibility of some subset of its successor's keys, while a node leaving the system passes responsibility for its keys onto its successor; no other system reconfiguration of keys is necessary. On the other hand, membership changes when using modulo-based hashing disrupt $O(\kappa)$ keys, leading to significant overhead.

If every node knows about every other node in the system, each node can easily determine to what node every key is mapped. Thus, such a structure can support put and get primitives by first determining the node responsible for the specified key, then directly contacting that node either to store the inserted value or to fetch the stored value(s).

Of course, if reliable storage in this key-value map is desired, the system must replicate key-value pairs across multiple nodes. Replication ensures that when nodes leave the system, the key-value pair can still be found at some live node. On the other hand, when a new node joins the system, it should get a copy of all new key-value pairs for which it will be subsequently responsible (due to a re-partitioning of the identifier space). The literature includes proposals for both reactive replication [39, 45] (*i.e.,* upon the detection of node failures) and proactive replication [149] (based on expected system behavior).

Such a global membership view can either be managed by a central membership server, or each node can occasionally ping every other node to check its liveness and avoid the need for centralization. In this decentralized model, a node can join the system if it knows at least one other existing member, *e.g.,* by fetching that node's membership list and subsequently contacting all newly-discovered nodes. Of course, this simple membership approach does not ensure that each node's view of the network is consistent. It does, however, provide the property of *eventual consistency*, given the use of an unreliable failure detector [32]. Strongly-consistent membership views, on the other hand, would require heavier-weight protocols based on group consensus (*e.g.,* Paxos [106]).

Application designers using such a key-value index should determine whether they require such strong consistency from the mapping of keys to nodes, or whether they can operate with only eventual consistency. The applications we later describe are able to function with the latter, either by only assuming best-effort reliability by the indexing layer (§3, §5) or by replicating keys at multiple hosts (§4), such that temporarily inconsistencies do not effect the correctness of the system. Thus, they can avoid the communication and complexity overhead of implementing consensus protocols.

2.2.2 Scaling liveness detection for global membership views

We now describe an alternative liveness detection mechanism that greatly reduces the amount of network probing in the system. Specifically, to avoid the $O(N^2)$ probing used by an all-pairs approach, participating nodes detect and share failure information cooperatively.

In this model, every node maintains a weakly-consistent view of all other nodes in the network, where each node is identified by its IP address and a globally-unique node identifier as before, as well as a new *incarnation number*. Each time period—*e.g.,* 3 seconds—every node picks a random neighbor to probe and, if the neighbor fails its liveness check, uses epidemic gossiping to quickly propagate its suspicion of failure.

Two techniques suggested by SWIM [48] reduce false failure announcements. First, if the initiating node's direct probe of its neighbor fails to elicit a response, it next chooses several intermediates to again probe this target. Only if these indirect probes fail as well does the initiator announce its suspicious of failure. Such indirect probing alleviates the problem caused by partial network, as opposed to end-host, failures. (We discuss the implications of such non-transitive failures in §2.2.4.) Second, incarnation numbers help disambiguate failure messages: *alive* messages for incarnation i override anything for $j < i$; *suspect* for i overrides anything for $j \leq i$. If a node learns that it is suspected of failure, it increments its incarnation number and gossips its new number as alive (no other party should increment a node's incarnation number). A node will only conclude that another node with incarnation i is dead if it has not received a corresponding alive message for $j > i$ after some time—3 minutes in our implementation—although the node may treat suspicious nodes differently (*e.g.,* use more aggressive request timeouts when communicating). This approach provides live nodes with sufficient time to respond to and correct false suspicions of failure, while still bounding the time that a node failure escapes notice.

To gossip these failure messages, each node maintains a buffer of announcements to be sent (or piggybacked on other system messages being disseminated throughout the system) to *randomly-*

chosen nodes. This buffer includes messages both originated by the local node or received from neighbors for retransmission. Every time a node transmits a message to a random neighbor, it increments a transmission count. When the message has been gossiped $O(\log N)$ times for N-node networks, the message is removed from this buffer. Such an epidemic algorithm propagates a message to all nodes in logarithmic time with high probability [48].[2]

While the all-pairs approach is limited solely by network size, scalability in this gossip-based approach is limited by a function of the product of failure rate *and* network size; *i.e.,* it can support small networks with high failure rates, or it can scale to larger networks of stable nodes. Thus, it may be better suited for decentralized services deployed on managed infrastructure nodes—as we use for the anycast system of Chapter 4—than for peer-to-peer services on end-hosts.

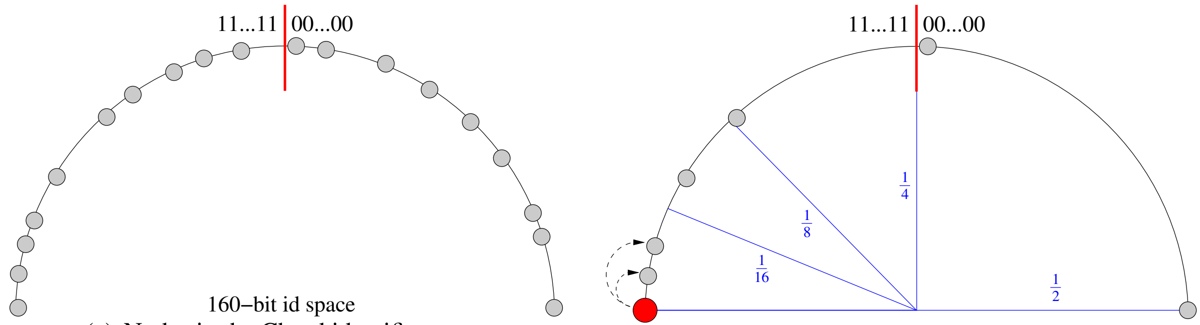
2.2.3 Partial membership views with DHTs

This previous approach of ID-space partitioning can be scaled to much larger networks if every node need only know about a subset of the network's nodes, as opposed to a global view. Enter the routing layer proposed for distributed hash tables (DHTs), which we review here. While there are some algorithm differences between the various proposals [116, 150, 162, 177], they all share the same basic approach.

As in consistent hashing, a DHT assigns every key in the ID-space to a node based on some distance metric. In consistent hashing and Chord [177], this distance metric is simply the successor relationship across integers modulo 2^K for K-bit keys. Kademlia [116] defines the distance between two values in the ID-space to be their bitwise exclusive or (XOR), interpreted as an unsigned integer; thus, IDs with longer matching prefixes (of most significant bits) are numerically *closer*. The node closest to a key can be called either the *root* or *successor* of the key, terms we use interchangeable.

The main primitive that DHTs contribute is $lookup$, in which a node can efficiently discover a key's root, even through each node only has a partial view of the entire network. The lookup

[2]While structured gossiping based on consistent hashing could reduce the bandwidth overhead needed to disseminate a message [31], we use a randomized epidemic scheme for simplicity.



(a) Nodes in the Chord identifier space

(b) One node's view of Chord's ID-space

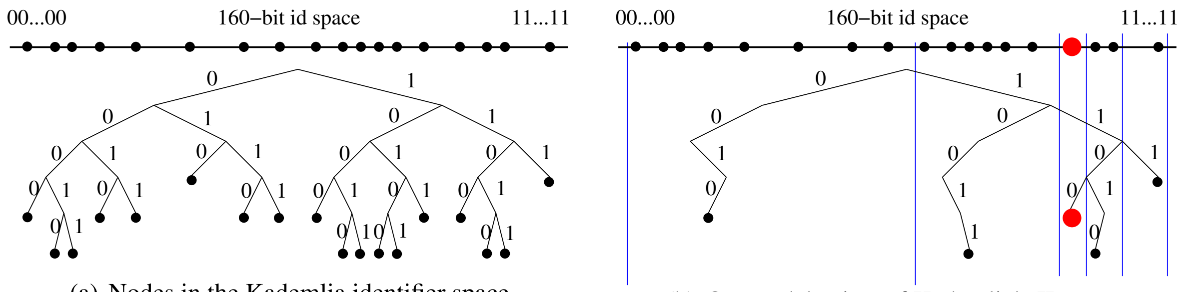
Figure 2.2: *Chord DHT.* The left-hand graph visualizes all system nodes in half a Chord ring based on their node identifiers. The right-hand graph displays the routing tables of one node (shown larger in red). Dotted lines are draw to the red node's successor list; all other visualized nodes appear in the node's finger tables. These fingers constitute those nodes that are successors to points that are powers-to-two distant (separated by thin blue lines) from the red node on the ring.

protocol discovers the key's root by greedily traversing known neighbors in the DHT, progressing closer to the root of the key at each step.

Each node maintains a set of neighbors that it uses to route packets. Typically, such neighbors are divided into (a) *short links* chosen from the node's immediate neighborhood in the ID space to ensure the correctness of lookups, and (b) *long links* chosen to ensure that lookups are efficient. In Chord, the set of short links is called the node's *successor list*, and the long links are called *fingers*. While Kademlia uses a single routing table, one can still differentiate between its closest set of short links and farther sets of long links. Figures 2.2 and 2.3 visualize nodes in a routing structure (left), as well as the routing table as seen from one node (right), for both Chord and Kademlia, respectively.

If the size of a node's DHT routing table is logarithmic in the total number of nodes comprising the DHT, the total amount of liveness probing that each node need perform is $O(\log N)$. This compares very favorably to the $O(N)$ communication complexity needed to maintain the global membership views described earlier.

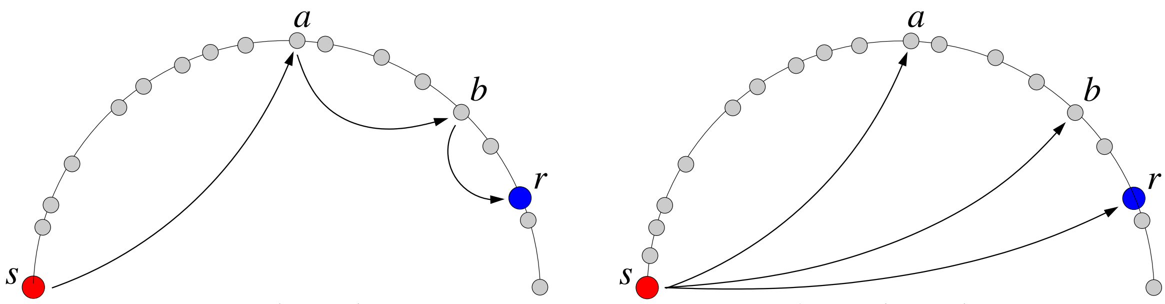
Additionally, if these logarithmic-sized routing tables are filled with other nodes whose IDs are chosen from a specific distribution, then a node can discover any key's root in a logarithmic number of routing hops with high probability. Specifically, if a node s is not the closest node to



(a) Nodes in the Kademlia identifier space

(b) One node's view of Kademlia's ID-space

Figure 2.3: *Kademlia DHT.* The left-hand graph visualizes all system nodes in a Kademlia tree based on their node identifiers. The right-hand graph displays the routing tables of one node (shown larger in red), who knows at least one node in each subtree (separated by thin blue lines) of increasing distance from it.



(a) Recursive routing

(b) Iterative routing

Figure 2.4: *Recursive* and *iterative* styles of Chord DHT routing

some key k, then s's routing table almost always contains either the closest node to k, or some node that is roughly one-half the distance to k. Repeated halvings of the distance leads to a logarithmic number of routing hops to find the key's root, node r.

Or, as described using the XOR metric, s discovers a node whose distance to k is at least one bit shorter. This permits s's *lookup* 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, again, the expected number of iterations for s to discover the closest node to k is logarithmic in the number of nodes.

These hops within the routing overlay can either be performed in an *iterative* or *recursive* fashion, as visualized on a Chord ring in Figure 2.4, where source node s initiates a *lookup* for some key k whose root is node r. In recursive routing, s contacts a, and a contacts b in turn, while in iterative routing, node s first contacts node a to learn about node b, and then s subsequently contacts b. Both routing techniques have different strengths. For example, recursive routing is faster than iterative routing using the same bandwidth budget [47, 156] and can use faster per-node timeouts [155]. On the other hand, iterative routing gives the initiating node more end-to-end control, which can be used, for instance, for better parallelization [60, 116, 156].

2.2.4 Non-transitive routing and failure detection

An implicit assumption in both consistent hashing (with global views) and DHT protocols (with partial views) is that all nodes are able to communicate with each other, yet we know this assumption is unfounded in practice. We say a set of three hosts, a, b, and c, exhibit *non-transitivity* if two pairs of these nodes can communicate with one another, but one pair (say, a and b) cannot. These transient periods of non-transitivity occur for many reasons, including link failures, BGP routing updates, and ISP peering disputes (*e.g.,* [128]).

2.2.4.1 The problem with non-transitive routing

Such non-transitivity in the underlying network is problematic for DHTs. Consider, for example, the segment of a Chord ring illustrated in Figure 2.5, where the dashed lines represent predecessor links. Identifiers increase from the left, so node b is the proper successor to key k. If nodes a and b are unable to communicate with each other, a will believe that c is its successor. Upon receiving an iterative lookup request for k, a will return c to the requester. If the requester then tries to insert a value associated with k at node c, node c would refuse, since according to its view it is not responsible for key k. The same problem exists with recursive routing, as a will simply try to do the insert at c itself.

Even if *individual* pairs of nodes exhibit non-transitivity with low probability, there still exists *some* pair of nodes in the network for which non-transitivity breaks routing with high probability. For example, hypothetically, if each pair of nodes with adjacent identifiers in a 300-node Chord network (independently) has a 0.1% chance of being unable to communicate, then we expect that

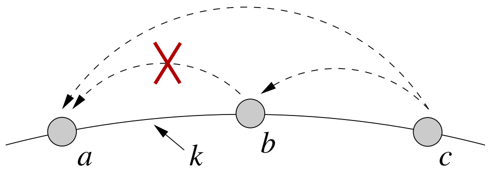


Figure 2.5: Non-transitivity in Chord routing.

there is a $(1-0.999^{300}) \approx 26\%$ chance that *some* pair will be unable to communicate at any time. However, both nodes in such a pair have a 0.999^2 chance of being able to communicate with the node that most immediately precedes them both.

Non-transitivity indeed exists with non-negligible probability. Gerding and Stribling [67] observed a significant degree of non-transitive network failures among hosts on PlanetLab [145], an internationally-deployed network of well-distributed servers (mostly at university sites) that we use to test and deploy the services described later (CoralCDN, OASIS, and Shark). They found that of all possible unordered three tuples of nodes (a, b, c), about 9% exhibited non-transitivity. They attributed this non-transitivity to the fact that PlanetLab consists of three classes of nodes: Internet1-only, Internet2-only, and multi-homed nodes. Although Internet1-only and Internet2-only nodes cannot directly communicate, multi-homed nodes can communicate with them both.

Extending the above study, we have found that *transient* routing problems are also a major source of non-transitivity in PlanetLab [60]. We found that, of all unordered pairs of nodes (a, b), about 5.2% of pairs cannot reach each other but another host c can reach both a and b. Of these pairs of nodes, about 56% of the pairs had persistent problems; these were probably because of the problem described above. However, the remaining 44% of the pairs exhibited problems intermittently; in fact, about 25% of the pairs could not communicate with each other only in one of the 15-minute snapshots we analyzed across a three-hour window [178]. This suggests that non-transitivity is *not* entirely an artifact of the PlanetLab testbed, but also caused by transient routing problems.

The very problem of non-transitivity has been leveraged by the RON [4] and SOSR [76] research projects to improve resilience by routing around network outages. Commercially, Akamai has also offered its SureRoute [3] overlay routing service to find faster or more reliable paths by routing through intermediate nodes in its deployed clusters.

2.2.4.2 Fixing non-transitive routing

The basic way to handle non-transitive connectivity is to simply route around the problem. To do so, we make two modifications to the above key-value indexes: (1) modifying routing tables to include indirect entries and (2) adding support for a basic $forward$ primitive.

First, we now represent entries in our routing tables as paths to the target destinations, where direct connectivity allows a path of length one, while indirect connectivity requires paths of length two or more. Using Figure 2.5 as an example, node a's routing table would include the following entries:

Destination	Routing entries
b	$\langle c, b \rangle$
c	$\langle c \rangle$

Second, we introduce a $forward$ primitive, which takes a destination and message, and wraps the two for overlay source routing. Given the above example with routing entry $b \rightarrow \langle c, b \rangle$, $forward(b, M)$ sends (b, M) to c, who forwards (a, M) to b in turn. If this message is part of a two-way communication such as RPC, b responds with (a, M') to c, who forwards (b, M') to a. (And more generally, b can choose to update its own routing table with the entry $a \rightarrow \langle c, a \rangle$.)

This basic approach can be used in overlays with both global and partial views. In the context of consistent hashing, every node keeps a routing entry about every other node in the above manner. In fact, our failure detection subsystem (§2.2.2) provides special support for precisely this functionality: Recall that before suspicion of a node's failure is raised after direct probing fails, an initiator will try to use several intermediates next (five in our implementation). Any such intermediate node that returns success at contacting the target can immediate be added to the initiator's routing table as an indirect route.

In the DHT context, at least when using recursive routing, we only need to be concerned with such non-transitivity in the context of short links (required for correctness), because the routing structure offers flexibility in the choice of long links (used for efficiency) [78]. To maintain these short links, nodes can reactively [29, 162] or periodically [155] exchange copies of their local routing tables; doing such, nodes discover indirection points with which to reach other local peers, even if no direct path exists. This process is akin to learning about link-state of all local peers. This solves such problems as *inconsistent roots* in DHTs [60], where nodes, when only using direct communication as a means for determining liveness, may have different views of who is the closest node to some key.

Iterative routing presents more problems when confronted with non-transitive routing, as nodes may not have direct connectivity to the next-hop peers returned by intermediates, even though those intermediates do—so called *invisible nodes* in [60]. In practice, this problem is mitigated in two ways. First, implementations often cache additional information beyond $O(\log N)$ routing state as a performance optimization (caching live nodes aids in accurate RTT information to set good timeouts; caching unreachable nodes prevents unnecessary delays while waiting for a request to time out). It is especially useful also to include nodes with indirect connectivity, accompanied by their routing entry, in this cache. (The state requirements for such are reasonable, given their relative scarcity with respect to a single node). Second, iterative routing implementations can keep a window of multiple outstanding routing requests, and thus avoid the performance hit of the single node's failure or delayed response. As a request approaches the key's root, indirect requests can be sent through the root's immediate neighbors, and thus avoid inconsistent roots.

2.3 Weakening the consistency model and adding locality with Coral

The academic community has proposed the use of distributed hash tables as an efficient, scalable, and robust storage layer for building higher-level peer-to-peer applications. We now ask whether DHTs are well-suited for some desired applications of the wider Internet population. For example, can DHTs be used to implement file-sharing, by far the most popular peer-to-peer application? Or could DHTs replace proprietary content distribution networks (CDNs), such as Akamai, with a more democratic client caching scheme that saves it from flash crowds at no cost to the server operator?

We suggest that the answer to these questions is largely no. DHTs fail to meet the needs of these real peer-to-peer applications for three main reasons.

DHTs provide the wrong abstraction. Suppose many thousands of nodes store a popular music file or cache a widely-accessed web page. How might a hash table help others find the data? Using the hash of a web object's URL as a key, one might use the DHT to store an *index*: a list of every node that has the object. That is, have puts append values (*e.g.,* IP addresses) under a given key, while gets should fetch a consistent list of all such servers caching the content.

DHTs typically replicate popular data for load balancing, but replication helps only with gets, not puts. Any node seeking a web page will likely also cache it: Any URL-to-node-list mapping would be updated almost as frequently as it is fetched. Thus, any single node charged with maintaining the URL-to-node mapping for a popular object can itself become a *hot-spot* in the indexing infrastructure.

DHTs have poor routing and data locality. Though some DHTs make an effort to route requests through nodes with low network latency [28, 47, 78], the last few hops in any lookup request are essentially random. Thus, a node might need to route a query half-way around the world to learn that its neighbor is caching a particular data object. This is of particular concern for any peer-to-peer CDN, as the average node may have considerably worse network connectivity than the web server itself.

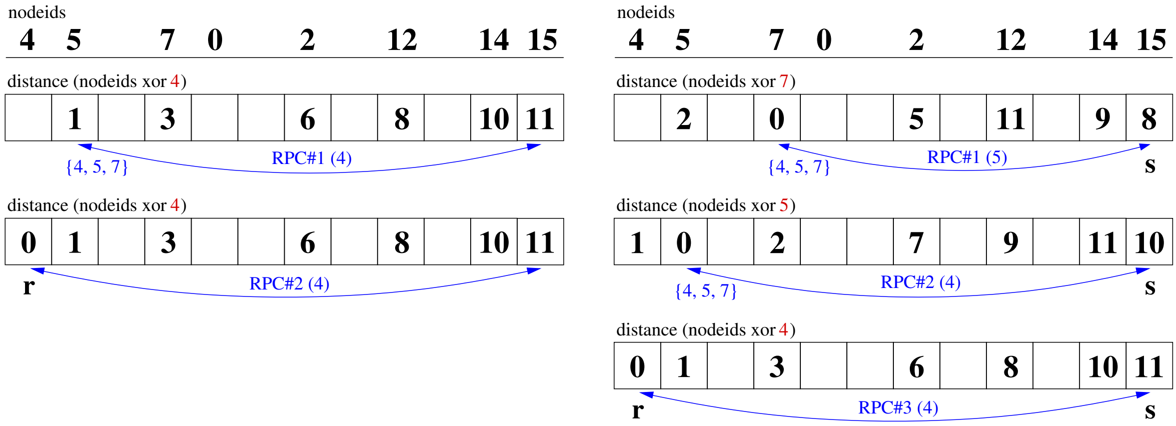
Even worse, if a get just returns a list of all nodes (or some randomly-chosen subset) caching the object, the initiating node does not have an easy way to determine which node is nearby and thus more efficient from which to download the content.

Store-based DHTs control data placement. An alternative approach to using a DHT as an indexing layer, with its "*write-many, read-many*" access patterns, is to store actual content in the hash table and thus support "*write-once, read-many*" access patterns. Unfortunately, this approach—taken by CFS [45], PAST [161], Squirrel [89], and CobWeb [172], among others—wastes both storage and bandwidth, as data must be stored at nodes where it is not needed and large amounts of data must be shifted around when nodes join and leave the system, a common occurrence in many systems [166]. The replication techniques of these systems must be careful: overly-aggressive replication wastes bandwidth and storage [39], while nodes become overloaded if data is insufficiently replicated. Moreover, as discussed in the Introduction, there are non-technical concerns with this approach, as it requires that users store and dedicate upstream bandwidth to content they have no interest in nor control in choosing.

This section presents a new type of key-value index, called a *distributed sloppy hash table* (DSHT), that overcomes these limitations of traditional DHTs. DSHTs are designed for applications storing soft-state key-value pairs, where multiple values may be stored under the same key. DSHTs weaken the semantics of get operations, such that they are designed to only return a *subset* of values inserted under a given key, as opposed to *all* values as in a hash table. Finally, we present *Coral*, an indexing layer that adds both good routing and data locality through a hierarchical structure of DSHTs. Coral is well-suited for content distribution and similar peer-to-peer applications: A node need not discover *all* peers caching a particular object, it only needs to find *several*, *nearby* ones.

2.3.1 Distributed sloppy hash tables

Given the weakened semantic requirements of its get operations, DSHTs only require that nodes satisfying a get request store a subset of the key's values, not necessarily all inserted values. Thus, DSHTs use what we term a "sloppy" storage technique that leverages a cross-layer interaction between the routing and storage layers: A DSHT caches key-value pairs at nodes whose identifiers are increasingly close to the key being referenced, as an inverse function of load. Values for unpopular keys get stored on the closest node—*i.e.,* the key's *root*—while values for popular keys get stored at nodes progressively further from the key's root. Yet precisely because such keys are already popular, other nodes, no matter where they are located in the network's identifier space,



(a) Opportunistic routing (b) Constrained routing

Figure 2.6: Comparing opportunistic and constrained routing

can retrieve some valid subset of values. Thus, at least for popular items, a key's closest node need not be contacted when fulfilling either a put or a get operation.

This design differs from the key-value layers of §2.2, which always attempt to insert a key-value pair at the key's root. Yet frequent references to the same key can generate high levels of traffic at nodes close to the key. This congestion, called *tree saturation*, was first identified in shared-memory interconnection networks [143]. To minimize tree saturation, DSHTs combine on-route caching (alluded to above) with constrained routing (*i.e.,* not deviating from the logarithmic-length path), which we describe next.

2.3.1.1 Constrained routing

Our basic routing algorithm is a constrained form of Kademlia [116], fixing $b = 1$ bit of the key at each iteration. For concreteness, our DSHT implementation uses 160-bit identifiers. Nodes are assigned IDs in the same 160-bit identifier space as keys; a node's ID is the SHA-1 hash of its IP address.

Figure 2.3.1 compares the opportunistic and constrained version of Kademlia routing. The illustration shows eight nodes with IDs $\{4, 5, 7, 0, 2, 12, 14, 15\}$, and node s with ID 15 is looking up the node closest to key $k = 4$. Nodes are listed in sorted order by their XOR distance to k.

In opportunistic routing, node $s = 15$ attempts to find the node closest to key $k = 4$ during each of its hop in the routing overlay. The top boxed row illustrates the XOR distances to k for the nodes $\{5, 7, 2, 12, 14, 15\}$ that are initially known by s, namely, distances $[1, 3, 6, 8, 10, 11]$, respectively. Data in RPC requests and responses are shown in parentheses and braces, respectively: First, node s asks the closest node to k that it knows about (*i.e.,* node with ID 5) for any closer nodes. Node s inserts these references $\{4, 5, 7\}$ it learns about into its routing table. In the next iteration (bottom boxed row), node s contacts the node with ID 4. As this is the closest node to k (with distance 0), s's lookup is finished.

In constrained routing, each iteration of a DSHT routing operation prefers to correct exactly b bits at a time. More specifically, let $\mathrm{splice}(k, s, i)$ designate the most significant $b+i$ bits of k followed by the least significant $160-b+i$ bits of s. If a node with ID s wishes to search for key k, the node first initializes a variable $t \leftarrow s$. At each iteration, it updates $t \leftarrow \mathrm{splice}(k, t, i)$, using the smallest value of $i \geq 0$ that yields a new value of t. The next hop in the route is the closest node to t that already exists in s's routing table.

With respect to Figure 2.3.1, we first calculate $7 \leftarrow \mathrm{splice}(4, 15, 0)$, then calculate the distance for each node known to s to ID 7, shown in the boxes of the top row: distances $[2, 0, 5, 11, 9, 8]$, respectively. Node s contacts the known node closest to key 7, which is the node with ID 7 in this example, and inserts the newly discovered nodes into its routing table. Node s then updates its target for the iteration of lookup: As $7 \leftarrow \mathrm{splice}(4, 7, i)$ for $i \in \{0, 1\}$ does not yield a new value for the target t, s calculates the next target as $5 \leftarrow \mathrm{splice}(4, 7, 2)$, and computes its known nodes distances to $t = 5$, shown in the boxes of the middle row: $[1, 0, 2, 7, 9, 11, 10]$. Node s subsequently contacts the node with ID 5. Finally (bottom row), s updates its target to $4 \leftarrow \mathrm{splice}(4, 5, 3)$, computes each node's distances to $t = 4$, and sends an RPC to the node with ID 4, completing the lookup.

In these examples, opportunistic routing was able to find the closest node in fewer hops than constrained routing, as it always greedily routes to the nodes closest to the key k. One might think that this problem is contrived: Given routing tables of size logarithmic in the network size, there should not be an asymptotic difference between the two approaches.

In practice, however, implementations *should* cache information about nodes beyond a strict $O(\log N)$ bound when possible, leading to the above differences. This caching is done for several reasons, especially when systems use iterative routing. (1) To achieve high performance, it is important to have accurate knowledge of a node's RTT to determine when to timeout a request and try an alternative node. (This was a primary motivation behind later research into virtual coordinate systems [46, 129].) (2) Due to non-transitive routing conditions (§2.2.4), it is important that nodes do either negative result caching or maintain a $forward$ entry to these locally-unreachable nodes; otherwise, a node may continue to add other nodes into its routing table based on information learned during a lookup, even though these new neighbors are not locally reachable. (3) Finally, most DHTs would *prefer* to perform one-hop routing if possible: The logarithmic overhead is usually viewed as a necessary evil due to scalability limitations, not as the *enabling* means by which to perform load balancing. Yet, for relatively-stable networks on the order of a few thousand nodes, a node may cache information about nearly all system participants. (Indeed, Accordion [107] was designed to dynamically trade-off routing table size for routing performance based on network conditions and a node's available bandwidth.)

Our constrained algorithm allows one to gain these benefits of additional node caching, while still preserving the traditional logarithmic-depth lookup path in the routing overlay. As described next, by limiting the use of potentially-closer known hops in this way, a DSHT can avoid overloading any node, even in the presence of heavily-accessed keys.

The potential downside of longer lookup paths, of course, is higher lookup latency in the presence of slow, stale, or non-reachable nodes. In order to mitigate these effects, our iterative implementation keeps a window of multiple outstanding RPCs during a lookup (currently three), thus avoiding blocking a lookup on a single node's failure or delayed response.

2.3.1.2 Inserting values

A DSHT performs a two-phase operation to insert a key-value pair. In the first, or "forward," phase, the system 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 already stores l values for k whose expiry times—each calculated as the sum of the value's insertion time and its specified time-to-live (TTL)—are all at least one-half that of the new value (as calculated from present).

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 $\beta = 12$, the latter 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 towards the root to ensure that nodes near to the key retain fresh values.

In the forward phase, the 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 inserting client 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 inserting client attempts to insert the value at the remote node referenced by the top stack element, *i.e.,* the contacted node that is closest to the key. If this operation does not succeed—perhaps due to others' concurrent 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 $(\beta \cdot (2^b - 1) \cdot \lceil \frac{\log n}{b} \rceil)$ 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 $(\beta \cdot (2^b - 1) \cdot \lceil \frac{\log n}{b} \rceil)$ 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 §2.3.3.2 provides supporting experimental evidence.

2.3.1.3 Retrieving values

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.,* an application proxy contacts multiple sources addressed by these values in parallel to download cached content.

Note that, as multiple stores of the same key will be spread over multiple nodes, the values (addresses) retrieved by the application are distributed among those stored. Thus, DSHTs provides load balancing both *within* the indexing layer (where the requests are handled) and *between* peers using the indexing layer (to where the values point).

2.3.1.4 Handle concurrency via combined put-and-get

If we use a DSHT for applications such as web content distribution—as we do in Chapter 3 with CoralCDN—we would want to enable participating nodes to offload all traffic from an origin webserver by using a put to index themselves as caching an object, and then having other nodes discover these cached copies via a DSHT get. Ideally, each object would only be downloaded from the origin server a single time, even when the system experiences an abrupt flash crowd for a single object/key.

However, the protocols we described above fail to provide this property, even beyond get failures that may occur due to routing problems or packet loss. (These network failures are minimized by our use of multiple outstanding RPCs and requirement that lookups to multiple nodes closest to the key all fail before the higher-lever get fails.) Namely, redundant fetches may occur also because a race condition exists in the protocol.

Consider that a key k is not yet inserted into the system. Two nodes both execute a $get(k)$. Having failed, these nodes fetch the content from the origin server, then optimistically perform a $put(k, self)$ so that other peers can immediate get the data from them. (We discuss such optimistic references in §3.2.3.2.) On the node closest to k, however, the operations may serialize with both gets being handling (and thus returning no values) before either put.

Simply inverting the order of operations is even worse. If multiple nodes first optimistically perform a put, followed by a get, they can discover one another and effectively form cycles waiting for one another, with nobody actually fetching data from the origin server (a form of distributed deadlock).

To eliminate this condition, we extend insert operations to provide return status information, like test-and-set in shared-memory systems. Specifically, we introduce a single put+get RPC which performs both operations. The RPC behaves similar to a put as described above, but also returns the first values discovered in *either* direction to satisfy the "get" portion of the request. (Values returned during the forward put direction help performance, as a put may need to traverse past the first stored values, while a get should return as soon as the first values are found; values returning during the reverse put phase prevent this race condition.)

2.3.2 Coral: Locality-optimized hierarchical DSHTs

Instead of one global overlay, Coral uses several *levels* of DSHTs called *clusters* to achieve locality-optimized routing and data placement. This section describes how Coral implements its put and get DSHT primitives within a hierarchy of clusters, as well as the clustering mechanisms that self-organize this hierarchical structure.

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, our evaluation considers a three-level hierarchy with thresholds of ∞, 60 ms, and 20 ms 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.

Given this support for a hierarchy, Coral slightly modifies the basic indexing layer's interface to expose hierarchy-aware features:

- $put(key, value, ttl, [levels])$: Inserts a mapping from the key to some arbitrary value into *each* cluster of the node's current hierarchy, starting at the highest level (*e.g.,* 2), 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 (default is all levels).

- $get(key, [levels])$: Retrieves some subset of the values stored under a key, starting at the highest level. Again, one can optionally specify a subset of the cluster hierarchy.

- $put{+}get(key, value, ttl, [levels])$: The single put-and-get primitive both inserts a mapping from the key to some arbitrary value on each level and retrieves some subset of values stored under a key. Because a get may complete (by finding *some* values) before a put does—puts may insert at a closer node within a cluster, and puts also need to walk multiple levels of the hierarchy—the response from the get and put functions of this primitive arrive asynchronously.

Figure 2.7 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 layer, in particular to support switching between a node's distinct DSHTs midway through a lookup.[3]

2.3.2.1 Hierarchical retrieval

A sending node s 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. 2.7), 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 r_2 in this cluster (RPC 2), signifying failure at this level. So, node s continues the search onward in its level-1 cluster (RPC 3), using the next target ID specified by constrained routing's splice, having already traversed the ID-space up to r_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 as a function of cluster diameter, and (3) all pointers in higher-level clusters reference data *within* that local cluster.

2.3.2.2 Hierarchical insertion

A node starts by performing a put on its level-2 cluster as the simple DSHT insertion algorithm, so that other nearby nodes can take advantage of locality. However, this placement is only "correct"

[3]We initially built Coral using the Chord [177] routing layer as a block-box; difficulties in maintaining distinct clusters and the complexity of the subsequent system caused us to scrap the implementation.

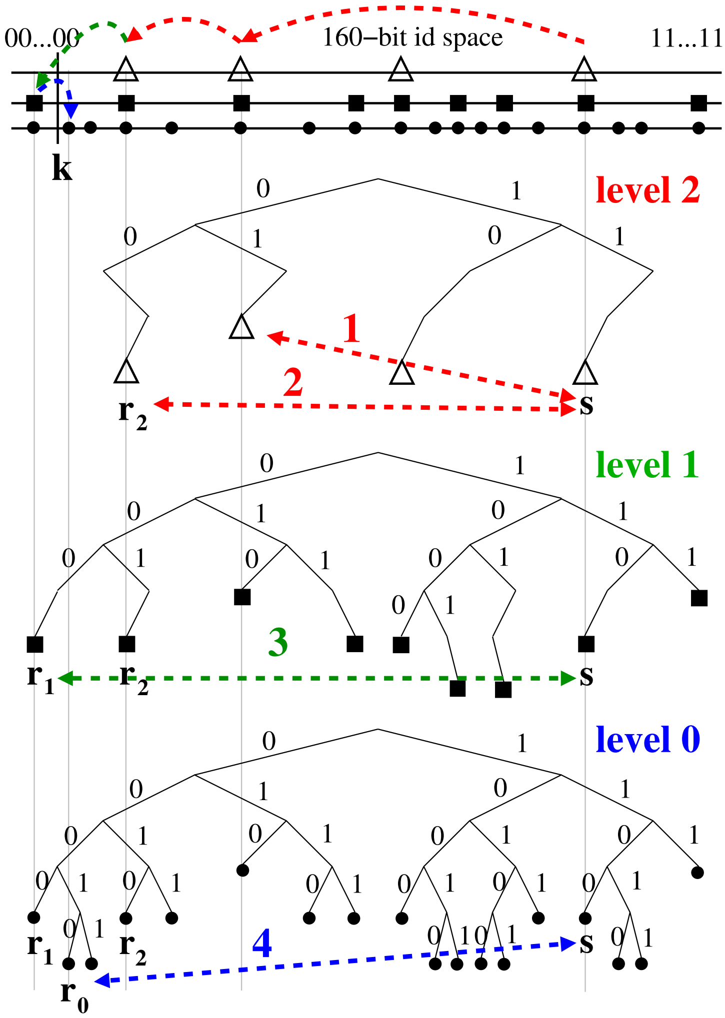


Figure 2.7: *Coral's hierarchical routing structure from the perspective of node s.* Nodes use the same IDs in each of their clusters; higher-level clusters are naturally sparser. Note that a node can be identified in a cluster by its shortest unique ID prefix, *e.g.*, `11` for s in its level-2 cluster; 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. We demonstrate a get request for key k on this structure; get RPCs are numbered sequentially in the figure.

within the context of the local level-2 cluster. Thus, provided that the key is not already loaded and full, 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 2.7, 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 list of clusters to which it belongs. 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, and (2) cluster information is accumulated to drive cluster management, as described next.

2.3.2.3 Joining and managing clusters

To join the Coral network, a Coral node simply needs to contact any existing node in the system (either using some well-known IP address, DNS lookup, or an anycast system per §4). Once a Coral node has joined the network, it can build its routing tables by making several lookup queries to selected keys (especially its own node identifier to explore its local region of the ID-space).

However, for joining non-global clusters, Coral adds one important requirement: A node will only join an *acceptable* cluster, where acceptability requires that the pair-wise RTTs to 80% of the nodes be below the cluster's threshold (a configurable parameter). A node can easily determine whether this condition holds by recording minimum 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 a built-in fast traceroute mechanism to determine the addresses of routers up to five hops out. Excluding any private IP addresses (per RFC-1918 [152]), Coral uses these router /24 prefixes as keys under which to index clustering hints in its DSHTs. More specifically, a node a stores mappings from each router address to its own IP address and UDP port number. When a new node b, sharing a gateway with a, joins the network, it will find one or more of a's hints and quickly cluster with it, assuming a is, in fact, near b.

Nodes continuously collect clustering information from peers: All RPCs provide round-trip-times to one's neighbors and their cluster membership. Each level-i cluster is named by a randomly-chosen 160-bit cluster identifier, while the level-0 cluster ID is predefined as 0^{160}.

Every five minutes, nodes consider changing their cluster membership based on this piggy-backed data. If data indicates that an alternative candidate cluster is desirable, a 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. (The difference threshold for joining and leaving a cluster are used to prevent rapid oscillations in cluster membership.) If probes indicate that 80% of a cluster's members are within acceptable RTTs and the alternate cluster is larger (in the logarithmic scale), a node switches to the new 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 i and j are of similar sizes, inaccurate estimations could lead to oscillation as nodes flow back-and-forth. 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:

$$\left|\log(size_i) - \log(size_j)\right| \;>\; \delta\left(\min(age_i, age_j)\right)$$

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: 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.

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.

2.3.2.4 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 (Chapter 3), to connect to and interface with the Coral daemon. Coral is about 25,000 lines of C++, and it uses the asynchronous I/O library provided by the SFS toolkit [118], with its control flow structured as asynchronous events and callbacks. Coral network communication is via RPC over UDP. It uses the Berkeley DB [171] for persistent key-value storage. We have successfully run Coral on Linux, OpenBSD, FreeBSD, and Mac OS X.

For reference, we now show the makeup of several of Coral's main RPCs. Figure 2.8 shows the header information included in most RPC query and responses. First, header information includes a network identifier as a means of admission control; all packets received with an invalid $netid$ are discarded.[4] Second, headers address both sender and recipient (in node-ID space and with potentially-multiple public IPs to provide support for NAT traversal). Finally, headers also include information about a node's cluster membership at all non-global levels of the hierarchy—cluster ID, level, estimated cluster size, and creation time—which is used for the discovery and selection of new clusters (per §2.3.2.3).

Figure 2.9 provides the routing RPC. A lookup includes the key being requested, the potentially-

[4] Our implementation currently uses a simple globally-shared secret, although a cryptographic keyed hash function $F_{netid}(nodeid_S)$ would provide improved security. In practice, this network identifier is also useful when rolling out new versions of Coral, to ensure that non-wire-compatible versions do not interact with one another during overlapping deployments.

```
struct coral_qry_hdr {       /* RPC request headers                  */
  qry_hdr {
    u_int32 netid;           /* Coral network ID known to S          */
    id srvid;                /* 160-bit nodeID of recipient R        */
    sock_addr srv;           /* Known IP address of R                */
    node_info {              /* Description about S                  */
      id clntid;             /* 160-bit nodeID of S                  */
      vec<sock_addr> a;      /* Public IP address(es) of S           */
      u_int32 flags;         /* Services run on S                    */
    }
  }
  hierarchy_info {           /* Information about sender's clusters  */
    vec<cluster_info> {      /* List of clusters to which S belongs  */
      cluster_desc {        /*   to properly scope request          */
        id clusterid;        /* 160-bit cluster identifier           */
        u_int32 level;       /* Cluster level (1, 2, )               */
      }
      enum policy_type;      /* Type of clustering performed         */
      u_int32 logsize;       /* Log of estimated size of cluster     */
      u_int32 crtime;        /* Creation time in secs (since Epoch)  */
    }
  }
};

struct coral_rsp_hdr {       /* RPC response headers                 */
  rsp_hdr {
    u_int32 net_id;          /* Coral network ID known to R          */
    sock_addr clnt;          /* Known IP address of S                */
    node_info srv;           /* Desc about R, same as in qry_hdr     */
  }
  hierarchy_info hinfo;      /* Same as in coral_qry_hdr             */
};
```

Figure 2.8: Format of Coral RPC headers from sender S to recipient R

```
struct coral_lookup_arg {
  coral_qry_hdr hdr;        /* Request header                       */
  id key;                   /* 160-bit key to lookup values         */
  id target;                /* 160-bit target to route towards      */
  u_int32 win;              /* Return # next-hop nodes near target  */
};

struct coral_lookup_res {
  coral_rsp_hdr hdr;        /* Response header                      */
  u_int32 srv_time;         /* R's clock for time synch issues      */
  vec<level_lookup_res> {   /* Lookup result for levels i=0,1,2,... */
    enum routing_status;    /* Status of routing req (OK, ERR, UNK) */
    enum load_status;       /* How loaded is node for key?          */
    u_int32 min_expiry;     /* Min expiry time for cached values    */
    vec<node_info> closer;  /* Known <win> peers closest to <targ>  */
    vec<dsht_value> values; /* Known values at level i w.r.t. S     */
  }
};
```

Figure 2.9: Format of Coral lookup RPCs

different target ID used in constrained routing (§2.3.1.1), and the *window size* of known neighbors nearest to $target$ that the recipient R should return to the sender S. This RPC is used for both get operations and the "forward direction" of put operations. In its response, the recipient provides information *for each of the clusters on which it matches with* S; namely, the $closer$ peers it knows with respect to $target$, any $values$ it has cached under key, and the load status and min-expiry times of its entries under key. Hierarchically-scoping the returned values is necessary because two nodes in the same level-2 cluster may or may not be in the same level-1 cluster. If they do match on both levels, we want to find all matching values in a single RPC. Finally, the load and expiry information is used to tell S when to terminate the "forward phase" of puts.

Figure 2.10 shows the insert RPC, used during the "reverse phase" of put operations. The query includes the key and DSHT $value$, the latter being the actual data being indexed, its time-to-live ttl, and the inserting node's cluster information. Again, this information is needed to ensure when nodes belong to different clusters at various levels: if multiple nodes insert values under key k, and node S does a lookup on k, node S must determine which of these nodes, if any, belong to

```
struct coral_insert_arg {
  coral_qry_hdr hdr;        /* Request header                         */
  id key;                   /* 160-bit key on which to insert value */
  dsht_value value {        /* Value to be inserted                   */
    opaque data;            /* IP address, URL, node info, etc        */
    vec<cluster_desc> cds;  /* Clusters to which value belongs        */
    u_int32 ttl;            /* TTL of new entry                       */
  }
  bool get;                 /* If true, return prior stored values  */
};

struct coral_insert_res {
  coral_res_hdr hdr;        /* Response header                        */
  vec<level_insert_res> {   /* Results from insertion at each level */
    enum cache_status;      /* Status of R's cache (full, empty, )  */
    bool insert_status;     /* Success at insert                      */
  }
  vec<level_insert_get_res> {
    enum routing_status;    /* Status of routing req (OK, ERR, UNK) */
    vec<dsht_value> values; /* Known values at level i w.r.t. S     */
  }
}
```

Figure 2.10: Format of Coral insert RPCs

the same level-2 cluster as itself, the same level-1 cluster, etc. Finally, the get flag is true iff the request is a $put{+}get$, in which case R's response should include any known values for key. The recipient responds with the result of the insertion request, as well as any cached DSHT values in the case of $put{+}get$ queries.

2.3.3 Evaluation

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

1. Coral naturally forms suitable clusters.

2. Coral prevents hot spots within its indexing system.

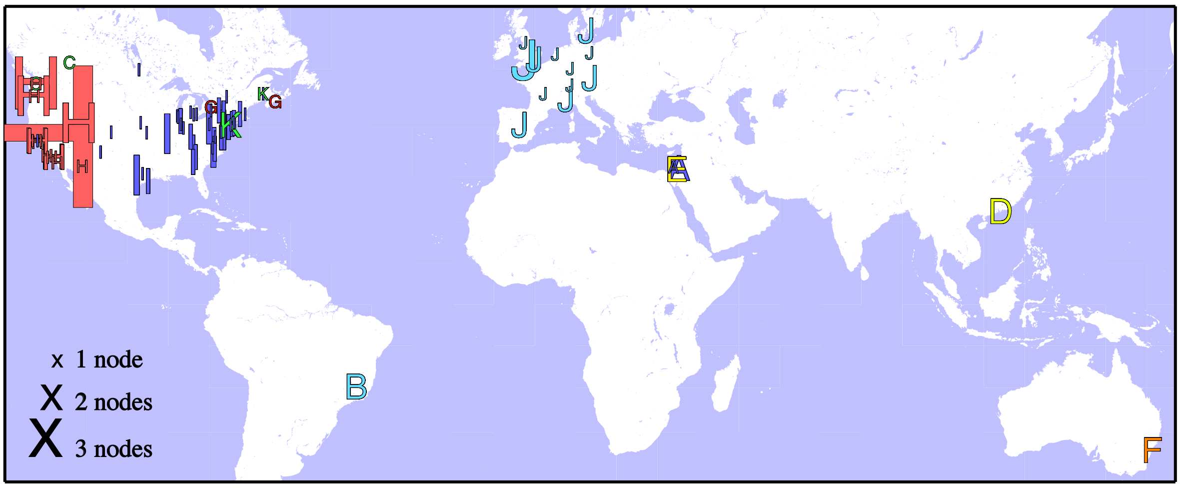
In the next chapter, we present a content distribution network built on the Coral indexing layer and quantify the performance benefit of Coral's hierarchical design.

2.3.3.1 Clustering

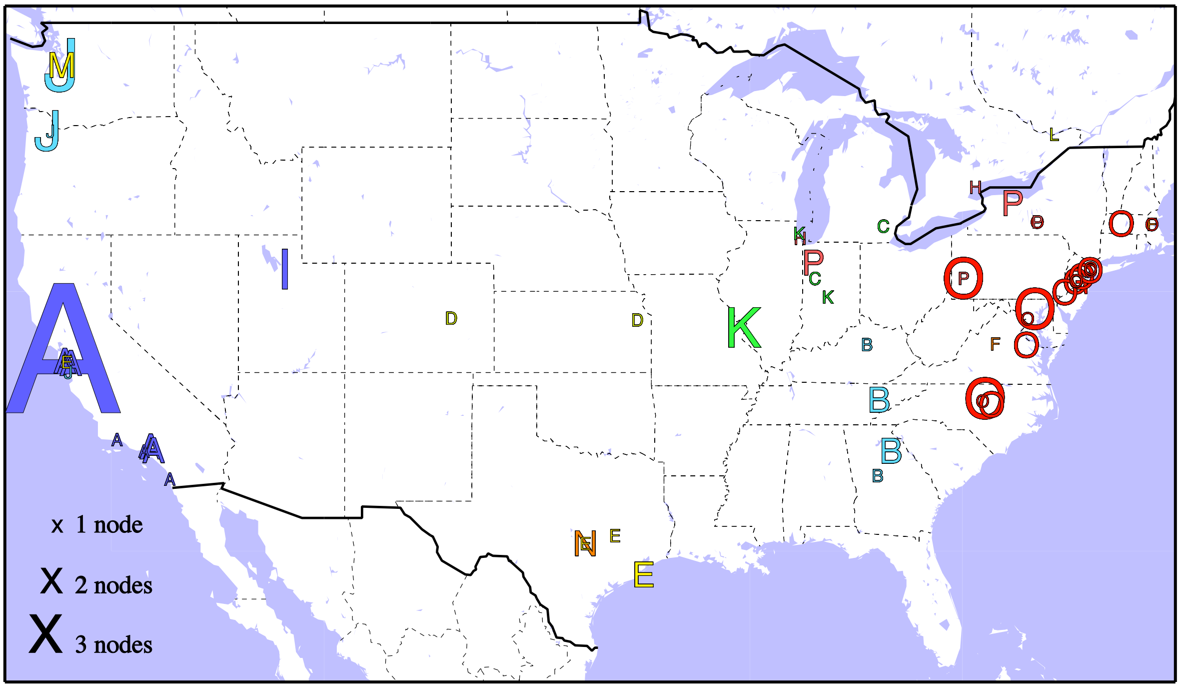
To examine how Coral's clustering mechanism behaves in real operating conditions, we launched a Coral daemon on 166 PlanetLab machines [145], geographically distributed mainly over North America and Europe. These Coral nodes were configured to organize themselves into a three-level hierarchy by setting the clustering RTT threshold of level 1 to 60 ms and level 2 to 20 ms. For simplicity, all nodes were seeded with the same list of well-known hosts for bootstrapping the network. The network was allowed to stabilize for 30 minutes.[5]

Figure 2.11 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.

[5]We should note that 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.



(a) World view of level-1 clusters (60 ms threshold)



(b) United States view of level-2 clusters (20 ms threshold)

Figure 2.11: *Clustering within the Coral deployment on PlanetLab.* Each unique, non-singleton cluster is assigned a letter on the map; the size of the letter corresponds to the number of collocated nodes in the same cluster.

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.

The world map shows that Coral found natural divisions between sets of nodes along geospatial lines at a 60 ms threshold. The map shows several distinct regions, the most dramatic being the Eastern U.S. (70 nodes in "I"), the Western U.S. (37 nodes in "H"), and Europe (19 nodes in "J"). 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 in "A"), the Pacific Northwest (9 nodes in "J"), the South, the Midwest, and so forth. The Northeast Corridor "O" 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.

2.3.3.2 Load balancing

To evaluate whether Coral prevents tree saturation within its indexing system, we instrumented Coral nodes to generate requests at very high rates, all for the same key.

Figure 2.12 shows the extent to which a DSHT balances requests to a single key. In this experiment, we ran three 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 nodes began to issue back-to-back put and get requests at their maximum (non-concurrent) rates. All operations referenced the same

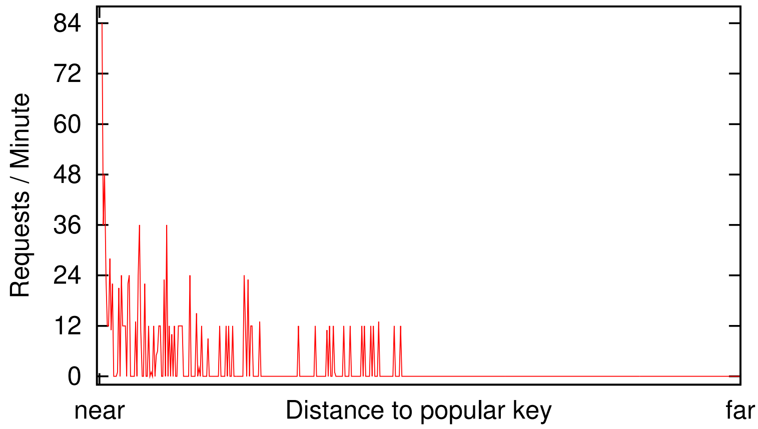


Figure 2.12: *Load balancing per key within Coral.* The total number of put RPCs hitting each Coral node per minute, sorted by distance from node ID to specific target key.

(randomly-chosen) key; the values stored during put requests were randomized. On average, each node issued 400 put and get operation pairs per second, for a total of approximately 12 million put and get requests per minute, although only a fraction hit the network: Once a node is storing a key, get requests are satisfied locally, and once it is *full* and *loaded*, each node only allows the leakage rate β put RPCs "through" it per minute.

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 2.12, the system reached steady-state with the closest node receiving 83 put RPCs per minute. Recall that our equation in §2.3.2 predicts that it should receive $(\beta \cdot \lceil \log n \rceil) = 108$ RPCs per minute. The difference between the analytically-compute load $(12 \cdot 9)$ and the experimentally-observed load of approximately $(12 \cdot 7)$ is due to irregularities in the node ID distribution, as nodes choose their identifiers at random and the analytical results apply to a perfectly-balanced tree. That said, the plot still 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.

2.4 Related work on DHTs and directory services

There is a large body of work in designing DHTs with various algorithmic properties; we review a few here.

A 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 [150], Chord [177], Kademlia [116], Pastry [162], or Tapestry [195]—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 and availability. Gummadi *et al.* [78] shows how all these routing geometries allow some flexibility in neighbor and path selection to provide better routing locality, a generalization of the performance optimization considered in [28, 47]. Accordion [107] dynamically balances routing state with performance, allowing the system to transition from $O(N)$ to $O(\log N)$ state, based on network conditions and a node's available bandwidth.

DHTs can act either as actual data stores or merely as directory services storing pointers. CFS [45] (using the DHash layer in Chord) and PAST [161] take the former approach to build a distributed file system for immutable data. Kelips [79] takes a gossip-based approach to replicate data and provides $O(\sqrt{N})$ performance with $O(\sqrt{N})$ state. Beehive [149] also uses proactive data replication, but can guarantee any asymptotic lookup behavior: Rather than increase the number of neighbors known to peers, they replicate content in the DHT (Pastry) based on observed access patterns.

Using the network as an directory service, Tapestry [195] 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 rate-limiting and constrained routing algorithms prevent multiple stores from forming hot spots.

A few other DHTs have also leveraged a hierarchical structure. But rather than using self-organizing clustering algorithms like Coral, SkipNet [84] builds a hierarchy by explicitly grouping nodes based on domain name in order to support organizational disconnect. Canon takes a similar approach for multiple DHT geometries [65].

To our knowledge, our integration of Coral into the CoralCDN system, described in the next chapter, was the *first* use of a DHT-like structure in a production environment. DHTs have subsequently been used in other peer-to-peer services, *e.g.,* Kademlia [116] serves as the basis for BitTorrent's and Azureus's "trackerless" option [17].

Chapter 3

Building CoralCDN, a Web Content Distribution Network

This chapter couples the Coral indexing layer with a complete system design for scalable and efficient web content distribution. Using CoralCDN, web-site publishers who cannot otherwise afford data centers or commercial CDNs—and hence rather limited in the size of audience and type of content they can normally serve—can reach more clients by leveraging the network and server resources of third parties in a scalable and self-organizing fashion. In other words, CoralCDN helps democratize web content distribution by automating the process of replicating and propagating popular content.

3.1 Motivation

To use CoralCDN, a content publisher, end-host client, or someone posting a link to a high-traffic portal simply appends "`.nyud.net`" to the hostname in a URL. Through DNS redirection, clients with unmodified web browsers are transparently redirected to nearby CoralCDN web caches. These caches cooperate to transfer data from nearby peers whenever possible, thus utilizing the aggregate bandwidth of participants running the software to absorb and dissipate most of the traffic for websites using the system. In doing so, CoralCDN minimizing both the load on the origin webserver, and it may even improve the end-to-end latency experienced by clients.

CoralCDN is built on top of the Coral key-value indexing layer, described previously in §2.3. 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 indexing infrastructure, even under degenerate loads.

Taken together, the resulting system enables people to publish content that they previously could not or would not because of distribution costs. To our knowledge, CoralCDN is the first decentralized and self-organizing web-content distribution network, and it contains the first peer-to-peer DNS redirection infrastructure, allowing the system to interoperate with unmodified web browsers.

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

CoralCDN has been deployed on 300-400 servers on the PlanetLab network [145] since March 2004. For the past two years, it has served an average of 20-25 million requests for more than two terabytes of web content to over one million unique client IPs per day. In doing so, it has allowed websites to scale to otherwise intractable audience sizes, while needed near zero changes to origin websites to achieve this result.

This chapter describes the usage scenarios of CoralCDN and its system design (§3.2), including its HTTP proxy and DNS server. We evaluate the system in §3.3, before discussing additional engineering mechanisms that we introduced in response to operating challenges learned during the course of CoralCDN's deployment (§3.4). We conclude by discussing related CDN work (§3.5).

3.2 Design

CoralCDN is comprised of three main parts: (1) a network of cooperative HTTP proxies that handle users' requests,[1] (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 rely.

3.2.1 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 website publisher can change selected URLs in their web pages to so-called "Coralized" URLs—*e.g.,* modify the URL `http://www.example.com/` to `http://www.example.com.nyud.net/`—either through static or dynamic rewriting.

- **Third-parties.** Any 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 in the web page to use CoralCDN as well.

- **Users.** CoralCDN-aware users can manually construct Coralized URLs (or automatically via installed software or browser plugins) when surfing slow or overloaded websites. Based on the way these URLs are modified, any HTTP redirects or relative links within returned web pages are automatically Coralized.

3.2.2 System overview

Figure 3.1 shows the steps that occur when a client accesses a Coralized URL, such as `http://www.example.com.nyud.net/`, using a standard web browser.

[1]While CoralCDN's HTTP proxy definitely provides proxy functionality, it is not an HTTP proxy in the strict RFC-2616 [54] sense. Rather, it serves requests that are syntactically formatted for an ordinary HTTP server.

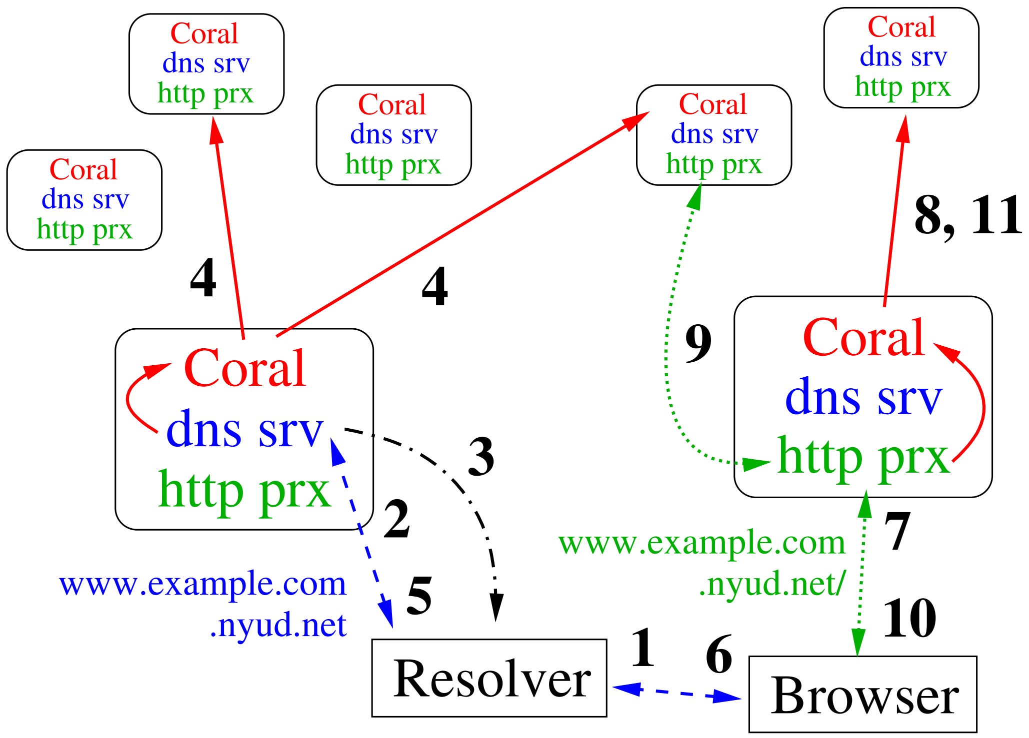


Figure 3.1: *Steps involved in using CoralCDN.* The client first resolves a Coralized URL, then contacts the resulting HTTP proxy to download the content. Rounded boxes represent CoralCDN nodes running Coral, DNS, and HTTP servers. Solid arrows correspond to Coral RPCs, dashed arrows to DNS traffic, dotted-dashed arrows to network probes, and dotted arrows to HTTP traffic.

1. A client sends a DNS request for `www.example.com.nyud.net` to its local resolver.
2. The client's resolver attempts to resolve the hostname using some CoralCDN DNS server(s), possibly starting at one of the few (~12) registered under the `.net` domain.
3. Upon receiving a query, the 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 §3.2.4).
6. The client's resolver returns the address of a CoralCDN HTTP proxy for the Coralized domain `www.example.com.nyud.net`.
7. The client sends the HTTP request `http://www.example.com.nyud.net/` to the specified proxy. If the proxy is caching the corresponding 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 begins fetching the object from this node. Otherwise, the proxy begins downloading the object from the origin server, `www.example.com` (this step is not shown in the figure).
10. The proxy begins storing the web object and returning it to the client browser.
11. The proxy stores a reference to itself in Coral, recording the fact that is now caching the URL so that its cached copy can be made available to other proxies in the future.

This section describes the design of CoralCDN's HTTP proxy and original DNS redirector, especially with regard to their use of the Coral indexing layer. In the deployed CoralCDN service, this original Coral-specific DNS service has been replaced with the use of a general-purpose anycast service we present in Chapter 4 as the preferred method of proxy selection. However, we still present this original design in order to discuss its observed shortcomings in §3.4.4, which motivated and influenced our subsequent redesign.

3.2.3 The CoralCDN HTTP proxy

The CoralHTTP proxy 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 point in the design space requires particular care in minimizing load on origin servers compared to traditional CDNs, for two reasons. First, many of CoralCDN'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 they select a client's proxy based in part on the URL requested—effectively partitioning objects across proxies—often having pre-provisioned their customer's content. CoralCDN, in contrast, selects proxies only based on client locality and does not seek to control data placement by design.

In this section, we describe how CoralHTTP proxy is designed to use the Coral indexing layer for effective inter-proxy cooperation and for rapid adaptation to flash crowds.

3.2.3.1 Locality-optimized inter-proxy transfers

To aggressively minimize load on origin servers, a CoralHTTP proxy 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, CoralHTTP proxy 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 the URL. Only after the Coral indexing layer provides no referrals or none of its referrals return the data, CoralHTTP proxy attempts to fetch the resource directly from the origin.

If a CoralHTTP proxy discovers that one or more other proxies have the data, it attempts to open TCP connections to multiple other proxies in parallel (currently configured to two), and issues an HTTP request to the first proxy to which it successfully connects. These pre-established TCP connections also provide faster failover if the initial request to a different CoralHTTP proxy fails—

e.g., the CoralHTTP proxy has already evicted the content from its cache, returns an unexpected error message, or is unresponsive at the application layer—at which time the initiating CoralHTTP proxy will issue a new HTTP request to an open connection while simultaneously opening additional TCP connections to maintain the appropriate window size for failover, provided additional proxies are known. Once a neighboring proxy begins to send back valid content, all outstanding TCP connections are closed.

Note that these inter-proxy transfers are locality-optimized, both from their use of parallel connection requests and, more importantly, by the order which neighboring proxies are contacted. The properties of Coral's hierarchical indexing layer implies that the list of proxy IP addresses returned by $get(\text{SHA-1}(url))$ will be sorted based on their cluster distance to the request initiator, preferring level-2 neighbors before level-1 and level-0 nodes, respectively.

3.2.3.2 Rapid adaptation to flash crowds

Unlike many web proxies, CoralHTTP proxy is explicitly designed for flash crowd scenarios: If a flash crowd suddenly fetches a web object, all CoralHTTP proxies, other than the first simultaneous requests to the origin server, will naturally form a kind of "multicast tree" for retrieving the web page: Data streams from the proxies that initially fetch the object from the origin server to those arriving later. CoralCDN provides such behavior by combining optimistic references and cut-through routing.

As soon as a CoralHTTP proxy begins receiving the first bytes of a web object—either from the origin or from another CoralHTTP proxy—it inserts a reference to itself in the Coral indexing layer with a short time-to-live (30 seconds). It continually renews this short-lived references until either it completes the download or the download fails.

CoralCDN's cut-through routing at each proxy helps reduce transmission time for larger files. In other words, proxies begin uploading content as soon as it is downloaded, not waiting until they receive the entire file.[2] Once any CoralHTTP proxy successfully finishes downloading the content, it inserts much longer-lived reference to itself. Because the Coral insertion algorithm accounts for

[2]The CoralHTTP proxy implements such behavior through its event-based structure communicating through the file system. Multiple client connections can register read callbacks on a single file, and whenever

expiry times (per §2.3.1.2),[3] these longer-lived references will overwrite shorter-lived ones, and they can be stored on well-selected nodes even under high insertion load. CoralHTTP proxies renew their pointers stored in Coral to these cached objects every time-to-live period, but only provided that the content has been requested subsequent to its initial download event.

A CoralHTTP proxy manages its disk using a least-recently-used eviction policy (each proxy is configured with some maximum-allocated disk space, which is 3-4 GB per proxy on our PlanetLab deployment). Note, however, that the LRU eviction policy can cause a proxy to evict an item, even while a reference in Coral still exists.[4] Recovery from such failures is handled by our use of multiple HTTP requests for failover, described above. In practice, however, the working set of our deployed system experiences sufficiently slow turnover as to make such a situation relatively rare and thus have little impact.

3.2.4 The (original) CoralCDN DNS server

The CoralDNS server returns IP addresses of CoralHTTP proxies when browsers look up the hostnames of Coralized URLs. To improve locality, it attempts to return proxies near requesting clients. In particular, whenever a DNS resolver (client) contacts a nearby CoralDNS server instance, the CoralDNS server seeks to both return proxies within an appropriate cluster and ensure that future DNS requests from that client will not leave the cluster. CoralDNS also exploits on-the-fly network measurement and stores topology hints in Coral to increase the chances of clients discovering nearby DNS servers.

More specifically, every instance of CoralDNS is an authoritative nameserver for the do-

this file is written to as content is received from a remote node, these connection callbacks are dispatched and the newly-written content is sent to other remote nodes downstream in the tree.

[3]The deployed system uses two-hour TTLs for "successful" transmissions (*e.g.,* status code results including 200 (OK), 301 or 302 (Moved), etc.). It also uses 15-minute TTLs for non-transient "unsuccessful" transmissions (*e.g.,* 403 (Forbidden), 404 (File Not Found), etc.) in order to perform negative-result caching across the network.

[4]Coral does not provide any mechanism to delete a reference. If a *delete* primitive was only best-effort, an application relying on such a function would still need to be able to handle the cases in which the *delete* failed. On the other hand, a *delete* primitive that provided strong consistency would require much heavier-weight protocols than Coral is designed to support.

main `nyucd.net`. Assuming a 3-level hierarchy, as Coral is generally configured, CoralDNS maps any domain name ending `http.L2.L1.L0.nyucd.net` to one or more CoralHTTP proxies. (For an $(n+1)$-level hierarchy, the domain name is extended out to Ln in the obvious way.) Because such names are somewhat unwieldy, we established a DNS DNAME alias [44], `nyud.net`, with target `http.L2.L1.L0.nyucd.net`: This translates to the DNS rule that any domain name ending with the suffix `nyud.net` is equivalent to the same name with the longer suffix `http.L2.L1.L0.nyucd.net`, allowing Coralized domain names to be more concise.

The CoralDNS servers assume 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 it good enough that Akamai [2], Mirror Image [123], and LimeLight Networks [108] have all successfully deployed commercial CDNs based on DNS redirection. (In fact, we later present one of the first quantitative analyses of client-DNS resolver proximity in Figure 4.6 of §4.2.2.2.) The locality problem therefore is reduced to returning proxies that are near the source of a DNS request. In order to achieve locality, a CoralDNS server measures its round-trip-time to the client's resolver (via ICMP and TCP probes) 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 (*e.g.,* $\leq$ 20ms, $\leq$ 60ms, and $>$ 60ms, respectively).

When asked for the address of a hostname ending `http.L2.L1.L0.nyucd.net`, the CoralDNS server'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. CoralDNS server returns addresses of CoralHTTP proxies in the cluster whose level corresponds to the client's level categorization. In other words, if the RTT between the DNS resolver and CoralDNS server is below the level-i threshold (for the best i), CoralDNS server will only return addresses of Coral nodes in its level-i cluster (we extended Coral's interface to provide a $nodes(level)$ primitive, which returns a randomly-chosen subset of peers known to a node within a specified-level cluster). CoralDNS always returns CoralHTTP proxy addresses with short time-to-live fields (30 seconds for levels 0 and 1, 60 for level 2).

To achieve better locality, a CoralDNS server also attempts to find proxies near to the client's DNS resolver based on network hints. That is, it probes the addresses of the last five network hops to the resolver and uses the results to look for clustering hints stored in the Coral index. To avoid significantly delaying clients, it maps these network hops using a fast, built-in traceroute-like mechanism that combines concurrent probes and aggressive timeouts to minimize latency. The entire mapping process generally requires around two RTTs and 350 bytes of bandwidth. The server caches these measurement results to avoid repeatedly probing the same client.

The closer a CoralDNS server is to a client, the better its selection of CoralHTTP proxy addresses will likely be for the client. The CoralDNS server therefore exploits the authority section of DNS replies to "lock" a DNS client into a good cluster whenever it happens upon a nearby CoralDNS server. As with the answer section, a CoralDNS server selects the nameservers it returns from the appropriate cluster level and exploits measurement and network hints as well. Unlike addresses in the answer section, however, it gives nameservers in the authority section a long TTL (one hour). A nearby CoralDNS server must therefore override any inferior nameservers that a DNS client may be caching from previous queries; it does so by manipulating the domain for which returned nameservers are servers. To clients more distant than the level-1 timing threshold, CoralDNS server 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 CoralDNS server 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 CoralHTTP proxy addresses with short TTL fields.) As an added precaution, a CoralDNS server only returns CoralHTTP proxy addresses which it has recently verified first-hand. This sometimes means synchronously checking a proxy's liveness status (via a UDP RPC) prior to replying to a DNS query.

3.3 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.

Recall that we showed earlier in §2.3.3 that Coral (1) naturally forms suitable clusters and (2) prevents hot spots within its indexing system.

To examine these two claims, we present wide-area measurements of a synthetic work-load on CoralCDN nodes running on PlanetLab. We use such an experimental setup because traditional tests for CDNs or webservers 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 a webserver's throughput on disk-bound access patterns, while CoralCDN is designed to reduce load on off-the-shelf webservers that are *network-bound*.

The basic structure of the experiments were the same as in §2.3.3. First, on each of 166 PlanetLab machines geographically distributed mainly over North America and Europe, we launch a Coral daemon, as well as a CoralDNS server and CoralHTTP proxy. For experiments referred to as *multi-level*, we configure a three-level hierarchy by setting the clustering RTT threshold of level 2 to 20 ms and level 1 to 60 ms. Experiments referred to as *single-level* use only a single (level-0) global cluster. No objects are evicted from CoralHTTP proxy caches during these experiments.

Next, we run an unmodified Apache webserver sitting behind a DSL line with 384 Kbit/s upstream bandwidth, serving 12 different 41 KB 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

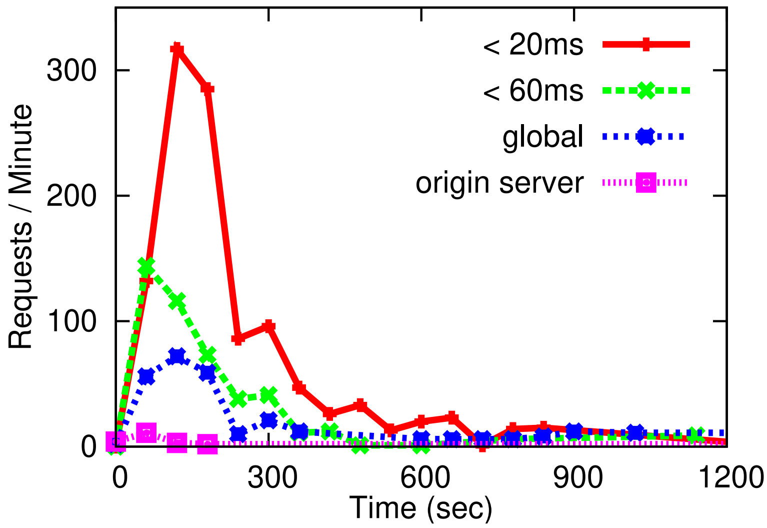


Figure 3.2: *CoralCDN reducing server load.* The number of client accesses to CoralCDN proxies and the origin HTTP server. Proxy accesses are reported relative to the cluster level from which data was fetched, and do not include requests handled through local caches.

web page, for a period of 30 minutes. While we recognize that web traffic generally has a Zipf distribution [19], we are attempting merely to simulate a flash crowd to a popular web page with multiple, large, embedded images. With 166 clients, we are generating 99.6 requests/second, resulting in a cumulative download rate of approximately $32,800$ Kb/s. 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. (Indeed, our current deployment on PlanetLab sees 200-300 Mb/s of traffic on average, although this again is not a technical maximum, as the PlanetLab deployment is artificially limited by bandwidth-shaping mechanisms, discussed in §3.4.2.)

3.3.1 Server load

Figure 3.2 plots the number of requests per minute that could not be handled by a CoralHTTP proxy's local cache. During the initial minute, a total of 15 requests hit the origin webserver (for the 12 unique files, shown by dotted purple line with squares). The 3 redundant lookups are due to the simultaneity at which requests are generated;[5] 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 an origin webserver.

During this first minute, equal numbers of requests were handled by the level-2 (solid red line with crosses) and level-1 (dashed green line with x's) cluster caches. However, as the files propagated into CoralHTTP proxy 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.

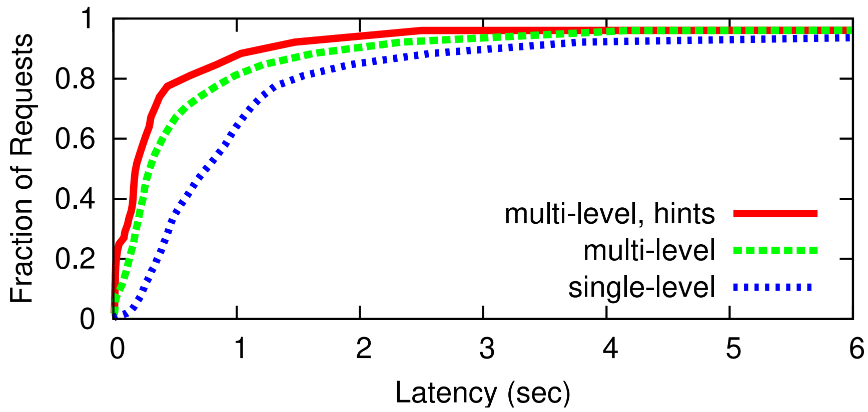
3.3.2 Client latency

Figure 3.3 shows the end-to-end latency for a client to fetch a file from CoralCDN, following the steps given in §3.2.2. The top graph shows the cumulative distribution function (CDF) of latency across all PlanetLab nodes used in the experiment. The bottom graph only includes data from the clients located on five nodes in Asia: Hong Kong (two), 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.

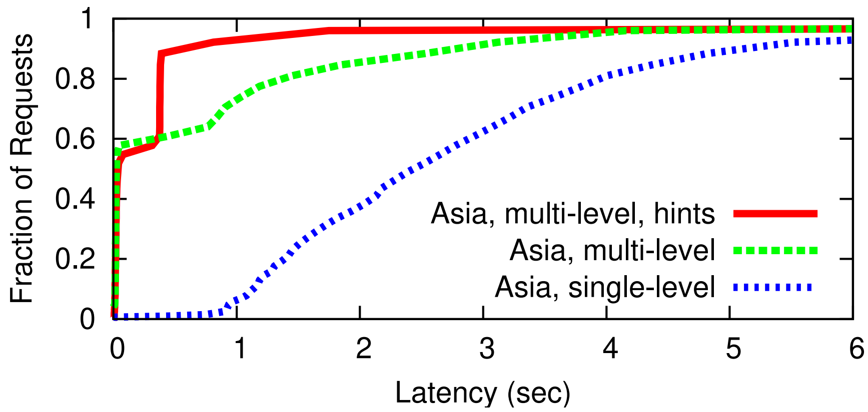
Recall that this end-to-end latency includes the time for the client to make a DNS request and to connect to the discovered CoralHTTP proxy. The contacted proxy attempts to fulfill the client request first through its local cache, then through Coral, and finally through the origin web server. Because the CoralHTTP proxy implements cut-through routing, clients begin receiving a file as soon as the proxy begins downloading it.

These figures report three results: (1) the distribution of latency of clients using only a single level-0 cluster (the dotted blue line), (2) the distribution of latencies of clients using multi-level

[5]This race condition may be eliminated through the single *put+get* primitive of §2.3.1.4. Although now used in our live PlanetLab deployment, this extension to the Coral indexing layer was not available at the time of these experiments.



(a) Performance of nodes world-wide



(b) Performance of nodes in Asia

Request latency (seconds)	**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 3.3: 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.

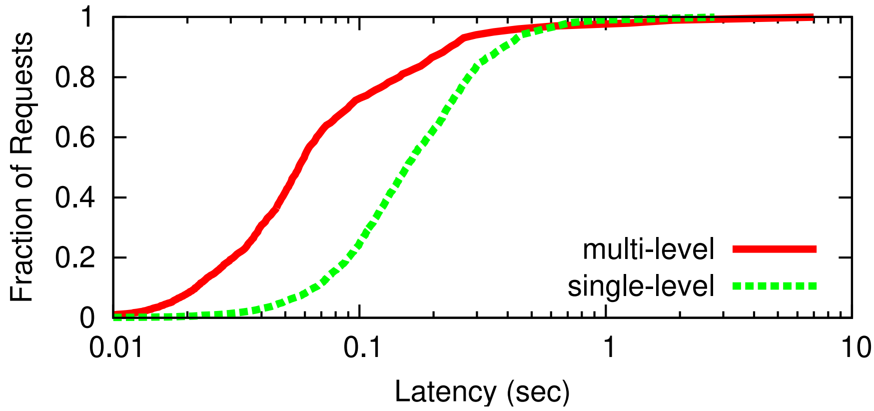


Figure 3.4: Latencies for proxy to get keys from Coral

clusters (the dashed green line), and (3) the same hierarchical network, but using traceroute during DNS resolution to map clients to nearby proxies (the solid red line).

All clients ran on the same subnet (and host, in fact) as a CoralHTTP proxy in our experimental setup. This would not be the case in the real deployment: We would expect a combination of hosts sharing networks with CoralHTTP proxies—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 3.4 shows the latency of get operations, as seen by CoralHTTP proxies when they lookup URLs in Coral (Step 8 of §3.2.2). We plot the get latency on the single level-0 system versus 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—which PlanetLab can (excessively) experience—the multi-level system requires a few more timeouts as it traverses its hierarchy levels.

3.4 Deployment lessons

CoralCDN has been deployed on 300-400 servers on the PlanetLab network [145] for over three years. For the past two years, it has served an average of 20-25 million requests for more than two terabytes of web content to over one million unique client IPs per day. This section describes some of the mechanisms developed and lessons learned during the course of CoralCDN's deployment.

3.4.1 Robustness mechanisms

Unlike many web proxies, a CoralHTTP proxy regularly interacts with overloaded or poorly-behaving origin servers, and our explicit goal is to keep content available in the face of such conditions.

For example, a fairly normal situation for CoralCDN is the following: First, a portal like Slashdot [170] or digg [50] runs a story that links to a third-party website, driving a sudden influx of readers to this previously unpopular site. Second, a user posts a Coralized link to the same website in a "comment" to the portal's story, providing an alternate means to fetch the content. (Similarly, several third-party browser plugins automatically Coralize their users' links for either all web content or for that of special pages, *e.g.,* Slashdot.) Several things can occur given these situations, all of which should be handled differently.

- The website's origin server(s) become unavailable before CoralCDN proxies can download a copy of the content.
- CoralCDN already has a copy of the content, but requests arrive to CoralCDN after the content's *expiry* time has passed; subsequent HTTP requests to the origin webserver result in failures or error statements.
- CoralCDN already has a copy of the content, but requests arrive to CoralCDN after the content's *expiry* time has passed; subsequent requests to the origin server yield only partial transfers.

We next consider how CoralCDN handles these different situations.

Negative result caching. CoralCDN may be hit with a flood of requests for a URL that is not accessible for a variety of reasons (*i.e.,* DNS resolution fails, the webserver does not respond to TCP connection requests, etc.). For these situations, CoralCDN maintains a negative result cache, whereby after repeated failures, CoralHTTP proxies will *locally* cache information that certain hostnames will not resolve or establish TCP sessions. Otherwise, we have seen excessive load and resource exhaustion on both CoralHTTP proxies and the proxies' local DNS resolvers given flash crowds to apparently dead sites (*e.g.,* because the proxy would need to hold open client connections for several seconds while attempting to resolve or connect to the specified hostname). Notice that, unlike when an origin server returns a permanent status-code error, this failure status is only cached locally, as it may be caused by a local network or DNS failure at the proxy and not at the origin server at all.

Serving stale content. CoralCDN proxies obey the expiry times of content, as specified in `Cache-Control` (HTTP/1.1) or `Expires` (HTTP/1.0) header.[6] (If no expiry time is specified, CoralHTTP proxies default to twelve hours.) As such, after content has expired, CoralHTTP proxies will attempt to perform a conditional (`If-Modified-Since`) request. If the content has not changed, of course, these requests minimize overhead given that the origin can simply respond with a 304 (Not Modified) message and not resend the same body content.

What happens, however, if the origin server does not respond at all, or simply responds with some temporary error condition? Rather than return an error to the client, a CoralHTTP proxy will return stale content instead (and update its expiry timer again). Specifically, if the origin responds with a 403 (Forbidden), 404 (Not Found), 408 (Timeout), 500 (Internal Server Error), or

[6] The only caveat is that a CoralHTTP proxy sets a *minimum* expiry time of some duration (currently configured as five minutes), and thus does not recognize `No-Cache` directives as such. Note, however, because CoralCDN does not support cookies, SSL bridging, or POSTs, most of the privacy concerns normally associated with caching such content are alleviated. Our anecdotal experience suggests many sites use such directives to provide their sites with complete HTTP logs (*e.g.,* for counting page views), not for privacy reasons. We note that because CoralCDN does not re-write web pages, the origin site will see still requests for any absolute URLs embedded in the page; thus, normal tricks with web-beacons can provide their sites with accurate logs.

503 (Service Unavailable) message, a CoralHTTP proxy will serve stale data for up to 24 hours (configurable) after it expires.

This decision is, of course, a trade-off that will not satisfy every situation. Is a Forbidden message due to the website publisher seeking to make the content unavailable, or it is caused by the website going over its daily bandwidth quota and its hosting service returning an error? Does a 404 indicate whether the condition is temporary (from a `php` or database error) or permanent (from a third-party issuing a take-down notice to the website)? (Indeed, such ambiguity led to the introduction of 410 (Gone) messages in HTTP/1.1, denoting permanence, which does result in the eviction of content from our caches.) We have seen all these situations; the problem is simply that many status-code responses are inherently ambiguous. But we found the above behavior to satisfy both our users' desire to robustly get content, as well as our publishers' desire to control caching behavior. Unfortunately, we have also seen many situations caused by semantically-incorrect server-side results as well, often generated by poorly-written `php` or other server scripts: Too often do our servers receive a 200 (OK) message with humanly-readable body content to the tune of "an error occurred," resulting in CoralCDN replacing valid content with such useless information.

Whole-file overwrites. Finally, consider when the CoralHTTP proxy is already caching an expired file, but a subsequent re-request yields a partial or excessively-slow response from the origin site (due to the fact that it is being overloaded). Rather than having the proxy lose a valid copy of a stale file, CoralHTTP proxies perform *whole-file overwrites*: Namely, new versions of content are written to temporary files; only after the file completes downloading and appears valid (*e.g.,* based on `Content-Length`) will a CoralHTTP proxy replace its existing copy with this new version of the data.

3.4.2 Bandwidth-shaping fairness mechanisms

Operators that run a CoralHTTP proxy would like to be able to configure the node's total bandwidth usage. This demand arose even on CoralCDN's use of the PlanetLab platform, which is comprised largely of university sites: Without any bandwidth-limited mechanism, CoralCDN's bandwidth use spiked immediately following the Asian tsunami of December 2004, as CoralCDN was being used to distributed many amateur videos of the natural disaster. PlanetLab sites were threatening to pull their servers offline if such use could not be curtailed.

While PlanetLab has implemented its own kernel-level bandwidth-limiting mechanism, this is not sufficient for our goals. First, such limitations are done on a per-"slice" and per-packet basis, *i.e.,* they will indiscriminately limit *all* traffic from a service such as CoralCDN, not intelligently choosing between different CoralCDN requests (flows) and communication types (system maintenance traffic versus client requests). Second, such a technique still sees CoralCDN processing requests and sending data in response. These response packets, however, are subsequently being queued (and ultimately dropped) in the kernel, causing CoralCDN servers to perform much useless work.

Instead, a CoralHTTP proxy implements its own application-level bandwidth-shaping mechanism, where it attempts to provide *fairness* across websites using CoralCDN on a per-domain basis. We have found that the majority of CoralCDN's bandwidth is consumed by a relatively small subset of websites, as the primary adoption of CoralCDN is server-side, not client-side. Thus, it is important that any bandwidth limits are also applied on a server-side basis. (We do have some limited mechanisms to prevent the malicious overuse of CoralCDN's bandwidth by clients, but such misuse occurs relatively rare, is easy to prevent, and accounts for only a small portion of CoralCDN's total bandwidth.)

Our fairness mechanisms ultimately attempt to ensure that a CoralHTTP proxy only sends some upper-bounded amount of traffic per day in steady-state (on PlanetLab, our configured limit is 10 GB per node per day). We attempt to balance three goals. Firstly, CoralCDN will get flash crowds and changing traffic patterns, so it needs that the hard limits for peak traffic differs from steady-state limits to allow short bursts of unexpected activity. Secondly, however, our primary consumers of bandwidth are sites that have permanently adopted CoralCDN, not short-term flash crowds caused by a link being posted on some web portal. Thus, we also seek to ensure that *normal* traffic patterns to a site domain are limited over short time-scales, such that the expected usage over longer time-scales is acceptable. Finally, we want this accounting to be *fair*: Network

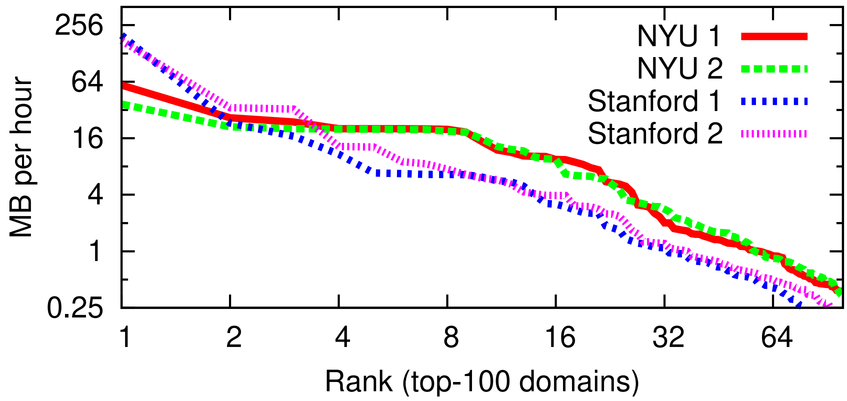


Figure 3.5: Average bandwidth consumed by the top-100 sites at four CoralCDN proxies on August 2, 2007.

wide, CoralCDN currently distributes approximately 2 TB per day, while our demand sometimes exceeds 10 TB per day. The difference between these two limits—caused by only a small number of bandwidth hogs—is due to CoralCDN rejecting a significant number of requests. But it does so by rejecting the bandwidth hogs, not indiscriminate requests.

Figure 3.5 plots the amount of hourly traffic that four CoralHTTP proxies permitted per domain for the top-100 bandwidth hogs: Note that the graphs (in logarithmic scale) show that a relatively small number of domains use most of the bandwidth. Indeed, it is only a handful of domains towards the left-side of this graph that will see any of their requests rejected due to excessive use. Due to our fairness mechanisms, the remaining will be unaffected.

CoralCDN implements the following algorithm on each CoralHTTP proxy. The system is configured with three maximum rates. $maxpeak$ provides a *hard*-upper bound of the maximum traffic can be sent per time period (our deployment sets $maxpeak$ to 1000 MB per each one-hour period per node); $maxsteady$ provides an upper bound on the *expected* amount of total traffic in steady state per time period, calculated as an exponentially-weighted moving average (set to 400 MB per hour); $maxsteadyper$ provides this upper bound *per* domain (200 MB per hour).

During each time period, we record the number of bytes transmitted for *every* domain (bw). At the end of every time period, we calculate and store the average bandwidth usage (bw^{avg}) of the top-k sites at time t as an exponentially-weighted moving average:

$$\forall i = 1 \ldots k \quad : \quad bw_{i,t}^{avg} = \left(\frac{\alpha - 1}{\alpha} \cdot bw_{i,t-1}^{avg} \right) + \left(\frac{1}{\alpha} \cdot bw_{i,t} \right) \tag{3.1}$$

where $\alpha = 168$ and time periods are one hour in our deployment, thus serving to smooth bandwidth usage out over a one week period.

We then calculate the maximum dynamically-adapted steady-state traffic allowed per host: Let $lim_t = (\frac{\alpha-1}{\alpha} \cdot lim_{t-1}) + (\frac{1}{\alpha} \cdot maxpeak)$; reduce lim_t until a point whereby if that limit had been enforced during the previous period (see below), the expected bandwidth use $\sum_{i=1 \ldots k} bw_{i,t}^{avg}$ would be $\leq maxsteady$.

For any particular request, a CoralHTTP proxy rejects it (with a Forbidden message) if and only if (1) the current time period has too much total peak traffic across all domains ($\sum_{\forall i} bw_{i,t} \geq maxpeak$), ***or*** (2) the marginal bandwidth cost of the request (calculated as in Equation 3.1) would exceed either the total steady-state static limit $maxsteady$, ***or*** the domain's specific static limit $maxsteadyper$, ***or*** the domain's dynamically-adapted limit lim_t. Any request for content ≤ 1 KB in size is allowed, however, given that bandwidth and cycles processing the small request will have already been spent.

Note that this mechanism is applied independently on each node. It may be an interesting piece of future work to enable (approximate) accounting across the entire network, yet do so without centralization and in a bandwidth-efficient manner (*i.e.,* with $o(kN)$ communication complexity for N-node networks).

In the above description, a CoralHTTP proxy simply rejects a request when the site is over its bandwidth allotment. Operators of some sites, on the other hand, have expressed their preference that such requests instead be redirected back to their site and allow their server(s) to deal with the additional load, as opposed to rejecting the request outright. To specify such functionality, sites can include a `X-Coral-Control` header in their server's HTTP response—additional meta-data that a CoralHTTP proxy caches along with the object—that details how CoralHTTP proxy should handle such overload conditions. (If clients are redirected back to the origin server, a CoralHTTP proxy appends an additional query-string `coral-no-serve` on their request. This helps ensure that the origin site knows that it should serve the request, as opposed to creating a request loop by dynamically redirecting the client back to CoralCDN again.)

3.4.3 Security mechanisms

CoralHTTP proxies also incorporate a number of security mechanisms to prevent misuse. This section discusses some of the security protections *and limitations* of CoralCDN's current deployment.

Limited functionality. CoralCDN proxies only support GET and HEAD requests. Therefore, many of the attacks for which "open" proxies are infamous [134] are simply not feasible. For example, clients cannot use CoralCDN to POST passwords for brute-force cracking. CoralHTTP proxies do not support CONNECT requests, and thus they cannot be used to resend email as an SMTP relay, to forge "From" addresses in web mail, or to spam IRC networks. Furthermore, because CoralCDN only handles Coralized URLs, it cannot be used directly by pointing a browser's proxy settings at it. Although this certainly does not offer any real security, it results in CoralCDN not appearing on most "open-proxy lists" found on the Web, thus reducing CoralCDN's exposure.

Domain blacklisting. We currently maintain a global domain-name blacklist that each CoralHTTP proxy regularly fetches and reloads during run-time. This blacklist is used to satisfy origin servers that do not wish their content to be accessed via CoralCDN. Such restrictions are especially important for sites that perform IP-based authentication, as CoralHTTP proxies may be located within IP blocks that have been whitelisted by such authorities (this is true for many academic journals, for instance, as the deployed CoralHTTP proxies mostly run on university networks).

The technical need for such blacklists is not apparent, however; certainly any modern webserver could reject CoralCDN requests based on its unique `User-Agent` string ("`CoralWebPrx`"), as CoralCDN does not attempt to anonymize itself as an end-user. Indeed, CoralCDN HTTP requests include both `Via` and `X-Forwarded-For` headers as well. (In fact, we show in [26] how web proxies can be detected as such in real-time, even when they do not include required proxy headers.) In practice, however, many site operators are either unwilling or ill-trained to perform such security checks themselves, and instead issue abuse complaints to operators whose networks host CoralCDN proxies (*i.e.,* PlanetLab sites). Our use of such blacklists was necessary to enable CoralCDN to continue to run in such networks.

Interestingly, we have found users that have circumvented these *domain*-based blacklists by creating dynamic DNS records for random domains that point to the IP addresses of the desired target, *e.g.,* an online journal. In these cases, the CoralHTTP proxy check on the domain passes, but the request is sent through to the origin site. (An IP-based blacklist, on the other hand, could stop this attack but instead would have to be concerned with origin sites having multiple or dynamic IP addresses.) It is interesting to note that such circumvention techniques would not be a problem if origin sites would verify the `Host` field of the HTTP request, *i.e.,* it corresponds to a valid name for the website, not a randomly-generated one as above. Unfortunately, again, such security measures are often not taken.

In the longer term, research towards supporting individualized blacklists at proxies would help enable deployments under more decentralized control.

Excessive use and recursive looping. As we allude to with the discussion of CoralCDN's bandwidth-shaping techniques, CoralHTTP proxies keep a sliding window of client-side usage patterns. Not only do we seek to prevent excessive bandwidth usage by clients, but also an excessive number of (even small) requests; these are usually caused either by "bots" or recursive HTTP redirection loops.

Bots occasionally attempt to use CoralCDN to hide their identity while attempting to gain login access to various servers. However, as CoralCDN both does not support POSTs and forwards the client's origin IP address in the `X-Forwarded-For` header, its usefulness in such settings is certainly questionable.

We have found multiple instances of recursive looping from misconfigured servers that use dynamic URL rewriting to Coralize URLs, *i.e.,* respond to requests for `http://example.com/` with an HTTP redirect to `http://example.com.nyud.net/`. Some sites choose always to use such rewriting so that their logs continue to reflect all client requests. Others do so in more inventive ways: for example, if and only if the request's `Referer` header is from a portal such as Slashdot or digg, suggesting a potential flash crowd; or by combining such rewriting with load detection, so that their servers only start rewriting requests in order to shed load that they otherwise could not handle.

When using dynamic rewriting, it is critical that sites check that the request is not from a CoralHTTP proxy itself; otherwise, an HTTP redirect will be cached and returned to the client, causing a loop (and the real content to be unreachable). For example, a regular-expression rule for Apache's `mod_rewrite` would look like the following:

```
RewriteEngine on
RewriteCond %{HTTP_USER_AGENT} !^CoralWebPrx
RewriteCond %{QUERY_STRING} !(^|&)coral-no-serve$
RewriteRule ^/(.*)$ http://example.com.nyud.net/$1 [R,L]
```

where the second line checks if the client is a CoralCDN proxy and the third-line checks if the request is not due to the "failover" condition described in §3.4.2. While CoralHTTP proxies do perform a simple check for looping—will the `Location` header returned cause a simple loop—we have found more complex, multi-hop loops to occur in practice (mostly through misconfigurations, not malice). Thus, tracking a client's request frequency at a proxy also helps protect CoralCDN against such conditions.

Amusingly, some users early during CoralCDN's deployment caused recursion in a different way: by submitting URLs with many copies of `nyud.net` in a hostname's suffix: L repeated instances of `nyud.net` would cause a CoralHTTP proxy to make L recursive requests, stripping the last instance off in each iteration. CoralHTTP proxies now verify that URLs do not contain such expressions.

No cookies. CoralCDN does not support cookies; it deletes any `Cookie` or `Set-Cookie` HTTP headers in both directions. Many websites now manage and set cookies via javascript, however, as opposed to via HTTP headers. This could be problematic from a privacy standpoint if the CoralHTTP proxies were running on untrusted clients, especially for "third-party" cookies that could expose linkages between a client's use of multiple sites. Even worse, however, is that the "same-origin" RFC-2965 security policies [103] are ill-suited for liberal configuration settings

	Jan 1	Mar 1	May 1	Jul 1
Level 0	49.6%	60.0%	53.6%	92.2%
Level 1	44.8%	36.3%	37.4%	7.6%
Level 2	5.6%	3.6%	9.0%	0.5%

Figure 3.6: Percentage of DNS responses for selected dates in 2007 that captured locality using the original CoralDNS servers.

in cookies when combined with CoralCDN's domain-name modifications and untrusted proxies. Specifically, if a site (say, `a`) sets their cookie's domain as `.nyud.net` instead of `a.nyud.net`, any other site `b.nyud.net` can read `a`'s `.nyud.net` cookies.

3.4.4 Failures of original DNS design

CoralCDN's DNS mechanism used two methods for server selection: (1) looking up topological hints in the indexing layer and (2) encoding the cluster level in the returned DNAME alias, so that if a client ever randomly finds a nearby nameserver, it will "stick" with it. Neither approach worked particularly well.

First, the topological hints only work if a CoralHTTP proxy is deployed at a site local to the client. Given that CoralCDN's user population is spread world-wide, yet CoralCDN is sparsely deployed—only at a few hundred sites, mostly university networks—fewer than 1% of clients were found to be local. Thus, not only is such a technique ineffective, but it also requires synchronous, on-demand probing, yielding both abuse complaints from and unnecessary delays for clients.

Second, most clients interact with CoralCDN to download only a few web objects; client sessions are relatively short. Thus, while the CoralDNS server approach of random walks to find a nearer nameserver will take only a few iterations in expectation, it is less common that the same resolver will make multiple DNS requests before its cached `nyud.net` NS records expires and it needs to start-over at a random nameserver. (This is compounded by the fact that many browsers *pin* name-to-IP mappings for a fixed duration (several minutes), no matter the A record's TTL. Such pinning was introduced to prevent some DNS rebinding attacks [90]). Furthermore, the CoralDNS servers were unable to estimate the locality of a majority of DNS requests (53.6%), after both ICMP and TCP probes failed to return a valid RTT estimate, given the existence of on-path firewalls and other middleboxes that confound such measurements.[7] In total, Figure 3.6 shows the percentage of real DNS requests that took advantage of nameserver locality using the original CoralDNS server: between 50% and 90% took advantage of no locality, while fewer than 9% used CoralHTTP proxies that were within the local cluster (*i.e.,* a 20 ms diameter from the CoralDNS server).

Third, the use of DNAME records proved problematic due to forward-compatibility bugs in some DNS resolvers (*e.g.,* in Windows 2000 and 2003 servers). Per RFC-1123 [18], if DNS resolvers receive a response with a mix of resource record types, they should accept the known types and preserve the unknown types for downstream transmission as opaque data. Unfortunately, some resolvers reject the entire DNS response if it includes an unsupported DNAME record (type 39), making CoralCDN appear unreachable. Thankfully, most clients do not appear to use such DNS resolvers; common implementations such as BIND support these record types. As a (ugly) workaround, the CoralDNS server began returning a DNAME record probabilistically (one-third of requests), and it otherwise synthesized CNAME records as aliases. (A CNAME record will only map a specific name `a` to name `a.b`, as opposed to the "wildcard" aliasing—any name `*` to `*.b`—provided by DNAMEs. This wildcard matching allows a client's resolver to satisfy requests for different Coralized domains using already cached information.) With such probabilistic behavior by the CoralDNS server, broken resolvers will need to retry one-third of the time, while working resolvers will cache the day-long-TTL DNAME record once it is received. Unfortunately, this does not handle every case: if a broken resolver recurses through a working resolver (*e.g.,* BIND) before it queries a CoralDNS server, the intermediate resolver will cache and propagate such DNAME records correctly, leading back to the problematic condition.

In the next chapter, we introduce an alternative design for DNS redirection. Based on these above experiences, this new system both avoids *any* on-demand probing (and minimizes probing in general), and it returns nearby servers upon the client's *first* request.

[7] In fact, this difficulty in performing active measurement led to our subsequent work on opportunistic client measurement [26]

3.5 Related work on web 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 [33, 53, 64, 112]. 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 [189], 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 [174], and PROOFS uses a randomized overlay to distribute popular content [175]. 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 [89]. 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 [36].

Akamai [2], LimeLight [108], 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 [173]. Then, upon reaching a cluster of collocated machines, hashing schemes [94, 180] map requests to specific machines in order to increase their usable storage capacity. These systems require deploying large numbers of highly-provisioned servers, and typically result in very good performance (both latency and throughput) for customers. CoralCDN offers less aggregate storage capacity than such networks, 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 (*e.g.,* LimeLight builds out their network by deploying higher numbers of servers at still relatively few data centers).

At about the same time as CoralCDN's deployment, CoDeeN started providing users with a set of open web proxies [42, 186]. Users can reconfigure their browsers to use a CoDeeN proxy and subsequently enjoy better performance. The system has also been deployed on PlanetLab, and it has been enjoyed similar success at distributing content efficiently. Although CoDeeN gives *participating* users better performance to *most* websites, CoralCDN's goal is to gives *most* users better performance to *participating* websites—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. More recently, CobWeb [172] has taken a similar approach to CoDeeN, but using proactive replication for managing content [149]. This approach can yield high performance, but, much like CoDeeN, it has the key-value layer control data placement on participating nodes, as opposed to just using an indexing system such as Coral to leverage nodes' underlying access patterns.

Chapter 4

Building OASIS, an Anycast System for Server Selection

This chapter focuses on a flexible solution for the server-selection problem faced by all CDN services. However, instead of designing an anycast system to serve a single service—as the original CoralCDN DNS servers (§3.2.4) were designed to do—the next system we present, OASIS, may be used by many services simultaneously. In fact, OASIS incentivizes participation as its accuracy improves and cost per participant decreases as more services adopt OASIS. OASIS again demonstrates our general approach to building democratizing systems: leverage resources across many unreliable or untrusted resources, wherever they may be found. While CoralCDN utilizes bandwidth and storage resources, OASIS leverages participants for *information* that may be gleaned from their unique and distinct network vantage points.

4.1 Motivation

When a client interacts with a distributed system such as CoralCDN, the performance and cost of these systems depend highly on the client's choice of servers. File download times can vary greatly based on the locality and load of the chosen service instance, or server *replica*. A service provider's costs may depend on their servers' load spikes, as many collocation data centers charge customers based on the 95th-percentile usage over all five-minute periods in a month.

Unfortunately, common techniques for replica selection produce sub-optimal results. Asking human users to select the best replica is both inconvenient and inaccurate. Round-robin and other primitive DNS techniques spread load, but do little for network locality.

Indeed, using peers to cooperatively distribute content should not happen haphazardly. For example, recent studies have shown that a overwhelming percentage of wide-area transfers in peer-to-peer file sharing are unnecessary: up to 90% of this ingress traffic at edge networks corresponds to data already residing at local hosts [93]. Such a situation goes directly to the need for better locality- and load-award mechanisms for content discovery and server selection, even more so when the choice of content sources is large.

In response, more accurate techniques for server selection have been developed. When a legacy client initiates an *anycast* request—*i.e.,* give me one from the set of all servers—these techniques typically probe the client from a number of vantage points, and then use this information to find the closest server. While more sophisticated efforts, such as virtual coordinate systems [46, 129] and on-demand probing overlays [167, 191], have sought to reduce the probing overhead, the savings in overhead comes at a cost of accuracy.

Nevertheless, significant on-demand probing is still necessary for all these techniques, and this overhead is reincurred by every new deployed service. While on-demand probing potentially offers greater accuracy, it has several drawbacks that we experienced first-hand when deploying CoralCDN. First, probing adds latency, which can be significant for small web requests. Second, performing several probes to a client often triggers intrusion-detection alerts, resulting in abuse complaints. This mundane problem can pose real operational challenges for a deployed system.

This chapter presents OASIS (*O*verlay-based *A*nycast *S*ervice *I*nfra*S*tructure), a shared server-selection infrastructure. OASIS is designed to be used by many services simultaneously: Services that adopt OASIS provide increasing numbers of network vantage points, resulting in improvements in OASIS's locality accuracy and decreases in its network probing cost per participating replica.

OASIS is organized as an infrastructure overlay, providing high availability and scalability. OASIS allows a service to register a list of servers, then answers the general query, "Which server should the client contact?" Selection is primarily optimized for network locality, but also incorporates liveness and load.

To eliminate on-demand probing when clients make anycast requests, OASIS probes clients in the background. One of OASIS's main contributions is a set of techniques that makes it practical to measure the entire Internet in advance. By leveraging the locality of IP prefixes—as learned through our previous measurement research [61]—OASIS probes only each prefix, not each client. In practice, IP prefixes from BGP dumps are used as a starting point. OASIS delegates measurements to the service replicas themselves, thus amortizing costs (approximately 2–10 GB/week) across multiple services, resulting in an acceptable per-node cost.

To share OASIS across services and to make background probing feasible, OASIS requires *stable network coordinates* for maintaining locality information. Unfortunately, virtual coordinates tend to drift over time. Thus, since OASIS seeks to probe an IP prefix as infrequently as once a week, virtual coordinates would not provide sufficient accuracy. Instead, OASIS stores the geographic coordinates of the replica closest to each prefix it maps.

OASIS is publicly deployed on PlanetLab [145] and has already been adopted by about a dozen services. We have implemented a DNS redirector that performs server selection upon hostname lookups, thus supporting a wide range of unmodified client applications. We also provide an HTTP and RPC interface to expose its anycast and locality-estimation functions to OASIS-aware hosts [130].

Experiments from our deployment have shown rather surprisingly that the accuracy of OASIS is competitive with Meridian [191], currently the best on-demand probing system. In fact, OASIS performs better than all replica-selection schemes we evaluated across a variety of metrics, including resolution and end-to-end download times for simulated web sessions, while incurring much less network overhead.

Keyword	Threads	Msgs	Keyword	Threads	Msgs
abuse	198	888	ICMP	64	308
attack	98	462	IDS	60	222
blacklist	32	158	intrusion	14	104
block	168	898	scan	118	474
complaint	216	984	trojan	10	56
flood	4	30	virus	24	82

Figure 4.1: *PlanetLab abuse complaints.* Frequency count of keywords in PlanetLab *support-community* archives from 14-Dec-04 through 30-Sep-05, comprising 4682 messages and 1820 threads. Values report number of messages and unique threads containing keyword.

4.2 Design

An anycast infrastructure like OASIS faces three main challenges. First, network peculiarities are fundamental to Internet-scale distributed systems. Large latency fluctuations, non-transitive routing (§2.2.4), and middleboxes such as transparent web proxies, NATs, and firewalls can produce wildly inaccurate network measurements and hence suboptimal anycast results.

Second, the system must balance the goals of accuracy, response time, scalability, and availability. In general, using more measurements from a wider range of vantage points should result in greater accuracy. However, probing clients on-demand increases latency and may overemphasize transient network conditions. A better approach is to probe networks in advance. However, services do not know which clients to probe *a priori*, so this approach effectively requires measuring the whole Internet, a seemingly daunting task.

A shared infrastructure, however, can spread measurement costs over many hosts and gain more network vantage points. Of course, these hosts may not be reliable. While structured peer-to-peer systems [162, 177] can, theoretically, deal well with unreliable hosts, such protocols add significant complexity and latency to a system and break compatibility with existing clients. For example, DNS resolvers and web browsers deal poorly with unavailable hosts since hosts cache stale addresses longer than appropriate.

Third, even with a large pool of hosts over which to amortize measurement costs, it is important to minimize the rate at which any network is probed. Past experience [59] has shown us that repeatedly sending unusual packets to a given destination often triggers intrusion detection systems and results in abuse complaints. For example, PlanetLab's *support-community* mailing list receives thousands of complaints yearly due to systems that perform active probing; Figure 4.1 lists the number and types of complaints received over one ten-month period. They range from benign inquiries to blustery threats to drastic measures such as blacklisting IP addresses and entire netblocks. Such measures are not just an annoyance; they impair the system's ability to function.

This section describes how OASIS's design tackles the above challenges. A two-tier architecture (§4.2.1) combines a reliable core of hosts that implement anycast with a larger number of replicas belonging to different services that also assist in network measurement. OASIS minimizes probing and reduces susceptibility to network peculiarities by exploiting *geographic coordinates* as a basis for locality (§4.2.2.2). We assume that every replica knows its latitude and longitude, which already provides some information about locality before any network measurement. Then, in the background, OASIS estimates the geographic coordinates of every netblock on the Internet. Because the physical location of IP prefixes rarely changes [153], an accurately pinpointed network can be safely re-probed very infrequently (say, once a week). Such infrequent, background probing both reduces the risk of abuse complaints and allows fast replies to anycast requests with no need for on-demand probing.

4.2.1 System overview

Figure 4.2 shows OASIS's high-level architecture. The system consists of a network of *core* nodes that help *clients* select appropriate *replicas* of various services. All services employ the same core nodes; we intend this set of infrastructure nodes to be small enough and sufficiently reliable so that every core node can know most of the others. Replicas also run OASIS-specific code, both to report their own load and liveness information to the core, and to assist the core with network measurements. Clients need not run any special code to use OASIS, because the core nodes provide DNS- and HTTP-based redirection services. An RPC interface is also available to OASIS-aware clients.

Though the three roles of core node, client, and replica are distinct, the same physical host

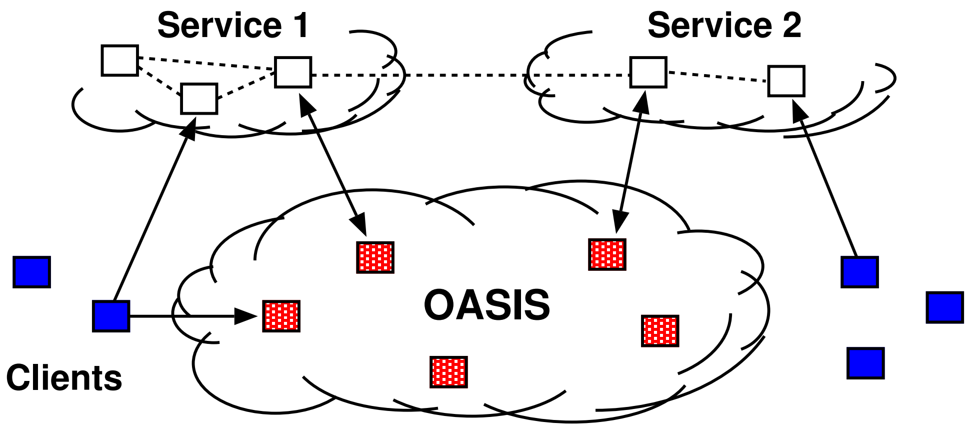


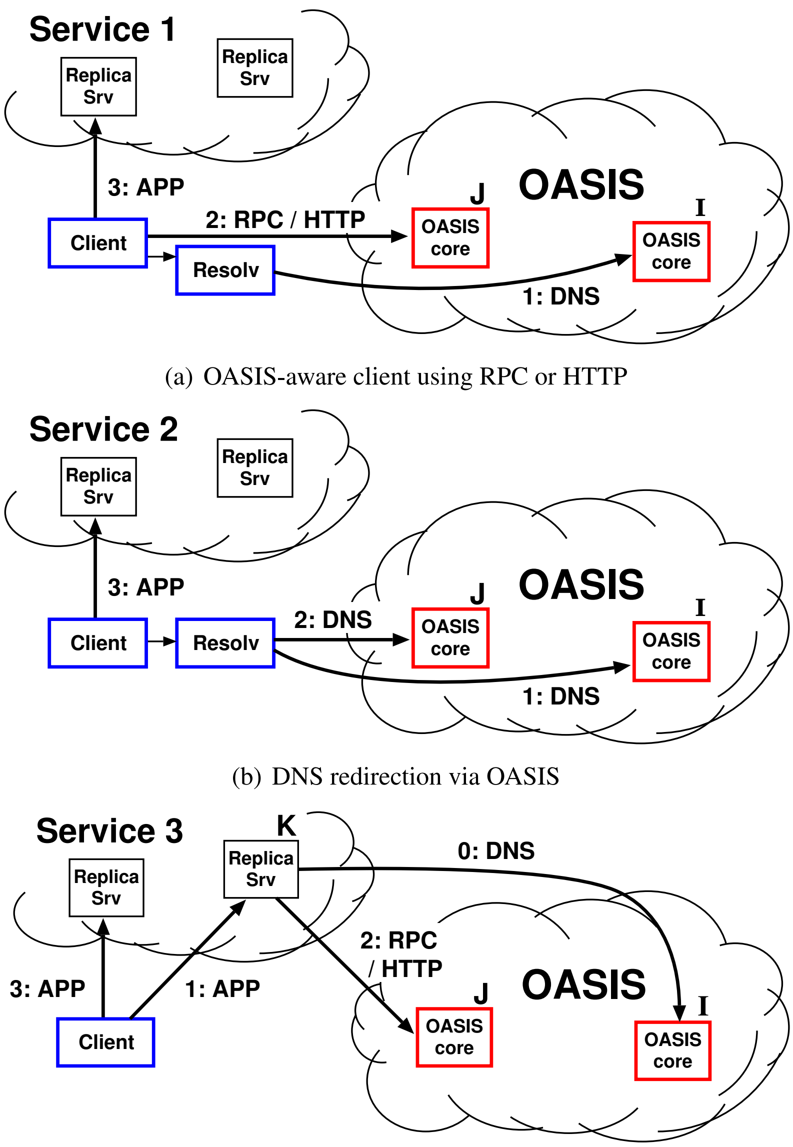
Figure 4.2: OASIS system overview

often plays multiple roles. In particular, core nodes are all replicas of the OASIS RPC service, and often of the DNS and HTTP redirection services as well. Thus, replicas and clients typically use OASIS itself to find a nearby core node.

Figure 4.3 shows various ways in which clients and services can use OASIS. The top diagram shows an OASIS-aware client, which uses DNS-redirection to select a nearby replica of the OASIS RPC or HTTP service (*i.e.,* a core node), then queries that node to determine the best replica of Service 1.

The middle diagram shows how to make legacy clients select replicas using DNS redirection. The service provider advertises a domain name served by OASIS. When a client looks up that domain name, OASIS first redirects the client's resolver to a nearby replica of the DNS service (which the resolver will cache for future accesses). The nearby DNS server then returns the address of a Service 2 replica suitable for the client. This result can be accurate if clients are near their resolvers, which is often, although not always, the case (*e.g.,* per [114] and 4.2.2.2).

The bottom diagram shows a third technique, based on service-level (*e.g.,* HTTP) redirection. Here the replicas of Service 3 are also clients of the OASIS RPC service. Each replica connects to a nearby OASIS core node selected by DNS redirection. When a client connects to a replica, that replica queries OASIS to find a better replica, then redirects the client. Such an approach does not require that clients be located near their resolvers in order to achieve high accuracy.



(a) OASIS-aware client using RPC or HTTP

(b) DNS redirection via OASIS

(c) Service-level redirection via OASIS

Figure 4.3: Various methods of using OASIS via its DNS, HTTP, or RPC interfaces, and the steps involved in each anycast request.

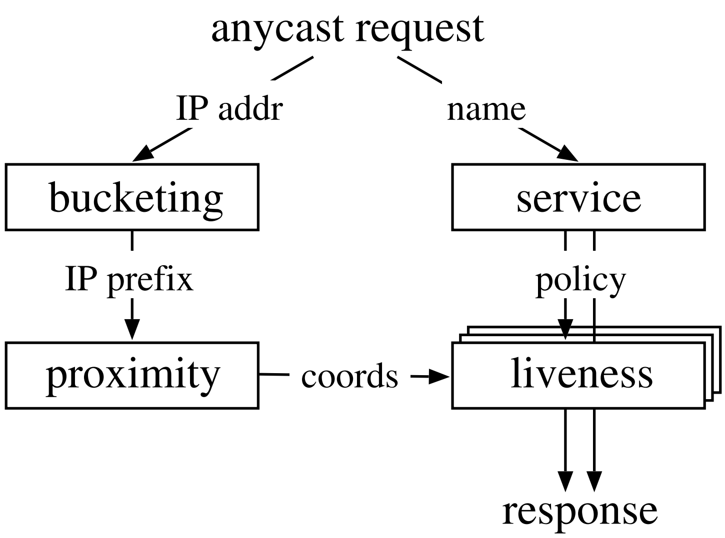


Figure 4.4: Logical steps to answer an anycast request

In the remainder of this section, we largely focus on DNS redirection, since it is the easiest to integrate with existing applications.

4.2.2 Design decisions

Given a client IP address and service name, the primary function of the OASIS core is to return a suitable service replica. For example, an OASIS nameserver calls its core node with the client resolver's IP address and a service name extracted from the requested domain name (*e.g.,* `coralcdn.nyuld.net` indicates service *coralcdn*).

Figure 4.4 shows how OASIS resolves an anycast request. First, a core node maps the client IP address to a *network bucket*, which aggregates adjacent IP addresses into netblocks of collocated hosts. It then attempts to map the bucket to a *location* (*i.e.,* coordinates). If successful, OASIS returns the closest service replica to that location (unless load-balancing requires otherwise, as described in §4.2.3.4). Otherwise, if it cannot determine the client's location, it returns a random replica.

The anycast process relies on four databases maintained in a distributed manner by the core:

(1) a *service table* lists all services using OASIS (and records policy information for each service), (2) a *bucketing table* maps IP addresses to buckets, (3) a *proximity table* maps buckets to locations, and (4) one *liveness table per service* includes all live replicas belonging to the service and their corresponding information (*e.g.,* coordinates, load, and capacity).

4.2.2.1 Buckets: The granularity of mapping hosts

OASIS must balance the precision of identifying a client's network location with its state requirements. One strawman solution is simply to probe every IP address ever seen and cache results for future requests. Many services have too large a client population for such an approach to be attractive. For DNS redirection, probing each DNS resolver would be practical if the total number of resolvers were small and constant. Unfortunately, measurements at DNS root servers [99] have shown many resolvers use dynamically-assigned addresses, thus precluding a small working set.

Fortunately, our previous research has shown that IP aggregation by prefix often preserves locality [61]. For example, of the autonomous systems we analyzed, more than 99% of $/24$ IP prefixes announced by stub autonomous systems (and 97% of $/24$ prefixes announced by all autonomous systems) are at the same location. Thus, we aggregate IP addresses using IP prefixes as advertised by BGP, using BGP dumps from RouteViews [160] as a starting point.[1]

However, some IP prefixes (especially larger prefixes) do not preserve locality [61]. OASIS discovers and adapts to these cases by splitting prefixes that exhibit poor locality precision,[2] an idea originally proposed by IP2Geo [133]. Using IP prefixes as network buckets not only improves scalability by reducing probing and state requirements, but also provides a concrete set of targets to *precompute* and hence avoids on-demand probing.

[1]For completeness, we also note that OASIS currently supports aggregating by the less-locality-preserving autonomous system number, although we do not present the corresponding results in this chapter.

[2]We deem that a prefix exhibits poor locality if probing different IP addresses within the prefix yields coordinates with high variance.

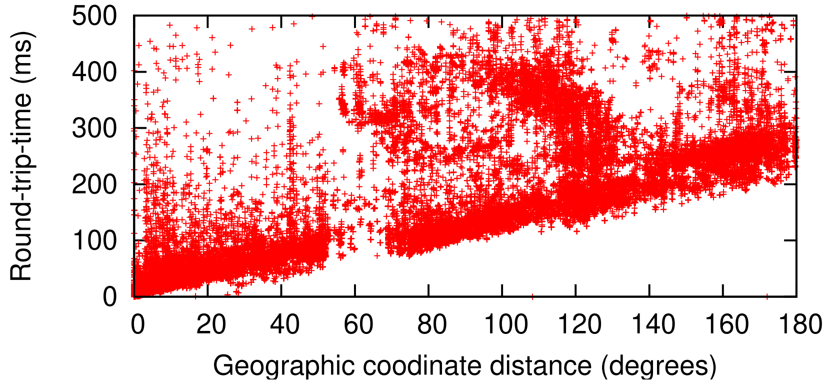


Figure 4.5: Correlation between round-trip-times and geographic distance across all PlanetLab hosts [178].

4.2.2.2 Geographic coordinates for location

OASIS takes a two-pronged approach to locate IP prefixes: We first use a direct probing mechanism [191] to find the replica closest to the prefix, regardless of service. Then, we represent the prefix by the geographic coordinates of this closest replica and its measured round-trip-time to the prefix. We assume that all replicas know their latitude and longitude, which can be easily obtained from a variety of online geocoding services given knowledge of a replica's street-level or city-level location.

While geographic coordinates are certainly not optimal predictors of round-trip-times, they work well in practice: The heavy band in Figure 4.5 shows a strong linear correlation between geographic distance and RTT. In fact, anycast only has the weaker requirement of predicting a relative ordering of nodes for a prefix, not an accurate RTT estimation. For comparison, we also implemented Vivaldi [46] and GNP [129] coordinates within OASIS; §4.3 includes some comparison results.

More importantly, however, as OASIS is fundamentally designed as a shared service, it has the potential for dense deployments. That is, the more third-party services that adopt OASIS, the more vantage points OASIS has with which to perform network measurement and probing (*i.e.,* using those services' replicas). This is a virtuous cycle, as higher accuracy may incentivize more services

CDF percentile	Distance (km)
50.0	5.74
75.0	299.59
90.0	681.14
95.0	1055.74
99.0	3885.64
99.9	12195.10

Figure 4.6: Distance (km) between clients and their DNS resolver

to join OASIS, which again leads to denser deployments: as such, OASIS will operate towards the lower-left-side of Figure 4.5, where there are relatively few outliers in which geographic distance is not a good predictor of network latency. In the limit, OASIS will gain a vantage point in nearly every significant network, and thus minimize the potential networks for which it does not have a good coordinate prediction.

Finally, at least for DNS redirection, there is a limit to the accuracy that any anycast system can hope to achieve, as DNS redirection is done with respect to the client's DNS resolver's IP address, not that of the client. Figure 4.6 shows the measured distance between clients and their DNS resolvers. (This large-scale study of clients we performed [26] included 1.78 million pairs of clients and their DNS resolvers; for this study, we got access to Quova's IP geolocation database [147].[3]) We found that while clients are often close to their resolvers, the association is not perfect: 10% are more than 680 km apart, while 1% are over 3800 km apart.[4] Thus, attempts to find more accurate representations of Internet locality may yield diminishing results.

The advantage of our two-pronged approach—denoting geographic coordinates with their RTT accuracy—is several-fold:

[3]It's important to recognize that Quova is attempting to solve a similar geolocation problem as OASIS. Quova's solution, however, requires the use of limited, manually-deployed vantage points and significant human operator analysis and expertise, while OASIS attempts to build a much larger network of participating replicas to leverage for network measurement and to perform geolocation in an automated fashion from known locations. Furthermore, Quova focuses solely on being a data provider, while OASIS builds a complete and flexible anycast system.

[4]Of course, this evaluation is only as accurate as Quova's geolocation database which, while an industry leader, cannot be easily evaluated in terms of *absolute* accuracy in any statistically-significant manner.

Time- and service-invariant coordinates. Since geographic coordinates are stable over time, they allow OASIS to probe each prefix infrequently. Since geographic coordinates are independent of the services, they can be shared across services—an important requirement since OASIS is designed as a shared infrastructure. Geographic coordinates remain valid even if the closest replica fails. In contrast, virtual coordinate systems [46, 129] fall short of providing either accuracy or stability [167, 191]. Similarly, simply recording a prefix's nearest replica—without its corresponding geographic coordinates—is useless if that nearest replica fails. Such an approach also requires a separate mapping per service.

Absolute error predictor. Another advantage of our two-pronged approach is that the RTT between a prefix and its closest replica is an *absolute* bound on the accuracy of the prefix's estimated location. This bound suggests a useful heuristic for deciding when to re-probe a prefix to find a better replica. If the RTT is small (a few milliseconds), reprobing is likely to have little effect. Conversely, reprobing prefixes having high RTTs to their closest replica can help improve accuracy when previous attempts missed the best replica or newly-joined replicas are closer to the prefix. Furthermore, a prefix's geographic coordinates will not change unless it is probed by a closer replica. Of course, IP prefixes can physically move, but this happens rarely enough [153] that OASIS only expires coordinates after one week. Moving a network can therefore result in sub-optimal predictions for at most one week.

Sanity checking via constraint satisfaction. A number of network peculiarities can cause incorrect network measurements. For example, a replica behind a transparent web proxy may erroneously measure a short RTT to some IP prefix, when in fact it has only connected to the proxy. Replicas behind firewalls may believe they are pinging a remote network's firewall, when really they are probing their own. Or a malicious replica or service may attempt to purposely inject incorrect locality information into OASIS. OASIS currently employs a number of tests to detect such situations (see §4.4); we do not, however, want to base the accuracy of our system on the assumption that such ad-hoc tests will provide full coverage from erroneous results.

Yet because our approach gives this absolute error bound, we can perform simple constraint satisfaction checks on returned results, *i.e.,* do they obey physical constraints due to the speed-of-light propagation delay in fiber? The core only accepts a prefix-to-coordinate mapping after seeing two consistent measurements from replicas on different networks (to capture network peculiarities such as firewalls) and belonging to different services (to capture potential service-wide malicious behavior).

In hindsight, another significant benefit of geographic coordinates is the ability to couple them with real-time visualization of the network [130], which has helped us identify, debug, and subsequently handle these various network peculiarities.

4.2.2.3 System management and data replication

To achieve scalability and robustness, the location information of prefixes must be made available to all core nodes. We now describe OASIS's main system management and data organization techniques.

Gossiping. Designed as an *infrastructure service*, every OASIS core node maintains a global view of all other nodes in the core, using the scalable failure detection subsystem (described in §2.2.2) to keep this membership set weakly-consistent. Implicit in this design is the assumption that nodes are relatively stable; otherwise, the system would incur a high bandwidth cost for failure announcements.

Additionally, the gossiping mechanism used by this group membership layer is used to efficiently disseminate other system messages, including information about service policy updates and new prefix coordinates. The group membership messages, in fact, can just be piggybacked on top of these preexisting system messages (as they are meant to be broadcast through the network, and thus are forwarded to randomly-chosen neighbors, as required by the epidemic gossiping algorithm).

Consistent hashing. OASIS tasks must be assigned to nodes in some globally-known yet fully-decentralized manner. For example, to decide the responsibility of mapping specific IP prefixes,

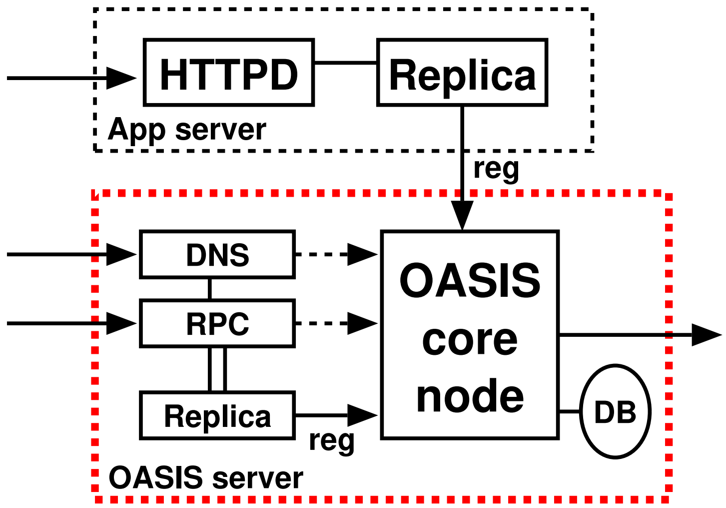


Figure 4.7: *OASIS system components.* Interfaces query core functionality to answer client anycast requests (dashed arrows from DNS and RPC). Services and interfaces register with the core to enable discovery.

we partition the set of prefixes over all nodes. Similarly, we assign specific nodes to play the role of a *service rendezvous* to aggregate information about a particular service (described in §4.2.3.3).

OASIS provides this assignment through its use of consistent hashing to build a distributed key-value index, as describe in §2.2. Each node has a random identifier; a key in the system—*e.g.,* the SHA-1 hash of the IP prefix or service name—is mapped to several nodes with the closest node identifiers. Finding these nodes is easy, of course, since all nodes have a global view. While nodes' views of the set of closest nodes are not guaranteed to be consistent, views can be easily reconciled using nodes' incarnation numbers. This replication of function is robust to node failures, while our use of a *forward* primitive that enables indirect routing (as described in §2.2.4) allows nodes to route around network failures to reach their desired target in the overlay.

We note that the design of OASIS is not at odds with a DHT-based key-value index. Rather, given the levels of scalability that we believe are required to provide OASIS's functionality, we eschewed such a design given the performance and simplicity of the above.

Soft-state replica registration. OASIS must know all replicas belonging to a service in order to answer corresponding anycast requests. To tolerate replica failures robustly, replica information is maintained using soft-state: replicas periodically send registration messages to core nodes (currently, every 60 seconds).

Hosts running services that use OASIS for anycast—such as the web server shown in Figure 4.7—run a separate replica process that connects to their local application (*i.e.,* the web server) every keepalive period (currently set to 15 seconds). The application responds with its current load and capacity. While the local application remains alive, the replica continues to refresh its locality, load, and capacity with its OASIS core node.[5]

Service discovery. OASIS's anycast functionality provides an elegant approach for service discovery and bootstrapping: OASIS can support multiple protocol interfaces—DNS, RPC, and HTTP currently—without requiring that every node support every interface. (Our current deployment on PlanetLab, for example, does not have access to UDP port 53 on every host on which to run its DNS server.) To enable clients to find a nearby or unloaded node running the desired interface, OASIS runs its various interfaces just as any other service, using a replica process to register its interfaces, as shown in Figure 4.7. Thus, to access a particular OASIS interface for subsequent requests, a client can simply issue a DNS request for the desired interface: `dns.nyuld.net`, `rpc.nyuld.net`, `http.nyuld.net`, and so forth.

Closest-node discovery. OASIS offloads all measurement costs to service replicas. All replicas, belonging to different services, form a lightweight overlay, in order to answer closest-replica queries from core nodes. Each replica organizes its neighbors into concentric rings of exponentially-increasing radii, as proposed by Meridian [191]: A replica accepts a neighbor for ring i only if its RTT is between 2^i and 2^{i+1} milliseconds. To find the closest replica to a destination d, a query operates in successive steps that "zero in" on the closest node in an expected $O(\log n)$ steps. At each step, a replica with RTT r from d chooses neighbors to probe d, restricting its selection to those with RTTs (to itself) between $\frac{1}{2}r$ and $\frac{3}{2}r$. The replica continues the search on its neighbor

[5] Alternatively, a service can adopt OASIS without modifying the local application by instead using scripts to check application liveness and load and to respond to keepalive queries, although this approach may have less application-level knowledge.

returning the minimum RTT to d. The search stops when the latest replica knows of no other potentially-closer nodes.

Our implementation differs from [191] in that we perform closest routing iteratively, as opposed to recursively: The first replica in a query initiates each progressive search step. This design trades overlay routing speed for greater robustness to packet loss.

4.2.3 Architecture

In this section, we describe the distributed architecture of OASIS in more detail: its distributed management and collection of data, locality and load optimizations, scalability, and security properties.

4.2.3.1 Managing information

We now describe how OASIS manages the four tables described in §4.2.2. OASIS optimizes response time by heavily replicating most information. Service, bucketing, and proximity information need only be weakly consistent; stale information only affects system performance, not its correctness. On the other hand, replica liveness information must be more fresh.

Service table. When a service initially registers with OASIS, it includes a service policy that specifies its service name and any domain name aliases, its desired server-selection algorithm, a public signature key, the maximum and minimum number of addresses to be included in client responses, and the TTLs of these responses (if applicable). Each core node maintains a local copy of the service table to be able to efficiently handle requests. When a new service joins OASIS or updates its existing policy, its policy is disseminated throughout the system by gossiping.

The server-selection algorithm specifies how to order replicas as a function of their distance, load, and total capacity when answering anycast requests. By default, OASIS ranks nodes by their coordinate distance to the target, favoring nodes with greater excess capacity to break ties. The optional signature key is used to authorize replicas registering with an OASIS core node as belonging to the service (see §4.2.3.5).

Bucketing table. An OASIS core node uses its bucketing table to map IP addresses to IP prefixes. We bootstrap the table using BGP feeds from RouteViews [160], which has approximately 200,000 prefixes. A PATRICIA trie [125] efficiently maps IP addresses to prefixes using longest-prefix matching.

When core nodes modify their bucketing table by splitting or merging prefixes [133], these changes are gossiped in order to keep nodes' tables weakly consistent. Again, stale information does not affect system correctness: prefix withdrawals are only used to reduce system state, while announcements are used only to identify more precise coordinates for a prefix.

Proximity table. When populating the proximity table, OASIS seeks to find accurate coordinates for every IP prefix, while preventing unnecessary reprobing.

OASIS maps an IP prefix to the coordinates of its closest replica. To discover the closest replica, a core node first selects an IP address from within the prefix and issues a probing request to one or more known replicas (or first queries a neighbor to discover some). Each selected replica traceroutes the requested IP to find the last routable IP address, performs closest-node discovery using the replica overlay (see §4.2.2.3), and, finally, returns the coordinates of the nearest replica and its RTT distance from the target IP. If the prefix's previously recorded coordinate has either expired or has a larger RTT from the prefix—and provided that the new prefix passes sanity checking (4.2.2.2)—the OASIS core node reassigns the prefix to these new coordinates and starts gossiping this information.

To prevent many nodes from probing the same IP prefix, the system assigns prefixes to nodes using consistent hashing. That is, several nodes closest to *hash(prefix)* are responsible for probing the prefix (three by default). All nodes go through their subset of assigned prefixes in random order, probing the prefix if its coordinates have not been updated within the last T_p seconds. T_p is a function of the coordinate's error, such that highly-accurate coordinates are probed at a slower rate (see §4.2.2.2).

Liveness table. For each registered service, OASIS maintains a liveness table of known replicas. Gossiping is not appropriate to maintain these liveness tables at each node: stale information could cause nodes to return addresses of failed replicas, while high replica churn would require excessive gossiping and hence bandwidth consumption.

Instead, OASIS aggregates liveness information about a particular service at a few *service rendezvous* nodes, which are selected by consistent hashing. When a replica joins or leaves the system, or undergoes a significant load change, the OASIS core node with which it has registered sends an update to one of the k nodes closest to *hash(service)*. For scalability, these rendezvous nodes only receive occasional state updates, not each soft-state refresh continually sent by replicas to their core nodes. Rendezvous nodes can dynamically adapt the parameter k based on load, which is then gossiped as part of the service's policy. By default, $k = 4$, which is also configured as the dynamic lower bound.

Rendezvous nodes regularly exchange liveness information with one another, to ensure that their liveness tables remain weakly consistent. If a rendezvous node detects that a core node fails (via OASIS's failure detection mechanism), it invalidates all replicas registered by that node. These replicas will subsequently re-register with a different core node and their information will be re-populated at the rendezvous nodes.

Compared to logically-decentralized systems such as DHTs [162, 177], this aggregation at rendezvous nodes allows OASIS to provide faster response (similar to one-hop lookups) and to support complex anycast queries (*e.g.,* as a function of both locality and load).

4.2.3.2 Putting it together: Resolving anycast

Given the architecture that we have presented, we now describe the steps involved when resolving an anycast request (see Figure 4.8). For simplicity, we limit our discussion to DNS redirection. When a client queries OASIS for a hostname—*e.g.,* `coralcdn.nyuld.net`—for the first time:

1. The client queries the DNS root servers, finding an OASIS nameserver I for `nyuld.net` to which it sends the request.

2. **Core lookup:** OASIS core node I finds other core nodes near the client that support the DNS interface by executing the following steps:

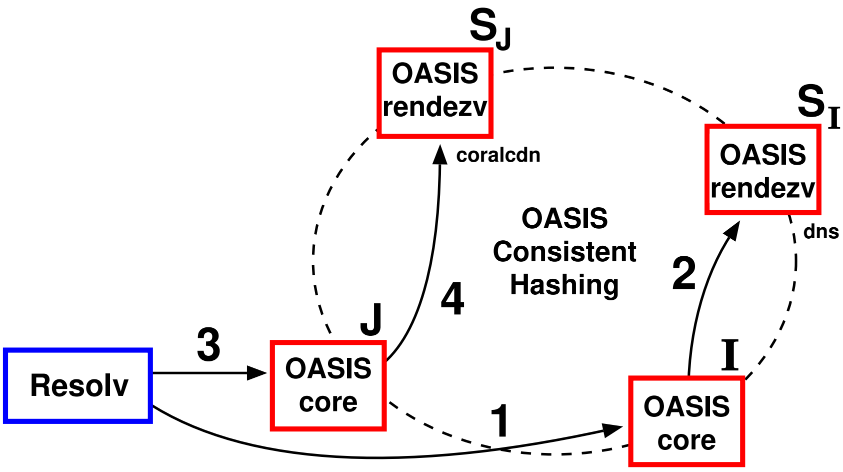


Figure 4.8: Steps involved in a DNS anycast request to OASIS using rendezvous nodes.

   (a) I locally maps the client's IP address to IP prefix, and then prefix to location coordinates.

   (b) I queries one of the k rendezvous nodes for service *dns*, call this node S_I, sending the client's coordinates.

   (c) S_I responds with the best-suited OASIS nameservers for the specified coordinates.

   (d) I returns this set of DNS replicas to the client. Let this set include node J.

3. The client resends the anycast request to J.

4. **Replica lookup:** Core node J finds replicas near the client using the following steps:

   (a) J extracts the request's service name and maps the client's IP address to coordinates.

   (b) J queries one of the k rendezvous nodes for service *coralcdn*, call this S_J.

   (c) S_J responds with the best *coralcdn* replicas, which J returns to the client.

Although DNS is a stateless protocol, we can force legacy clients to perform such two-stage lookups, as well as signal to their nameservers which stage they are currently executing. Section §4.2.4 gives implementation details.

4.2.3.3 Improving scalability and latency

While OASIS can support a large number of replicas by simply adding more nodes, the anycast protocol described in §4.2.3.2 has a bottleneck in scaling to large numbers of clients for a particular service: one of the k rendezvous nodes is involved in each request. We now describe how OASIS reduces these remote queries to improve both scalability and client latency.

Improving core lookups. OASIS first reduces load on rendezvous nodes by lowering the frequency of core lookups. For DNS-based requests, OASIS uses relatively-long TTLs for OASIS nameservers (currently 15 minutes) compared to those for third-party replicas (configurable per service, 60 seconds by default). These longer TTLs seem acceptable given that OASIS is an infrastructure service, and that resolvers can failover between nameservers since OASIS returns multiple, geo-diverse nameservers.

Second, we observe that core lookups are rarely issued to *random* nodes: Core lookups in DNS will initially go to one of the twelve primary nameservers registered for `.nyuld.net` in the main DNS hierarchy. So, we can arrange the OASIS core so that these 12 primary nameservers play the role of rendezvous nodes for *dns*, by simply having them choose $k = 12$ consecutive node identifiers for consistent hashing (in addition to their normal random identifiers). This configuration reduces latency by avoiding remote lookups.

Improving replica lookups. OASIS further reduces load by leveraging request locality. Since both clients and replicas are redirected to their nearest OASIS core nodes—when performing anycast requests and initiating registration, respectively—hosts redirected to the same core node are likely to be close to one another. Hence, on receiving a replica lookup, a core node first checks its local liveness table—which has cached liveness information about replicas directly registering with it—for any replica that satisfies the service request.

To improve the effectiveness of using local information, OASIS also uses *local flooding*: Each core node receiving registrations sends these local replica registrations to some of its closest neighbors. (“Closeness” is calculated using coordinate distance, not round-trip-time latency, to mirror the same selection criterion used for anycast.) Intuitively, this approach helps prevent situations in which replicas and clients select different collocated nodes and therefore lose the benefit of local information. We analyze the performance benefit of local flooding in §4.3.1.

OASIS implements other obvious strategies to reduce load, including having core nodes cache replica information returned by rendezvous nodes and batch replica updates to rendezvous nodes.

4.2.3.4 Selecting replicas based on load

While our discussion has mostly focused on locality-based replica selection, OASIS supports multiple selection algorithms incorporating factors such as load and capacity. However, in most practical cases, load-balancing need not be perfect; a reasonably good node is often acceptable. For example, to reduce costs associated with "95th-percentile billing," only the elimination of traffic spikes is critical. To eliminate such spikes, a service's replicas can track their 95% bandwidth usage over five-minute windows, then report their load to OASIS as the logarithm of this bandwidth usage. By specifying load-based selection in its policy, a service can ensure that its 95% bandwidth usage at its most-loaded replica is within a factor of two of its least-loaded replica; we have evaluated this policy in §4.3.2.

However, purely load-based metrics cannot be used in conjunction with many of the optimizations that reduce replica lookups to rendezvous nodes (§4.2.3.3), as locality does not play a role in such replica selection. On the other hand, the computation performed by rendezvous nodes when responding to such replica lookups is much lower: while answering locality-based lookups requires the rendezvous node to compute the closest replica(s) with respect to the client's location, answering load-based lookups requires the node simply to return the first element(s) of a single list of service replicas, sorted by increasing load. The ordering of this list needs to be recomputed only when replicas' loads change.

4.2.3.5 Security properties

OASIS has the following security requirements. First, it should prohibit unauthorized replicas from joining a registered service. Second, it should limit the extent to which a particular service's replicas can inject bad coordinates. Finally, it should prevent adversaries from using the infrastructure as a platform for DDoS attacks.

We assume that all OASIS core nodes are trusted; they do not gossip false bucketing, coordinates, or liveness information. We also assume that core nodes have loosely synchronized clocks to verify expiry times for replicas' authorization certificates. (Loosely-synchronized clocks are also required to compare registration expiry times in liveness tables, as well as measurement times when determining whether to reprobe prefixes.) Additionally, we assume that services joining OASIS have some secure method to initially register a public key. An infrastructure deployment of OASIS may have a single or small number of entities performing such admission control; the service provider(s) deploying OASIS's primary DNS nameservers are an obvious choice. Less secure schemes such as using DNS TXT records may also be appropriate in certain contexts.

To prevent unauthorized replicas from joining a service, a replica must present a valid, fresh certificate signed by the service's public key when initially registering with the system. This certificate includes the replica's IP address and its coordinates. By providing such admission control, OASIS only returns IP addresses that are authorized as valid replicas for a particular service.

OASIS limits the extent to which replicas can inject bad coordinates by evicting faulty replicas or their corresponding services. We believe that sanity-checking coordinates returned by the replicas—coupled with the penalty of eviction—is sufficient to deter services from assigning inaccurate coordinates for their replicas and replicas from responding falsely to closest-replica queries from OASIS.

Finally, OASIS prevents adversaries from using it as a platform for distributed denial-of-service attacks by requiring that replicas accept requests to perform closest-replica discovery only from core nodes. It also requires that a replica's overlay neighbors are authorized by OASIS (hence, replicas only accept probing requests from other approved replicas). OASIS itself has good resistance to DoS attacks, as most client requests can be resolved using information stored locally, *i.e.,* not requiring wide-area lookups between core nodes.

4.2.4 Implementation

OASIS's implementation consists of three main components: the OASIS core node, the service replica, and stand-alone interfaces (including DNS, HTTP, and RPC). All components are implemented in C++ and use the asynchronous I/O library from the SFS toolkit [118], structured using asynchronous events and callbacks. The core node comprises about 12,000 lines of code, the replica about 4,000 lines, and the various interfaces about 5,000 lines. The bucketing table is maintained using an in-memory PATRICIA trie [125], while the proximity table uses BerkeleyDB [171] for persistent storage.

OASIS's design uses static latitude/longitude coordinates with Meridian overlay probing [191]. For comparison purposes, OASIS also can be configured to use synthetic coordinates using Vivaldi [46] or GNP [129].

RPC and HTTP interfaces. OASIS's RPC and HTTP interfaces take additional inputs and expose extra functionality as compared to DNS. Figure 4.9 gives the queries currently supported by OASIS's HTTP interface (with function arguments passed as query string parameters). It can, for example, perform an anycast request with respect to any specified IP address, as opposed to that of the requesting client's, to support HTTP redirectors for third-party services (Figure 4.3). These interfaces also enable a localization service by simply exposing OASIS's proximity table, so that any client can ask "What are the coordinates of IP x?" In addition to HTML, the HTTP interface supports XML-formatted output for easy visualization using online mapping services (*e.g.,* Google Maps [71]).

DNS interface. OASIS takes advantage of low-level DNS details to implement anycast. First, a nameserver must differentiate between core and replica lookups. Core lookups only return *nameserver* (NS) records for nearby OASIS nameservers. Replica lookups, on the other hand, return *address* (A) records for nearby replicas. Since DNS is a stateless protocol, we signal the type of a client's request in its DNS query: replica lookups all have `oasis` prepended to `nyuld.net`. We force such signaling by returning CNAME records during core lookups, which map aliases to their *canonical names*.

Function	Purpose
clnt (ip)	Geolocate a particular IP address.
dist (srcip, dstip)	Geolocate two addresses and estimate their distance apart.
anycast (service, ?ip, ?lat, ?lng, ?max, ?min)	Return a set of IP addresses at which the specified service is running. If the optional latitude and longitude are supplied, servers nearby those coordinates are returned; otherwise, servers nearby to the optionally-supplied ip are returned (otherwise, the requesting client's address is used). The values min and max specify the desired number of addresses.
redir (service, ?ip, ?suffix)	Generate an HTTP redirect to a URL `http://ipaddr/suffix`, where the returned IP address belongs to the specified service (with respect to the optional ip, or to the requesting client if none is given).
rndz (service)	Return the set of core rendezvous nodes (see §4.2.3.1) responsible for the specified service.
pol (service)	Return all policy information about a service.
srvs (service)	Return a subset of known nodes belonging to a service.

Figure 4.9: Functions supported by OASIS's HTTP interface

```
;; QUESTION SECTION:
;example.net.nyud.net.              IN A

;; ANSWER SECTION:
example.net.nyud.net  600 IN CNAME  coralcdn.ab4040d9a9e53205.oasis.nyuld.net.

coralcdn.ab4040d9a9e53205.oasis.nyuld.net.  60 IN A  171.64.64.217

;; AUTHORITY SECTION:
ab4040d9a9e53205.oasis.nyuld.net. 600 IN NS 171.64.64.217.ip4.oasis.nyuld.net.
ab4040d9a9e53205.oasis.nyuld.net. 600 IN NS  169.229.50.5.ip4.oasis.nyuld.net.
```

Figure 4.10: Output of `dig` for a hostname using OASIS

This technique alone is insufficient to force many client resolvers, including BIND, to immediately issue replica lookups to these nearby nameservers. We illustrate this with an example query for CoralCDN. A resolver R discovers nameservers u, v for `nyud.net` by querying the root servers for `example.net.nyud.net`.[6] Next, R queries u for this hostname. Consider if R is returned a CNAME for the mapping

$$\texttt{example.net.nyud.net} \rightarrow \texttt{coralcdn.oasis.nyuld.net}$$

along with NS records x, y for `coralcdn.oasis.nyuld.net`. In practice, R will then reissue a new query for `coralcdn.oasis.nyuld.net` to nameserver v, which is not guaranteed to be close to R (and v's local cache may include replicas far from R).

Instead, we again use the DNS query string to signal whether a client is contacting the correct nameservers. When responding to core lookups, we encode the set of NS records in hex format (`ab4040d9a9e53205`) in the returned CNAME record (Figure 4.10). Thus, if v receives a replica lookup, it checks whether the query encodes its own IP address, and if it does not, immediately re-returns NS records for x, y (extracted from the hex-encoding). Now, having received NS records authoritative for the name queried, a resolver contacts the desired nameservers x or y, which returns an appropriate replica for `coralcdn`.

4.3 Evaluation

This section evaluates OASIS's performance benefits for DNS-based anycast, as well as its scalability and bandwidth trade-offs.

[6] To adopt OASIS yet preserve its own top-level domain name, CoralCDN points the NS records for `nyud.net` to OASIS's nameservers; `nyud.net` is an alias for `coralcdn` in its service policy registered with OASIS.

4.3.1 Wide-area evaluation of OASIS

Experimental setup. We present wide-area measurements on PlanetLab [145] that evaluate the accuracy of replica selection based on round-trip-time and throughput, DNS response time, and the end-to-end time for a simulated web session. In all experiments, we ran replicas for one service on approximately 250 PlanetLab hosts spread around the world (including 22 in Asia), and we ran core nodes and DNS servers on 37 hosts.[7]

We compare the performance of replica selection using six different anycast strategies: (1) *OASIS (LF)* refers to the OASIS system, using both local caching and local flooding (to the nearest three neighbors; see §4.2.3.3). (2) *OASIS* uses only local caching for replicas. (3) *Meridian* (our implementation of [191]) performs on-demand probing by executing closest-replica discovery whenever it receives a request. (4) *Vivaldi* uses 2-dimensional dynamic virtual coordinates [46], instead of static geographic coordinates, by probing the client from 8-12 replicas on-demand. The core node subsequently computes the client's virtual coordinates and selects its closest replica based on virtual coordinate distance. (5) *Vivaldi (cached)* probes IP prefixes in the background, instead of on-demand. Thus, it is similar to OASIS with local caching, except uses virtual coordinates to populate OASIS's proximity table. (6) Finally, *RRobin* performs round-robin DNS redirection amongst all replicas in the system, using a single DNS server located at Stanford University.

We performed client measurements on the same hosts running replicas. However, we configured OASIS so that when a replica registers with an OASIS core node, the node does *not* directly save a mapping from the replica's prefix to its coordinates, as OASIS would do normally (as this would give OASIS an unfair benefit in our evaluation, given the collocation of our test clients). Instead, we rely purely on OASIS's background probing to assign coordinates to the replica's prefix.

Three consecutive experiments were run at each site when evaluating ping, DNS, and end-to-end latencies. Short DNS TTLs were chosen to ensure that clients contacted OASIS for each request. Data from all three experiments are included in computing the following cumulative distribution function (CDF) graphs.

[7]This number was due to the unavailability of UDP port 53 on most PlanetLab hosts, especially given CoralCDN's concurrent use of same.

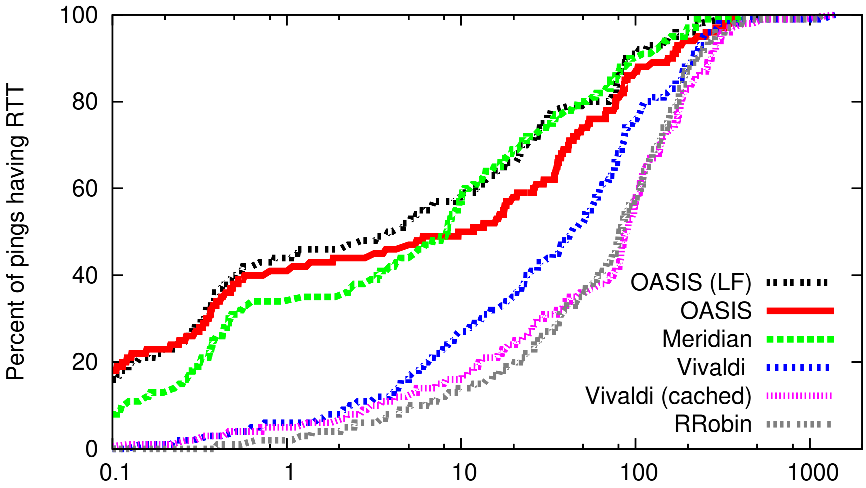


Figure 4.11: Round-trip-times (ms) between clients and servers selected by OASIS

Minimizing RTTs. Figures 4.11 shows the CDFs of round-trip-times in log-scale between clients and their returned replicas. We measured RTTs via ICMP echo messages, using the ICMP response's kernel timestamp when calculating RTTs. RTTs as reported are the minimum of ten consecutive probes. We see that OASIS and Meridian significantly outperform anycast using Vivaldi and round robin by one to two orders of magnitude.

Two other interesting results merit mention. First, *Vivaldi (cached)* performs significantly worse than on-demand *Vivaldi* and even often worse than *RRobin*. This arises from the fact that *Vivaldi* is not stable over time with respect to coordinate translation and rotation. Hence, cached results quickly become inaccurate, although some recent work has sought to minimize this instability [49, 144]. Second, OASIS outperforms *Meridian* for 60% of measurements, a rather surprising result given that OASIS *uses* Meridian as its background probing mechanism. It is here where we see OASIS's benefit from using RTT as an absolute error predictor for coordinates (§4.2.2.2): reprobing by OASIS yields strictly better results, while the accuracy of Meridian queries can vary.

Maximizing throughput. Figure 4.12 shows the CDFs of the steady-state throughput from replicas to their clients, to examine the benefit of using nearby servers to improve data-transfer rates. TCP throughput is measured using `iperf-1.7.0` [88] in its default configuration (a TCP

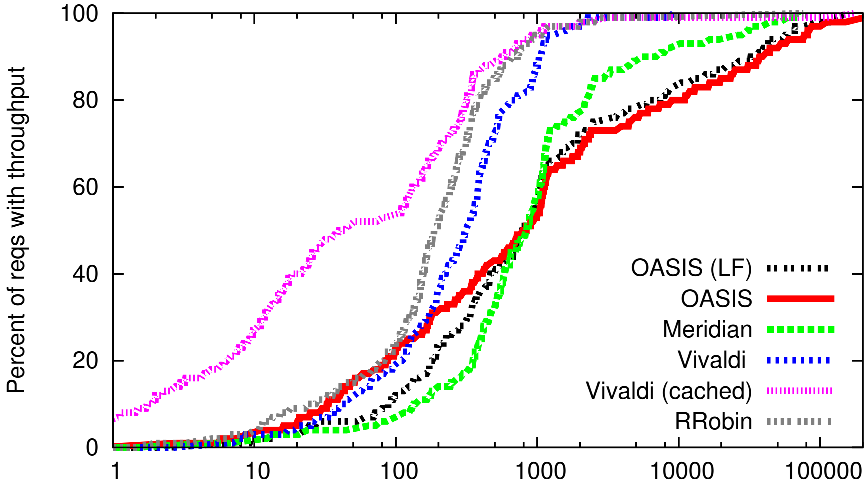
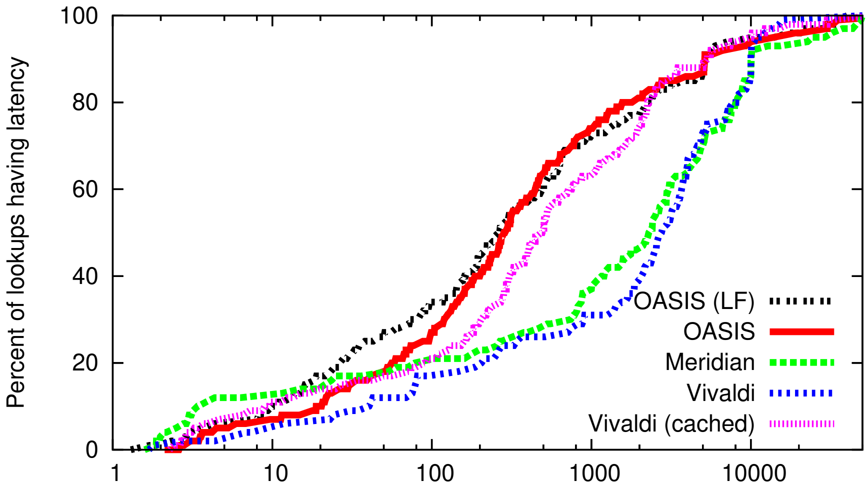


Figure 4.12: TCP throughput (KB/s) between clients and servers selected by OASIS

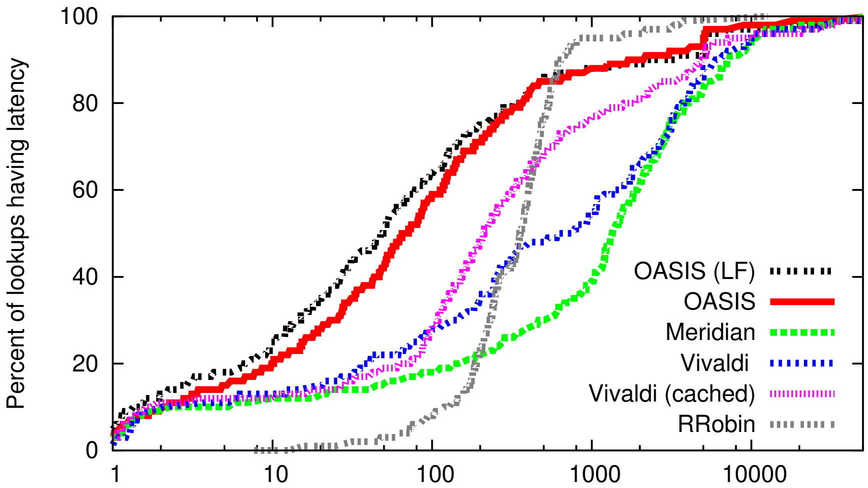
window size of 32 KB). The graph shows TCP performance in steady-state. OASIS is competitive with or superior to all other tested systems, demonstrating its benefit for large data transfers.

DNS resolution time. Figure 4.13 evaluates the DNS performance for new clients and for clients already caching their nearby OASIS nameservers. A request by a new client includes the time to perform three steps: (1) contact an initial OASIS core node to learn a nearby nameserver, (2) re-contact a distant node and again receive NS records for the same nearby nameservers (see §4.2.4), and (3) contact a nearby core node as part of a replica lookup. Note that we did not specially configure the 12 primary nameservers as rendezvous nodes for *dns* (see §4.2.3.3), and thus use a wide-area lookup during Step 1. This two-step approach is taken by all systems: *Meridian* and *Vivaldi* both perform on-demand probing twice. We omit *RRobin* from this experiment, however, as it always uses a single nameserver. Clients already caching nameserver information need only perform Step 3, as given in the bottom figure.

OASIS's strategy of first finding nearby nameservers and then using locally-cached information can achieve significantly faster DNS response times compared to on-demand probing systems. The median DNS resolution time for OASIS replica lookups is almost 30-times faster than that for



(a) Resolution time (ms) for new clients



(b) Resolution time (ms) for replica lookups

Figure 4.13: DNS resolution time (ms) using OASIS

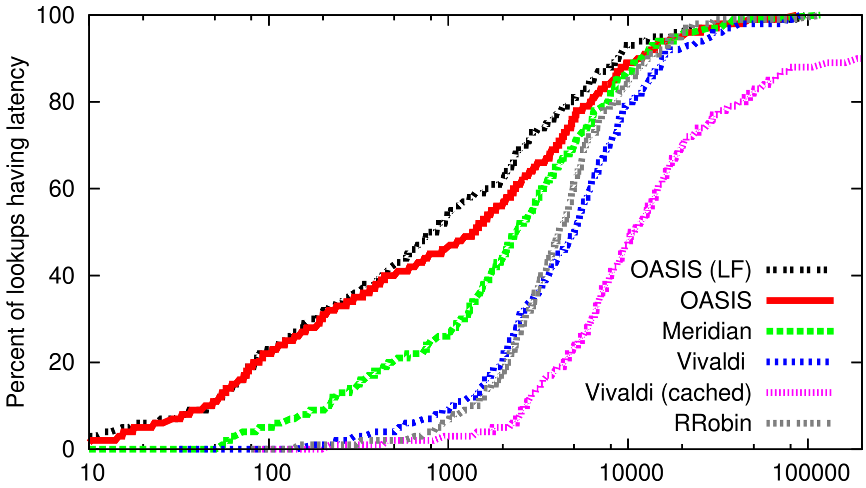


Figure 4.14: End-to-end download performance (ms) experienced by clients when interacting with OASIS-selected servers

Meridian.[8] We also see that local flooding can improve median performance by 40% by reducing the number of wide-area requests to rendezvous nodes.

End-to-end latency. Figure 4.14 shows the end-to-end time for a client to perform a synthetic web session, which includes first issuing a replica lookup via DNS and then downloading eight 10 KB files sequentially. This file size is chosen to mimic that of common image files, which are often embedded multiple times on a given web page. We do not simulate persistent connections for our transfers, so each request establishes a new TCP connection before downloading the file. Also, our faux-webserver never touches the disk, so does not take (PlanetLab's high) disk-scheduling latency into account.

End-to-end measurements underscore OASIS's true performance benefit, coupling fast DNS response time with accurate server selection. Median response time for OASIS is 290% faster than *Meridian* and 500% faster than simple round-robin systems.

[8]A recursive Meridian implementation [191] may be faster than our iterative implementation: our design emphasizes greater robustness to packet loss, given our preference for minimizing probing.

metric	California	Texas	New York	Germany
latency	23.3	0.0	0.0	0.0
load	9.0	11.3	9.6	9.2

Figure 4.15: 95th-percentile bandwidth usage (MB) at servers selected by OASIS

4.3.2 Load-based replica selection

This section considers replica selection based on load. We do not seek to quantify an optimal load- and latency-aware selection metric; rather, we verify OASIS's ability to perform load-aware anycast. Specifically, we evaluate a load-balancing strategy meant to reduce costs associated with 95th-percentile billing (§4.2.3.4).

In this experiment, we use four distributed servers that run our faux-webserver. Each webserver tracks its bandwidth usage per minute, and registers its load with its local replica as the logarithm of its 95th-percentile usage. Eight clients, all located in California, each make 50 anycast requests for a 1 MB file, with a 20 second delay between requests. (Policy specifies that DNS records are returned with a TTL of 15 seconds.)

Table 4.15 shows that the bandwidth utilization at the webserver with highest bandwidth costs is easily within a factor of two of the least-loaded server. On the other hand, pure locality-based replica selection creates a traffic spike at a single webserver.

4.3.3 Scalability

Since OASIS is designed as an infrastructure system, we now verify that a reasonable-sized OASIS core can handle Internet-scale usage.

Measurements at DNS root servers have shown steady traffic rates of around 6.5M A queries per 10 minute interval across all $\{e, i, k, m\}$.*root-servers.net* [99]. With a deployment of 1000 OASIS DNS servers—and, for simplicity, assuming an even distribution of requests to nodes—even if OASIS received requests at an equivalent rate, each node would see only about 10 requests per second.

On the other hand, OASIS often uses shorter TTLs to handle replica failover and load bal-

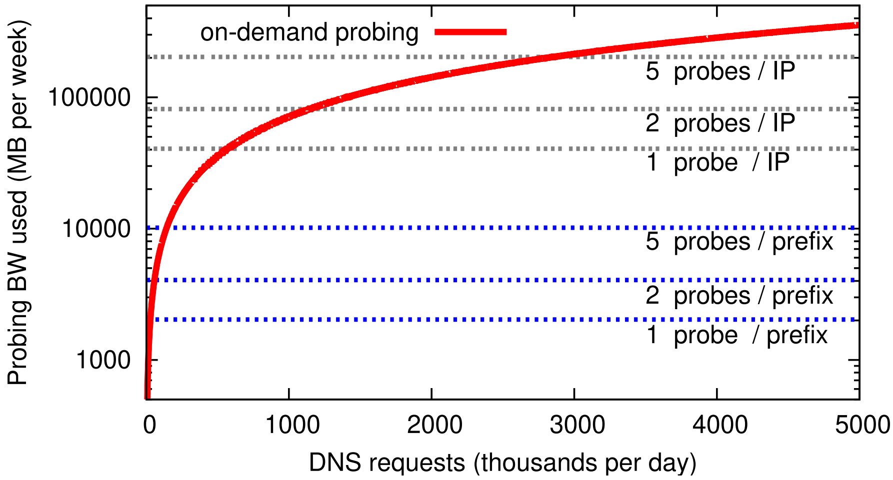


Figure 4.16: Bandwidth trade-off between on-demand probing, caching IP prefixes (OASIS), and caching IP addresses

ancing. The same datasets showed approximately 100K unique resolvers per 10 minute interval. Using the default TTL of 60 seconds, even if every client re-issued a request every 60 seconds for all s services using OASIS, each core node would receive at most $(1.6 \cdot s)$ requests per second.

To consider one real-world service, as opposed to some upper bound for all Internet traffic, CoralCDN's DNS service (in December 2005) answered slightly fewer than 5 million DNS queries (for all query types) per day, using TTLs of 30-60 seconds. This translates to a system *total* of 57 DNS queries per second.

4.3.4 Bandwidth trade-offs

This section examines the bandwidth trade-off between precomputing prefix locality and performing on-demand probing. If a system receives only a few hundred requests per week, for example, OASIS's approach of probing every IP prefix would not be worthwhile. Figure 4.16 plots the amount of bandwidth used in caching and on-demand anycast systems for a system with 2000 replicas. Following the results of [191], we estimate each closest-replica query to generate about 10.4 KB of network traffic (load grows sub-linearly with the number of replicas).

Project	Service	Description
ChunkCast [40]	chunkcast	Anycast gateways for large-file distribution
CoralCDN [59]	coralcdn	Web proxies
DONA [100]	dona	Data-oriented network architecture
Galaxy [193]	galaxy	Distributed file-system clients
Na Kika [73]	nakika	Web proxies
OASIS	dns	DNS interface
	http	HTTP interface
	rpc	RPC interface
OCALA [92]	ocala	Client IP gateways for overlay routing
	ocalarsp	Server IP gateways for overlay routing
OpenDHT [157]	opendht	Client DHT gateways
OverCite [179]	overcite	Distributed CiteSeer proxies
SlotNet [163]	slot	Overlay transport proxies

Figure 4.17: Services using OASIS as of May 2007. Services can be accessed using the domain name `<service>.nyuld.net`.

Figure 4.16 simulates the amount of bandwidth used per week for up to 5 million DNS requests per day (the request rate from CoralCDN), where each results in a new closest-replica query. OASIS's probing of 200K prefixes—even when each prefix may be probed multiple times—generates orders of magnitude less network traffic. We also plot an upper-bound on the amount of traffic generated if the system were to cache the IP addresses of observed DNS resolvers, as opposed to using IP prefixes.

While one might expect the number of DNS resolvers to be constant and relatively small, many resolvers use dynamically-assigned addresses and thus preclude a small working set: the root-servers saw more than 4 million unique clients in a week, with the number of clients increasing linearly after the first day's window [99]. Figure 4.16 uses this upper-bound to plot the amount of traffic needed when caching IP addresses. Of course, new IP addresses always need to be probed on-demand, with the corresponding performance hit (per Figure 4.13).

4.4 Deployment lessons

OASIS has been deployed on about 250 PlanetLab hosts since November 2005. Figure 4.17 lists the systems currently using OASIS and a brief description of their service replicas. We now present some lessons that we learned in the process.

Make it easy to integrate. Though each application server requires a local replica, for a shared testbed such as PlanetLab, a single replica process on a host can serve on behalf of multiple local processes running different applications. To facilitate this, we now run OASIS replicas as a public service on PlanetLab; to adopt OASIS, PlanetLab applications need only listen on a registered port and respond to keepalive messages.

Applications can integrate OASIS even without any source-code changes or recompilation. Operators can run or modify simple stand-alone scripts we provide that answer replica keepalive requests after simple liveness and load checks (via `ps` and the `/proc` filesystem).

Check for proximity discrepancies. Firewalls and middleboxes can lead one to draw false conclusions from measurement results. Consider the following two problems we encountered, mentioned earlier in §4.2.2.2.

To determine a routable IP address in a prefix, a replica performs a traceroute and uses the last reachable node that responded to the traceroute. However, since firewalls can perform egress filtering on ICMP packets, an unsuspecting node would then ask others to probe its own egress point, which may be far away from the desired prefix. Hence, replicas initially find their immediate upstream routers—*i.e.,* , the set common to multiple traceroutes—which they subsequently ignore.

When replicas probe destinations on TCP port 80 for closest-replica discovery, any local transparent web proxy will perform full TCP termination, leading an unsuspecting node to conclude that it is very close to the destination. Hence, a replica first checks for a transparent proxy, then tries alternative probing techniques.

Both problems would lead replicas to report themselves as incorrectly close to some IP prefix. So, by employing measurement redundancy, OASIS can compare answers for precision and sanity.

Be careful *what* you probe. No single probing technique is both sufficiently powerful and innocuous (from the point-of-view of intrusion-detection systems). As such, OASIS has adapted its probing strategies based on ISP feedback. ICMP probes and TCP probes to random high ports were often dropped by egress firewalls and, for the latter, flagged as unwanted port scans. Probing to TCP port 80 faced the problem of transparent web proxies, and probes to TCP port 22 were often flagged as SSH login attacks. Unfortunately, as OASIS performs probing from multiple networks, automated abuse complaints from IDSs are sent to many separate network operators. Currently, OASIS uses a mix of TCP port 80 probes, ICMP probes, and reverse DNS name queries.

Be careful *whom* you probe. IDSs deployed on some networks are incompatible with active probing, irrespective of the frequency of probes. Thus, OASIS maintains and checks a blacklist whenever a target IP prefix or address is selected for probing. We apply this blacklist at all stages of probing: Initially, only the OASIS core checked target IP prefixes. However, this strategy led to abuse complaints from autonomous systems that provide transit for the target, yet filter ICMPs; in such cases, replicas tracerouting the prefix would end up probing the upstream autonomous system, which would otherwise not be caught by the OASIS core.

4.5 Related work on anycast

We classify related work into two areas most relevant to OASIS: network distance estimation and server selection. Network distance estimation techniques are used to identify the location and/or distance between hosts in the network. The server-selection literature deals with finding an appropriately-located server (possibly using network distance estimation) for a client request.

Network distance estimation. Several techniques have been proposed to reduce the amount of probing per request. Some initial proposals (such as [81]) are based on the triangle-inequality assumption. IDMaps [57] proposed deploying *tracers* that all probe one another; the distance between two hosts is calculated as the sum of the distances between the hosts and their selected tracers, and between the two selected tracers. Theilmann and Rothermel described a hierarchical tree-like system [181], and Iso-bar proposed a two-tier system using landmark-based clustering algorithms [37]. King [75] used recursive queries to remote DNS nameservers to measure the RTT distance between any two non-participating hosts.

Recently, virtual coordinate systems (such as GNP [129] and Vivaldi [46]) offer new methods for latency estimation. Here, nodes generate synthetic coordinates after probing one another. The distance between peers in the coordinate space is used to predict their RTT, the accuracy of which depends on how effectively the Internet can be embedded into a d-dimensional (usually Euclidean) space.

Another direction for network estimation has been the use of geographic mapping techniques; the main idea is that if geographic distance is a good indicator of network distance, then estimating geographic location accurately would obtain a first approximation for the network distance between hosts. Most approaches in geographic mapping are heuristic. The most common approaches include performing queries against a `whois` database to extract city information [87, 139], or tracerouting the address space and then mapping router names to locations based on ISP-specific naming conventions [61, 133]. Commercial entities have sought to create exhaustive IP-range mappings [2, 147].

Server selection. IP anycast was proposed as a network-level solution to server selection [98, 137]. However, with various deployment and scalability problems, IP anycast is not widely used or available. Recently, PIAS has argued for supporting IP anycast as a proxy-based service to overcome deployment challenges [10]; OASIS can serve as a powerful and flexible server-selection backend for such a system.

One of the largest deployed content distribution networks, Akamai [2] reportedly traceroutes the IP address space from multiple vantage points to detect route convergence, then pings the common router from every data center hosting an Akamai cluster [37]. OASIS's task is more difficult than that of commercial CDNs: Akamai need only map the network relative to its clusters, while OASIS needs to find an intermediate representation of locality, given its goal of providing anycast for multiple services.

Recent literature has proposed techniques to minimize such exhaustive probing. Meridian [191] (used for DNS redirection by [190]) creates an overlay network with neighbors chosen from a particular distribution; routing to closer nodes is guaranteed to find a minimum given a growth-restricted metric space [95]. In contrast, OASIS completely eliminates on-demand probing.

OASIS allows more flexible server selection than pure locality-based solutions, as it stores load and capacity estimates from replicas in addition to locality information.

Chapter 5

Securing a Cooperative File System

The systems that we have described so far—CoralCDN and OASIS—have been characterized by backwards-compatible designs. CoralCDN went to great efforts to interoperate with unmodified web clients and servers, while OASIS made practical trade-offs given the lack of locality support and information from the network. Further, while both of these systems were designed for scalability and decentralized control, they were ultimately deployed on trusted servers due to security concerns. Malicious OASIS core servers could return incorrect, malicious replicas to anycast queries, akin to DNS hijacking attacks. Malicious CoralCDN proxies could otherwise modify content in transmission, and unmodified clients would not be able to detect the changes. Some of these problems are not technically difficult to solve, *e.g.,* a web client might only accept content if it is cryptographically signed by an origin webserver, but no such functionality universally exists in today's web browsers.

Rather than continuing to try to bolt on functionality to existing systems, we ask what we would do differently if not constrained by the goal of backwards-compatibility. The next two chapters focus on the clean-slate design of two systems for different classes of content distribution: distributed file systems and large-file distribution systems. We focus both on more advanced methods of content transmission directly between peers, as well as new security mechanisms for these protocols. Such security mechanisms serve as contrast to the limitations we encountered earlier, but also show that it is possible to build secure, robust systems through cryptographic protocols, even though participants may be malicious.

We first consider the case for Shark [8], a secure cooperative file system. Shark leverages participants' file caches in order to scale to large numbers of client read requests.

5.1 Motivation

Users of distributed computing environments often launch similar processes on hundreds of machines nearly simultaneously. Running jobs in such an environment can be significantly more complicated, both because of data-staging concerns and the increased difficulty of debugging. Batch-oriented tools, such as Condor [52], can provide I/O transparency to help distribute CPU-intensive applications. However, these tools are ill-suited to tasks like distributed web hosting and network measurement, in which software needs low-level control of network functions and resource allocation. An alternative is frequently seen on network testbeds such as RON [4] and PlanetLab [145]: users replicate their programs, along with some minimal execution environment, on every machine before launching a distributed application.

Replicating execution environments has a number of drawbacks. First, such replication wastes resources, particularly bandwidth. Popular file synchronization tools do not optimize for network locality, and they can push many copies of the same file across slow network links. Moreover, in a shared environment, multiple users will inevitably copy the exact same files, such as popular operating system add-on packages with language interpreters or shared libraries. Second, replicating run-time environments requires hard state, a scarce resource in a shared testbed. Programs need sufficient disk space, yet idle environments continue to consume disk space, in part because owners are loathe to consume the bandwidth and effort required for redistribution. Third, replicated run-time environments differ significantly from an application's development environment, in part to conserve bandwidth and disk space. For instance, users usually distribute only stripped binaries, not source or development tools, making it difficult to debug running processes in a distributed system.

Shark is a network file system specifically designed to support widely distributed applications. Rather than manually replicate program files, users can place a distributed application and its entire run-time environment in an exported file system, and simply execute the program directly from the file system on all nodes. In a chrooted environment such as PlanetLab, users can even make `/usr/local` a symbolic link to a Shark file system, thereby trivially making all local software available on all testbed machines.

Of course, the big challenge faced by Shark is scalability. With a normal network file system, if hundreds of clients suddenly execute a large, 40MB C++ program from a file server, the server quickly saturates its network uplink and delivers unacceptable performance. Shark, however, scales to large numbers of clients through a locality-aware cooperative cache. When reading an uncached file, a Shark client avoids transferring the file or even chunks of the file from the server, if the same data can be fetched from another, preferably nearby, client. For world-readable files, clients will even download nearby cached copies of identical files—or even file chunks—originating from different servers.

Shark leverages the Coral indexing layer (§2.3) to coordinate client caching. Shark clients form self-organizing clusters of well-connected machines. When multiple clients attempt to read identical data, these clients locate nearby replicas and stripe downloads from each other in parallel. Thus, even modestly-provisioned file servers can scale to hundreds, possibly thousands, of clients making mostly read accesses.

There have been serverless, peer-to-peer file systems capable of scaling to large numbers of clients, notably Ivy [127]. Unfortunately, these systems have highly non-standard models for administration, accountability, and consistency. For example, Ivy spreads hard state over multiple machines, chosen based on file-system data structure hashes. This leaves no single entity ultimately responsible for the persistence of a given file. Moreover, peer-to-peer file systems are typically noticeably slower than conventional network file systems. Thus, they do not provide a substitute for conventional file systems in either accountability or performance. Shark, by contrast, exports a traditional file-system interface, is compatible with existing backup and restore procedures, provides competitive performance on the local area network, and yet easily scales to many clients in the wide area.

For workloads with no read sharing between users, Shark offers performance that is competitive with traditional network file systems. However, for shared read-heavy workloads in the wide area, Shark greatly reduces server load and improves client latency. Compared to both NFSv3 [23] and SFS [119], a secure network file system, Shark can reduce server bandwidth usage by nearly an order of magnitude and can provide a 4x-6x improvement in client latency for reading large files, as shown by both local-area experiments on the Emulab [188] testbed and wide-area experiments on PlanetLab [145].

By providing scalability, efficiency, and security, Shark enables network file systems to be employed in environments where they were previously impractical. Yet Shark retains their attractive API, semantics, and portability: Shark interacts with the local host using an existing network file system protocol (NFSv3) and runs in user space.

5.2 Design

Shark's design incorporates a number of key ideas aimed at reducing the load on the server and improving client-perceived latencies. Shark enables clients to securely mount remote file systems and efficiently access them, by providing mechanisms by which mutually distrustful peers can read content from another, while still preserving traditional integrity and authorization guarantees.

When a client is the first to read a particular file, it fetches data from the file server. Upon retrieving *chunks* of the file, the client caches them and registers itself as a *replica proxy* (or *proxy* for short) for these chunks in the distributed index. (Notice that while CoralCDN used the Coral index to map entire files to proxies, Shark takes a finer-grained approach in order to better exploit parallelism when downloading large files.) Subsequently, when another client attempts to access the file, it discovers proxies for the file chunks by querying the distributed index. The client then establishes a secure channel to multiple such (untrusted) proxies and downloads the file chunks

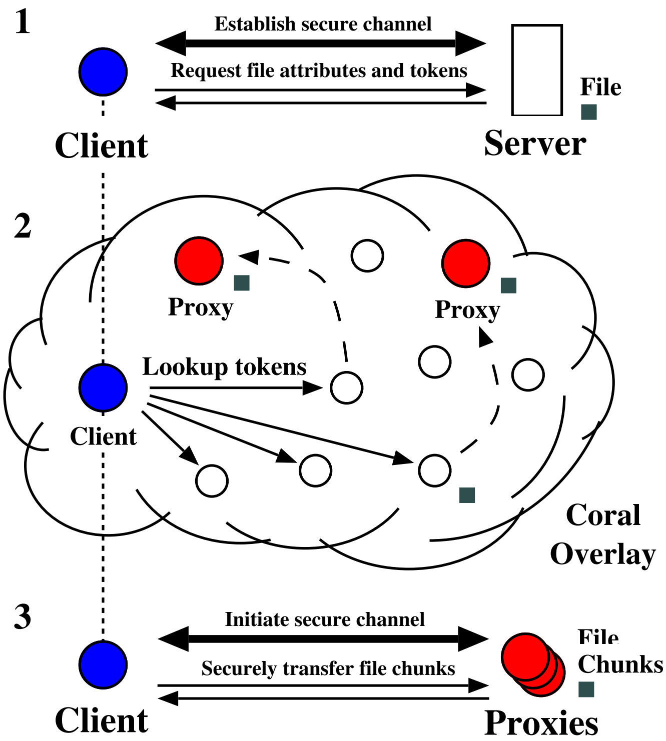


Figure 5.1: *Shark System Overview.* A client machine simultaneously acts as a client (to handle local application file system accesses), as a proxy (to serve cached data to other clients), and as a node (within the distributed index).

in parallel. Upon successfully fetching these chunks, the client registers itself also as a proxy for these chunks.

5.2.1 System overview

Figure 5.1 provides an overview of the Shark system. In a real deployment, there may be multiple file servers that each host separate file systems, and each client may access multiple file systems. For simplicity, however, we show a single file server.

When a client attempts to read a file, it queries the file server for the file's attributes and some opaque tokens (Step 1 as shown). One token identifies the contents of the whole file, while other tokens each identify a particular *chunk* of the file. A Shark server divides a file into chunks by running a Rabin fingerprint algorithm on the file. This technique (used previously by LBFS [126]) splits a file along specially chosen boundaries in such a way that preserves data commonalities across files. Such commonalities may be found, for example, between file versions or when concatenating files, such as building program libraries from object files.

Next, a client attempts to discover replica proxies for the particular file via Shark's distributed index (Step 2). Shark clients organize themselves into a Coral key-value indexing layer: A client executes put to declare that it has something; get returns the list of nearby clients who have something. A Shark client uses its tokens to derive *indexing keys* that serve as inputs to these operations. It uses this distributed index to register itself and to find other nearby proxies caching a file chunk.

Finally, a client connects to several of these proxies, and it requests various chunks of data from each proxy in parallel (Step 3). Note, however, that clients themselves are mutually distrustful, so Shark must provide various mechanisms to guarantee secure data sharing: (1) Data should be encrypted to preserve confidentiality and should be decrypted only by those with appropriate read permissions. (2) A malicious proxy should not be able to break data integrity by modifying content without a client detecting the change. (3) A client should not be able to download large amounts of even encrypted data without proper read authorization.

Shark uses the opaque tokens generated by the file server in several ways to handle these security issues. (1) The tokens serve as a shared secret (between client and proxy) with which to derive symmetric cryptographic keys for transmitting data from proxy to client. (2) The client can verify the integrity of retrieved data, as the token acts to bind the file contents to a specific verifiable value. (3) A client can "prove" knowledge of the token to a proxy and thus establish read permissions for the file. Note that the indexing keys used as input to the distributed index are only derived from the token; they do not in fact expose the token's value or otherwise destroy its usefulness as a shared secret.

Shark allows clients to share common data segments on a sub-file granularity. As a file server provides the tokens naming individual file chunks, clients can share data at the granularity of chunks as opposed to whole files.

In fact, Shark provides *cross-file-system sharing* when tokens are derived solely from file contents. Consider the case when users attempt to mount `/usr/local` (for the same operating system) using different file servers. Most of the files in these directories are identical and even when the file versions are different, many of the chunks are identical. Thus, even when distinct subsets of clients access different file servers to retrieve tokens, one can still act as a proxy for the other to transmit the data.

Given the benefit of cross-file-system sharing, all Shark clients organize themselves into a *global* Coral indexing layer, even though non-overlapping subsets of these clients may be directly interacting only with independent Shark file servers. If such sharing is not desired or required—it has some privacy implications, which we discuss in §5.2.3.1—then global scalability in the indexing layer is not necessary. (A single Shark server is practically limited to a few thousand clients, after all, due to the underlying file-system consistency protocols it uses, described next.) In those cases, therefore, one could deploy Shark using a much simpler and faster indexing layer based on global membership views (as we did with OASIS in §4 using techniques from §2.2). Our implementation and evaluation, however, uses Coral as the more scalable design choice.

5.2.2 File servers and consistency

Shark names file systems using self-certifying pathnames, as in SFS [119]. These pathnames *explicitly* specify all information necessary to securely communicate with remote servers. Every Shark file system is accessible under a pathname of the form:

$$/\texttt{shark}/@server, pubkey$$

A Shark client provides access to a remote file system by automounting requested directories [119]. This allows a client-side Shark NFS loop-back server to provide unmodified applications with seamless access to remote Shark file systems. Unlike NFS, however, all communication with the file server is sent over a secure channel, as the self-certifying pathname includes sufficient information to establish a secure channel.

Shark uses two network-file-system techniques to improve read performance and decrease

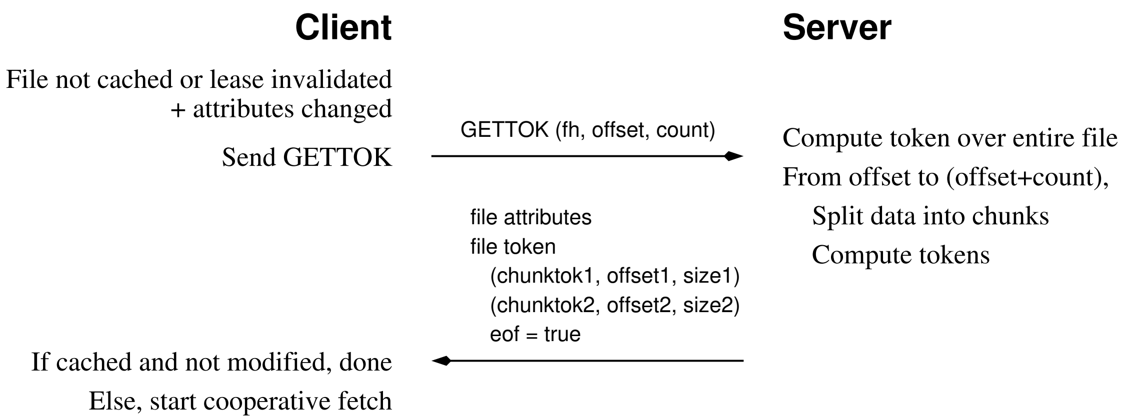


Figure 5.2: Shark GETTOK RPC

server load: leases [72] and AFS-style whole-file caching [85]. When a user attempts to read any portion of a file, the client first checks its disk cache.

If the file is not already cached, the client fetches a new version from Shark (either from the cooperative cache or directly from the file server). If its read lease on its cached copy is not up-to-date or the client has received a server callback, the client contacts the file server for fresh file attributes, determining thereafter whether its cached copy is fresh or whether the client needs to fetch a new version from Shark.

While Shark attempts to handle reads within its cooperative cache, all writes are sent to the origin server. When any type of modification occurs, the server must invalidate all unexpired leases, update file attributes, recompute its file token, and update its chunk tokens and boundaries.

5.2.3 Cooperative caching

File reads in Shark make use of one RPC procedure not in the NFS protocol, GETTOK, as shown in Figure 5.2.

GETTOK supplies a file handle, offset, and count as arguments, just as in a READ RPC. However, instead of returning the actual file data, it returns the file's attributes, the *file token*, and a vector of *chunk descriptions*. Each chunk description identifies a specific extent of the file by

Symbol	Description	Generated by ...	Only known by ...
F	File		Server and approved readers
F_i	ith file chunk	Chunking algorithm	Parties with access to F
r	Per-server randomness	$r = \mathsf{PRNG}()$ or $r = 0$	Parties with access to F
T	File/chunk token	$tok(F) = \mathsf{HMAC}_r(F)$	Parties with access to F/F_i
$\mathsf{I}, \mathsf{E}, \mathsf{A}_C, \mathsf{A}_P$	Special constants	System-wide parameters	Public
I	Indexing key	$\mathsf{HMAC}_T(\mathsf{I})$	Public
r_C, r_P	Session nonces	$r_C, r_P = \mathsf{PRNG}()$	Parties exchanging F/F_i
$Auth_C$	Client auth token	$\mathsf{HMAC}_T(\mathsf{A}_C, C, P, r_C, r_P)$	Parties exchanging F/F_i
$Auth_P$	Proxy auth token	$\mathsf{HMAC}_T(\mathsf{A}_P, P, P, r_P, r_C)$	Parties exchanging F/F_i
K_E	Encryption key	$\mathsf{HMAC}_T(\mathsf{E}, C, P, r_C, r_P)$	Parties exchanging F/F_i

Figure 5.3: Notation used for Shark values

offset and size, and includes a *chunk token* for that extent. The server will only return up to 1,024 chunk descriptions in one GETTOK call; the client must issue multiple calls for larger files.

The file attributes returned by GETTOK include sufficient information to determine if a local cached copy is up-to-date (as discussed). The tokens allow a client (1) to discover current proxies for the data, (2) to demonstrate read permission for the data to proxies, and (3) to verify the integrity of data retrieved from proxies. First, let us specify how Shark's various tokens and keys are derived.

5.2.3.1 Data integrity via content-based naming

Shark names content with cryptographic hash operations, as given in Table 5.3.

A *file token* is a 160-bit value generated by a cryptographic hash of the file's contents F and some optional per-file randomness r that a server may use as a key for each file (discussed later):

$$T_F = tok(F) = \mathsf{HMAC}_r(F)$$

Throughout our design, HMAC is a keyed hash function [14], which we instantiate with SHA-1. We assume that SHA-1 acts as a collision-resistant hash function, which implies that an adversary cannot find an alternate input pair that yields the same T_F.[1]

[1]While our current implementation uses SHA-1, we could similarly instantiate HMAC with SHA-256 for greater security.

The *chunk token* T_{F_i} in a chunk description is also computed in the same manner, but only uses the particular chunk of data (and optional randomness) as an input to SHA-1, instead of the entire file F. As file and chunk tokens play similar roles in the system, we use T to refer to either type of token indiscriminately.

The *indexing key* I used in Shark's distributed index is simply computed by $\mathsf{HMAC}_T(\mathsf{I})$. We key the HMAC function with T and include a special character I to signify indexing. More specifically, I_F refers to the indexing key for file F, and I_{F_i} for chunk F_i.

The use of such server-selected randomness r ensures that an adversary cannot guess file contents, given only I. Otherwise, if the file is small or stylized, an adversary may be able to perform an offline brute-force attack by enumerating all possibilities.

On the flip-side, omitting this randomness enables cross-file-system sharing, as its content-based naming can be made independent of the file server. That is, when r is omitted and replaced by a string of 0s, the distributed indexing key is dependent only on the contents of F: $I_F = \mathsf{HMAC}_{\mathsf{HMAC}_0(F)}(\mathsf{I})$. Cross-file-system sharing can improve client performance and server scalability when nearby clients use different servers. Thus, the system allows one to trade-off additional security guarantees with potential performance improvements. By default, we omit this randomness for world-readable files, although configuration options can override this behavior.

5.2.3.2 The cooperative-caching read protocol

We now specify in detail the cooperative-caching protocol used by Shark. The main goals of the protocol are to reduce the load on the server and to improve client-perceived latencies. To this end, a client tries to download chunks of a file from multiple proxies in parallel. At a high level, a client first fetches the tokens for the chunks that comprise a file. It then contacts nearby proxies holding each chunk (if such proxies exists) and downloads them accordingly. If no other proxy is caching a particular chunk of interest, the client falls back on the server for that chunk.

The client sends a GETTOK RPC to the server and fetches the whole-file token, the chunk tokens, and the file's attributes. It then checks its cache to determine whether it has a fresh local copy of the file. If not, the client runs the following cooperative read protocol.

The client always attempts to fetch k chunks in parallel. Each fetch is assigned a *random* chunk F_i from the list of needed chunks. The client attempts to discover nearby proxies caching that chunk by querying Coral using the primitive $get(I_{F_i} = \mathsf{HMAC}_{T_{F_i}}(\mathsf{I}))$. If this *get* request fails to find a proxy or does not find one within a specified time, the client fetches the chunk from the server. After downloading the entire chunk, the client announces itself in Coral as a proxy for F_i.

If the get request returns several proxies for chunk F_i, the client chooses one with minimal latency and establishes a secure channel with the proxy, as described later. If the security protocol fails (perhaps due to a malicious proxy), the connection to the proxy fails, or a newly specified time is exceeded, the client chooses another proxy from which to download chunk F_i. Upon downloading F_i, the client verifies its integrity by checking whether $T_{F_i} \stackrel{?}{=} tok(F_i)$. If the client fails to successfully download F_i from any proxy after a fixed number of attempts, it falls back onto the origin file server.

To reduce the number of requests to the Coral indexing layer,[2] while a client is downloading a chunk from a proxy, it attempts to reuse the connection to the proxy by *negotiating* for other chunks. The client picks α random chunks still needed. It computes the corresponding α indexing keys and sends these to the proxy. The proxy responds with those γ chunks, among the α requested, that it already has. If $\gamma = 0$, the proxy responds instead with β keys corresponding to chunks that it does have. The client, upon downloading the current chunk, selects a new chunk from among those negotiated (*i.e.,* needed by the client and known to the proxy). The client then proves read permissions on the new chunk and begins fetching the new chunk. If no such chunks can be negotiated, the client terminates the connection.

[2]More recent work [146], also using Rabin fingerprinting for content-based chunking, has shown how to reduce the amount of references stored in the indexing layer—and hence also reduce the number of index lookups—to a constant number (usually around 30). While perhaps not so much a practical concern, we note that this approach sacrifices the ability to *deterministically* find the same chunks that belong to different files and instead relies on *probabilistic* guarantees, based on the similarity of the files to which the chunks belong. That said, we could equally apply the handprint technique from [146] for naming content to Shark.

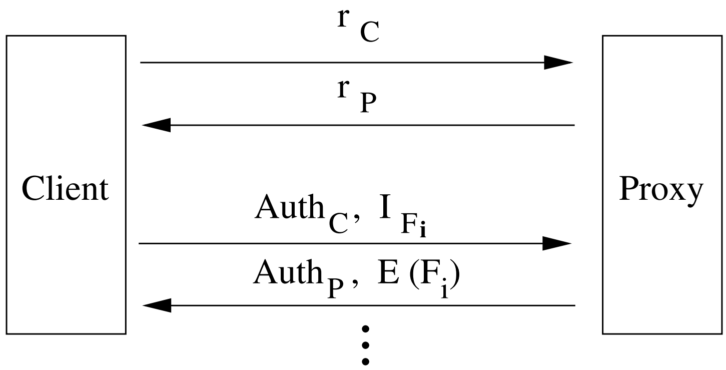


Figure 5.4: Shark session establishment protocol

5.2.3.3 Securing client-proxy interactions

We now describe the secure communication mechanisms between clients and proxies that ensure confidentiality and authorization. We already described how clients achieve data integrity by verifying the contents of files/chunks by their tokens.

To prevent adversaries from passively reading or actively modifying content while in transmission, the client and proxy first derive a symmetric encryption key K_E before transmitting a chunk. As the token T_{F_i} already serves as a shared secret for chunk F_i, the parties can simply use it to generate this key.

Figure 5.4 shows the protocol by which Shark clients establish a secure session. First, the parties exchange fresh, random 20-byte nonces r_C and r_P upon initiating a connection. For each chunk to be sent over the connection, the client must signal the proxy which token T_{F_i} to use, but it can do so without exposing information to eavesdroppers or malicious proxies by simply sending I_{F_i} in the clear. Using these nonces and knowledge of T_{F_i}, each party computes authentication tokens as follows:

$$
\begin{aligned}
Auth_C &= \mathsf{HMAC}_{T_{F_i}}(\mathsf{A}_C, C, P, r_C, r_P) \\
Auth_P &= \mathsf{HMAC}_{T_{F_i}}(\mathsf{A}_P, P, C, r_P, r_C)
\end{aligned}
$$

The $Auth_C$ token proves to the proxy that the client actually has the corresponding chunk token

T_{F_i} and thus read permissions on the chunk. Upon verifying $Auth_C$, the proxy replies with $Auth_P$ and the chunk F_i after applying E to it.

In our current implementation, E is instantiated by a symmetric block encryption function, followed by a MAC covering the ciphertext. However, we note that $Auth_P$ already serves as a MAC for the *content*, and thus this additional MAC is not strictly needed.[3] The symmetric encryption key K_E for E is derived in a similar manner as before:

$$K_E \quad = \quad \mathsf{HMAC}_{T_{F_i}}(\mathsf{E}, C, P, r_C, r_P)$$

An additional MAC key can be similarly derived by replacing the special character E with M. Shark's use of fresh nonces ensure that these derived authentication tokens and keys cannot be replayed for subsequent requests.

Upon deriving this symmetric key K_E, the proxy encrypts the data within a chunk using 128-bit AES in counter mode (AES-CTR). Per each 16-byte AES block, we use the block's offset within the chunk/file as its counter.

The proxy protocol has READ and READDIR RPCs similar to NFS, except they specify the indexing key I and $Auth_C$ to name a file (which is server independent), in place of a file handle. Thus, after establishing a connection, the client begins issuing read RPCs to the proxy; the client decrypts any data it receives in response using K_E and the proper counter (offset).

While this block encryption prevents a client without T_{F_i} from decrypting the data, one may be concerned if some unauthorized client can download a large number of encrypted blocks, with the goal of either wasting the proxy's upstream resources, or with the hope of either learning K_E later or performing some offline attack. The proxy's explicit check of $Auth_C$ prevents this. Similarly, the verifiable $Auth_P$ prevents a malicious party that does not hold F_i from registering itself under the public I_{F_i} and then wasting the client's bandwidth by sending invalid blocks (that later will fail hash verification).

[3]The results of Krawczyk [102] speaking on the *generic* security concerns of "authenticate-and-encrypt" are not really relevant here, as we already expose the raw output of our MAC via I_{F_i} and thus implicitly assume that HMAC does not leak any information about its contents. Thus, the inclusion of $Auth_P$ does not introduce any *additional* data confidentiality concerns.

Thus, Shark provides strong data integrity guarantees to the client and authorization guarantees to the proxy, even in the face of malicious participants.

5.2.4 Implementation

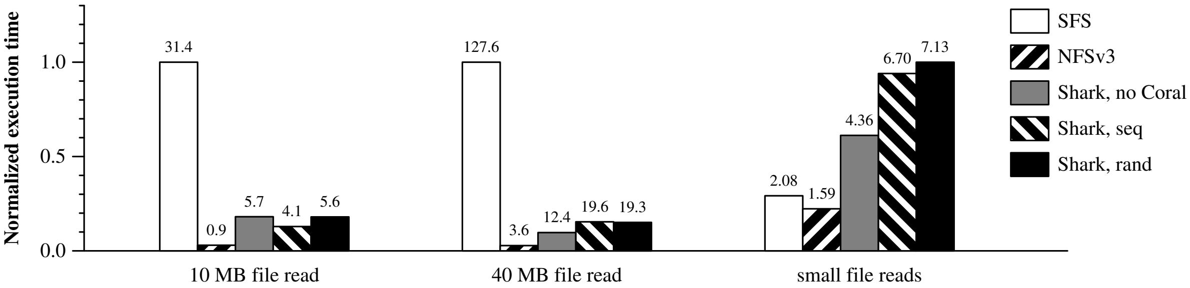
Shark consists of three main components, the server-side daemon `sharksd`, the client-side daemon `sharkcd` and the Coral daemon. All three components are implemented in C++ and are built using the SFS toolkit [118]. The file-system daemons interoperate with the SFS framework, using its automounter, authentication daemon, and so forth. `sharksd` is implemented as a loop-back client which communicates with the kernel NFS server. `sharkcd` comprises an NFS loop-back server which traps local user requests and forwards them to either the origin file server or a Shark proxy. In particular, a read for a file block is intercepted by `sharkcd`'s loop-back server and translated into a series of READ calls to fetch the entire file. The server side of `sharkcd` implements a proxy, accepting connections from other clients. If this proxy cannot immediately satisfy a request, it registers a callback for the request, responding when the block has been fetched.

5.3 Evaluation

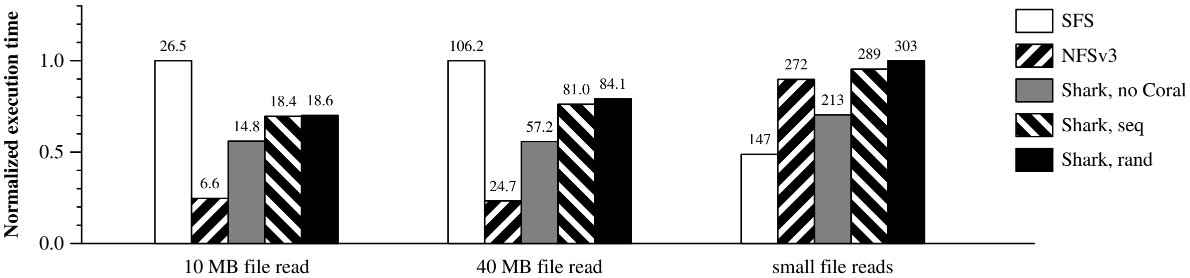
This section evaluates Shark against NFSv3 and SFS in order to quantify the benefits of its cooperative caching design for read-heavy workloads. To measure the performance of Shark against these file systems, without the gain from cooperative caching, we first present microbenchmarks for various types of file-system access tests, both in the local-area and across the wide-area.

Second, we measure Shark's cooperative caching mechanism by performing read tests both within the controlled Emulab LAN environment [188] and in the wide-area on PlanetLab [145]. In all experiments, we start with cold file caches on all clients, but first warm the server's chunk token cache. The server required 0.9 seconds to compute chunks for a 10 MB random file, and 3.6 seconds for a 40 MB random file.

We chose to evaluate Shark on Emulab, in addition to wide-area tests on PlanetLab, in order to test Shark in a more controlled, native environment: While Emulab allows one to completely



(a) Local-area microbenchmarks



(b) Wide-area microbenchmarks

Figure 5.5: *Shark microbenchmarks.* Normalized application performance for various types of file-system access. Execution times in seconds appear above the bars.

reserve machines, individual PlanetLab hosts may be executing tens or hundreds of experiments (slices) simultaneously. In addition, most PlanetLab hosts implement bandwidth caps of 10 Mb/s across all slices. For example, on a local PlanetLab machine operating at NYU, a Shark client took approximately 65 seconds to read a 40 MB file from the local (non-PlanetLab) Shark file server, while a non-PlanetLab client on the same network took 19.3 seconds. Furthermore, deployments of Shark on large LAN clusters (for example, as part of grid computing environments) may experience similar results to those we report.

The server in all the microbenchmarks and the PlanetLab experiments is a 1.40 GHz Athlon at NYU, running OpenBSD 3.6 with 512 MB of memory. It runs the corresponding server daemons for SFS and Shark. All microbenchmark and PlanetLab clients used in the experiments ran Fedora Core 2 Linux. The server used for Emulab tests was a host in the Emulab testbed; it did not simultaneously run a client. All Emulab hosts ran Red Hat Linux 9.0.

The Shark client and server daemons interact with the respective kernel NFS modules using the *loopback* interface. On the Red Hat 9 and Fedora Core 2 machines, where we did our testing, the loopback interface has a maximum MTU of 16436 bytes and any transfer of blocks of size $\geq$ 16 KB results in IP fragmentation which appears to trigger a bug in the kernel NFS code. Since we could not increase the MTU size of the loopback interface, we limited both Shark and SFS to use 8 KB blocks. NFS, on the other hand, issued UDP read requests for blocks of 32 KB over the *ethernet* interface without any problems. These settings could have affected our measurements.

5.3.1 Alternate cooperative protocols

This section considers several alternative cooperative-caching strategies for Shark in order to characterize the benefits of various design decisions.

First, we examine whether clients should issue requests for chunks sequentially (seq), as opposed to choosing a $random$ (previously unread) chunk to fetch. There are two additional strategies to consider when performing sequential requests: Either the client immediately pre-announces itself for a particular chunk upon requesting it (with the concurrent put+get from §2.3.1.4), or the client waits until it finishes fetching a chunk before announcing itself (via a put). We consider such sequential strategies to examine the effect of disk scheduling latency: for single clients in the local area, one intuits that the random strategy limits the throughput to that imposed by the file server's disk seek time, while we expect the network to be the bottleneck in the wide area. Yet, when multiple clients operate concurrently, one intuits that the random strategy allows all clients to fetch independent chunks from the server and later trade these chunks among themselves. Using a purely sequential strategy, the clients all advance only as fast as the few clients that initially fetch chunks from the server.

Second, we disable the $negotiation$ process by which clients may reuse connections with proxies and thus download multiple chunks once connected. In this case, the client must query the distributed index for each chunk.

5.3.2 Microbenchmarks

For the local-area microbenchmarks, we used a local machine at NYU as a Shark client. Maximum TCP throughput between the local client and server, as measured by `ttcp`, was 11.14 MB/s. For wide-area microbenchmarks, we used a client machine located at the University of Texas at El Paso. The average round-trip-time (RTT) between this host and the server, as measured by `ping`, is 67 ms. Maximum TCP throughput was 1.07 MB/s.

Access latency. We measure the time necessary to perform four types of file-system accesses: (1) to read 10 MB and (2) 40 MB large random files on remote hosts, and (3) to read large numbers of small files. The small file test attempts to read 1,000 1 KB files evenly distributed over ten directories.

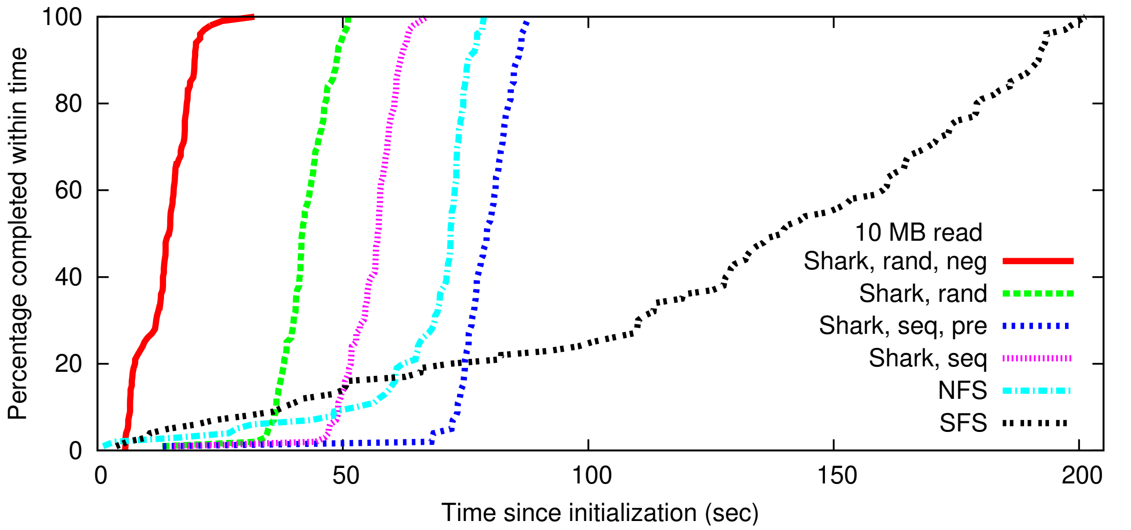
We performed single-client microbenchmarks to measure the performance of Shark. Figure 5.5 shows the performance on the local- and wide-area networks for these three experiments, We compare SFS, NFS, and three Shark configurations; namely, Shark without calls to its distributed indexing layer ($noCoral$), fetching chunks from a file sequentially (seq), and fetching chunks in random order ($rand$). Shark issues up to eight outstanding RPCs (for seq and $rand$, fetching four chunks simultaneously with two outstanding RPCs per chunk). SFS sends RPCs as requested by the NFS client in the kernel.

For all experiments, we report the normalized *median* value over three runs. We interleaved the execution of each of the five file systems over each run. We see that Shark is competitive across different file system access patterns and is optimized for large read operations.

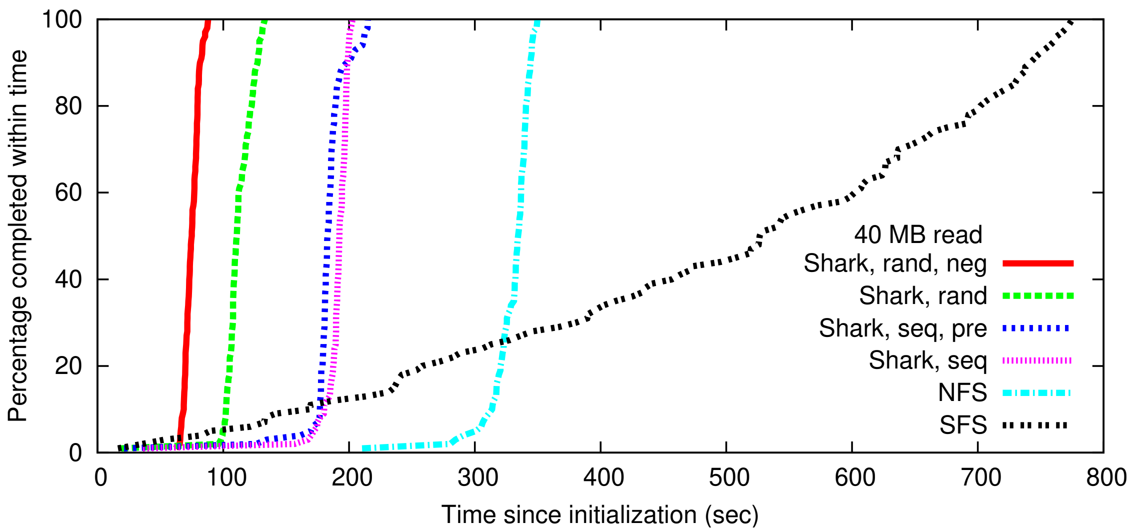
5.3.3 Local-area cooperative caching

Shark's main claim is that it improves a file server's scalability, while retaining its benefits. We now study the end-to-end performance of reads in a cooperative environment with many clients attempting to simultaneously read the same file(s).

In this section, we evaluate Shark on Emulab [188]. In all the configurations of Shark, clients attempt to download a file from four other proxies simultaneously.



(a) Latency for 10 MB file



(b) Latency for 40 MB file

Figure 5.6: *Client latency.* Time (seconds) for ˜100 LAN hosts (Emulab) to finish reading a 10 MB and 40 MB file.

Figure 5.6 shows the cumulative distribution functions (CDFs) of the time needed to read a 10 MB and 40 MB (random) file across 100 physical Emulab hosts, comparing various cooperative read strategies of Shark against vanilla SFS and NFS. In each experiment, all hosts mounted the server and began fetching the file simultaneously. We see that Shark achieves a median completion time less than one-quarter that of NFS and one-sixth that of SFS. Furthermore, its 95th percentile is almost an order of magnitude better than SFS.

Shark's fast, almost vertical rise (for nearly all strategies) demonstrates its cooperative cut-through routing: Shark clients effectively organize themselves into a distribution mesh. Considering a single data segment, a client is part of a chain of nodes performing cut-through routing, rooted at the origin server. Because clients may act as root nodes for some blocks and act as leaves for others, most finish at almost synchronized times. The lack of any degradation of performance in the upper percentiles demonstrates the lack of any heterogeneity, in terms of both network bandwidth and underlying disk/CPU load, among the Emulab hosts.

Interestingly, we see that most NFS clients finish at loosely synchronized times, while the CDF of SFS clients' times has a much more gradual slope, even though both systems send all read requests to the file server. Subsequent analysis of NFS over TCP (instead of NFS over UDP as shown) showed a similar slope as SFS, as did Shark without its cooperative cache. One possible explanation is that the heavy load on (and hence congestion at) the file server imposed by these non-cooperative file systems drives some TCP connections into back-off, greatly reducing fairness.

We find that a $random$ request strategy, coupled with inter-proxy $negotiation$, distinctly outperforms all other evaluated strategies. A sequential strategy effectively saw the clients furthest along in reading a file fetch the leading (four) chunks from the origin file server; other clients used these leading clients as proxies. Thus, modulo possible inter-proxy timeouts and synchronous requests in the non-pre-announce example, the origin server saw at most four simultaneous chunk requests. Using a $random$ strategy, more chunks are fetched from the server simultaneously and thus propagate more quickly through the clients' dissemination mesh.

Figure 5.7 shows the total amount of bandwidth served by each proxy as part of Shark's cooperative caching, when using a random fetch strategy with inter-proxy negotiation for the 40 MB

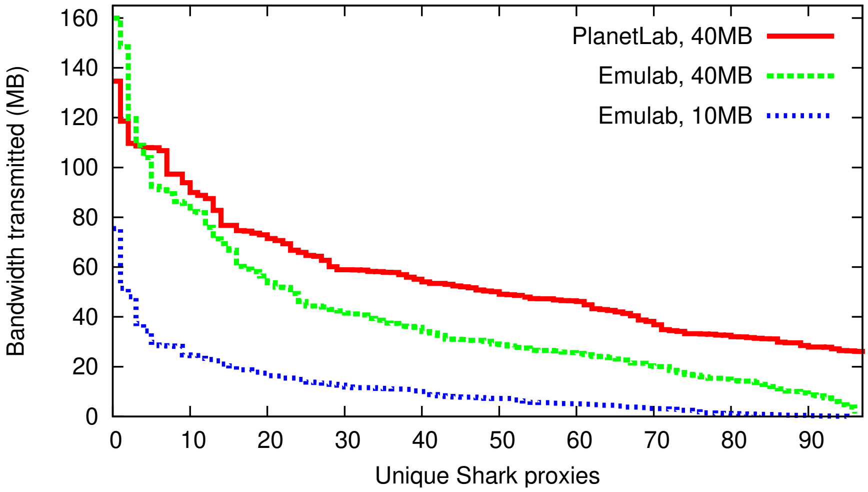


Figure 5.7: *Proxy bandwidth usage.* Upstream bandwidth (MBs) served by each Emulab proxy when reading 40 MB (Emulab and PlanetLab) and 10 MB (Emulab) files.

and 10 MB experiments on Emulab. (The graph also shows the proxy bandwidth use for wide-area experiments on PlanetLab, discussed next.) We see that the proxy serving the most bandwidth contributed four and seven times more upstream bandwidth than downstream bandwidth, respectively. During these experiments, the Shark file server served a total of 92.55 MB and 15.48 MB, respectively. Thus, we conclude that Shark is able to significantly reduce a file server's bandwidth utilization, even when distributing files to large numbers of clients. Furthermore, Shark ensures that any one cooperative-caching client does not assume excessive bandwidth costs.

5.3.4 Wide-area cooperative caching

Shark's main claim is that it improves a file server's scalability, which still maintaining security, accountability, and manageability. In our cooperative caching experiment, we study the end-to-end performance of attempting to perform reads within a large, wide-area distributed testbed.

On approximately 185 PlanetLab hosts, well-distributed across North America, Europe, and Asia, we attempted to simultaneously read a 40 MB random file. All hosts mounted the server and began fetching the file simultaneously.

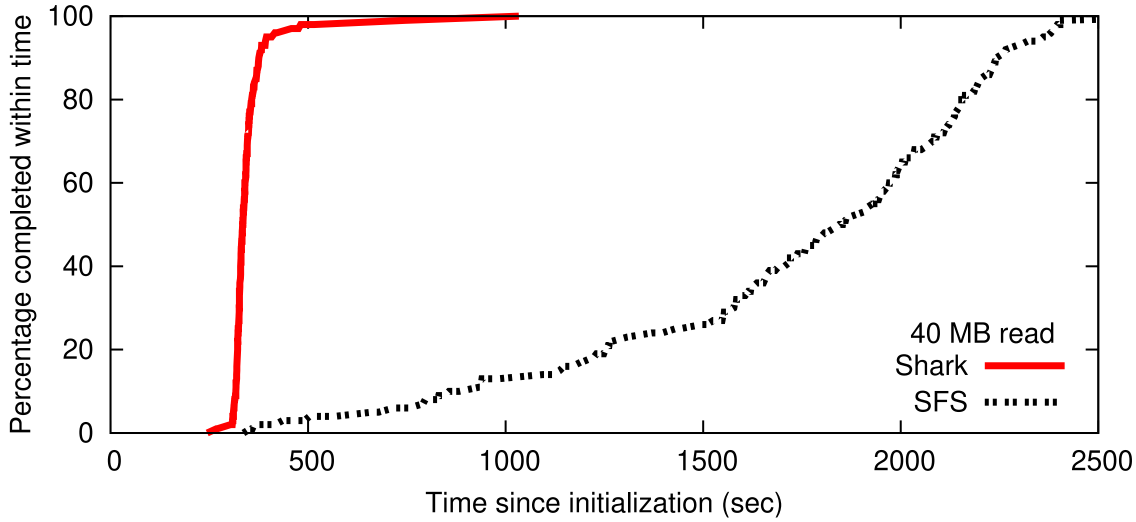


Figure 5.8: *Client latency.* Time (seconds) for 185 wide-area hosts (on PlanetLab) to finish reading a 40 MB file using Shark and SFS.

Figure 5.8 shows a CDF of the time needed to read the file on all hosts, comparing Shark with SFS, which can be summarized with the following percentile results:

% done in (sec)	50%	75%	90%	95%	98%
Shark	334	350	375	394	481
SFS	1848	2129	2241	2364	2396

We see that, between the 50th and 98th percentiles, Shark is five to six times faster than SFS. The graph's sharp rise and distinct knee demonstrates Shark's cooperative caching: 96% of the nodes effectively finish at nearly the same time. Clients in SFS, on the other hand, complete at a much slower rate. Wide-area experiments with NFS repeatedly crashed our file server (*i.e.,* it caused a kernel panic). We were therefore unable to evaluate NFS in the wide area.

Figure 5.7 shows the total amount of bandwidth served by each proxy during this experiment. We see that the proxy serving the most bandwidth contributed roughly three times more upstream than downstream bandwidth.

Finally, Shark reduces the server's bandwidth usage by an order of magnitude compared to SFS: During the experiment for 185 hosts to fetch the same 40 MB file, the Shark file server served

954 MB, while the SFS server served 7400 MB. We believe, however, that Shark's client-cache implementation may be improved to reduce bandwidth usage quite further, *e.g.,* by trading-off better client latency for continually retrying the cooperative cache to minimize server reads.

5.4 Related work on distributed file systems

There are numerous network file systems designed for local-area access. NFS [164] provides a server-based file system, while AFS [85] improves its performance via client-side caching. Some network file systems provide security to operate on untrusted networks, including AFS with Kerberos [176], Echo [113], Truffles [151], and SFS [119]. Even wide-area file systems such as AFS, however, do not perform any bandwidth optimizations necessary for the types of workloads and applications that Shark targets. Additionally, although not an intrinsic limitation of AFS, there are some network environments that do not work as well with its UDP-based transport compared to a TCP-based one. This section describes some complementary and alternate designs for building scalable file systems.

Scalable file servers. JetFile [74] is a wide-area network file system designed to scale to large numbers of clients, by using the Scalable Reliable Multicast (SRM) protocol, which is logically layered on IP multicast. JetFile allocates a multicast address for each file. Read requests are multicast to this address; any client which has the data responds to such requests. In JetFile, any client can become the manager for a file by writing to it—which implies the necessity for conflict-resolution mechanisms to periodically synchronize to a storage server—whereas all writes in Shark are synchronized at a central server. However, this practice implies that JetFile is intended for read-write workloads, while Shark is designed for read-heavy workloads.

High-availability file systems. Several local-area systems propose distributing functionality over multiple collocated hosts to achieve greater fault-tolerance and availability. Zebra [83] uses a single meta-data server to serialize meta-data operations (*e.g.,* i-node operations), and it maintains a per-client log of file contents striped across multiple network nodes. Harp [109] replicates file servers to ensure high availability; one such server acts as a primary replica in order to serialize updates. These techniques are largely orthogonal to, yet possibly could be combined with, Shark's cooperative caching design.

Serverless file systems. Serverless file systems are designed to offer greater local-area scalability by replicating functionality across multiple hosts. xFS [7] distributes data and meta-data across all participating hosts, where every piece of meta-data is assigned a host at which to serialize updates for that meta-data. Frangipani [182] decentralizes file storage among a set of virtualized disks, and it maintains traditional file system structures, with small meta-data logs to improve recoverability. A Shark server can similarly use any type of log-based or journaled file system to enable recoverability, while it is explicitly designed for wide-area scalability.

Farsite [1] seeks to build an enterprise-scale distributed file system. A single primary replica manages file writes, and the system protects directory meta-data through a Byzantine-fault-tolerant protocol [27]. When enabling cross-file-system sharing, Shark's encryption technique is similar to Farsite's convergent encryption, in which files with identical content result in identical ciphertexts.

Peer-to-peer file systems. A number of peer-to-peer file systems—including PAST [161], CFS [45], Ivy [127], and OceanStore [105]—have been proposed for wide-area operation and similarly use some type of DHT infrastructure for content discovery ([162, 177, 195], respectively). All of these systems differ from Shark in that they provide a serverless design: While such a fully decentralized design removes any central point of failure, it adds complexity, performance overhead, and management difficulties.

PAST and CFS are both designed for read-only data, where data (whole files in PAST and file blocks in CFS) are *stored* in the peer-to-peer DHT [162, 177] at nodes closest to the key that names the respective block/file. Data replication helps improve performance and ensures that a single node is not overloaded. In contract, Shark uses Coral to *index* clients caching a replica, so data is only cached where it is needed by applications and on nodes that have proper access permissions to the data.

Ivy builds on CFS to yield a read-write file system through logs and version vectors. The head of a per-client log is stored in the DHT at its closest node. To enable multiple writers, Ivy uses version vectors to order records from different logs. It does not guarantee read/write consistency. Also managing read/write storage via versioned logs, OceanStore divides the system into a large set of untrusted clients and a core group of trusted servers, where updates are applied atomically. Its Pond prototype [154] uses a combination of Byzantine-fault-tolerant protocols, proactive threshold signatures, erasure-encoded and block replication, and multicast dissemination mechanisms.

Chapter 6

Securing Rateless Erasure Codes for Large-File Distribution

The previous chapter on Shark introduced a new content distribution design in the context of file systems, which must handle the complexity of managing the consistency of read/write data. The final mechanism this thesis presents returns to the problem of delivering read-only content. However, instead of web content—with its mostly small- to medium-sized objects—we now consider a clean-slate approach to distributing large files. Specifically, we present a novel protocol to secure the dissemination of erasure-encoded content.

6.1 Motivation

Peer-to-peer file-sharing systems are trafficking larger and larger files, but end-users have not witnessed meaningful increases in their available bandwidth, nor have individual nodes become more reliable. As a result, the transfer times of files in these networks often exceed the average uptime of source nodes, and receivers frequently experience download truncations.

These peer-to-peer, large-file content distribution networks (LF-CDNs) can be extremely wasteful of bandwidth if they solely use whole-file, unicast transmissions, as a small number of files account for a sizable percentage of total transfers. Studies indicate that from a university network, for instance, KaZaa's 300 top bandwidth-consuming objects can account for 42% of all outbound traffic [165]. This overhead is partly due to these peer-to-peer systems ignoring locality: Recall that Karagiannis *et al.* [93] found that up to 90% of peer-to-peer *ingress* traffic into networks corresponds to data already residing locally.

Combining good peer selection (§4) with multicasting popular files might drastically reduce the total bandwidth consumed, by minimizing the frequency with which data repeatedly traverses expensive, long-distance links. Traditional multicast systems would fare poorly in such unstable networks, however, due to the overhead of maintaining the multicast structure (often a tree) upon node failures.

Developments in practical erasure codes [111] and *rateless* erasure codes [110, 115, 169] point to elegant solutions for this problem. Erasure codes of rate r (where $0 < r < 1$) map a file of n *message blocks* onto a larger set of n/r *check blocks*. Using such a scheme, a sender simply transmits a random sequence of these check blocks. A receiver can decode the original file with high probability once he has amassed a random collection of slightly more than n unique check blocks. At larger values of r, senders and receivers must carefully coordinate to avoid block duplication. In rateless codes, block duplication is much less of a problem: encoders need not pre-specify a value for r and can instead map a file's blocks to a set of check blocks whose size is exponential in n.

When using low-rate or rateless erasure codes, senders and receivers forgo the costly and complicated feedback protocols often needed to reconcile truncated downloads or to maintain a reliable multicast tree. Receivers can furthermore collect blocks from multiple senders simultaneously. One can envision an ideal situation, in which many senders transmit the same file to many recipients in a "forest of multicast trees." No retransmissions are needed when receivers and senders leave and re-enter the network, as they frequently do.

A growing body of literature considers erasure codes in the context of modern distributed systems. Earlier work applied fixed-rate codes to centralized multicast CDNs [20, 21]. More current work considers rateless erasure codes in unicast, multi-source LF-CDNs [22, 117]. Even more recently, SplitStream [30] has explored applying rateless erasure codes to overlapping peer-to-peer multicast networks, and Bullet [101] calls on these codes when implementing "overlay meshes."

There is a significant downside to this popular approach. When transferring erasure-encoded files, receivers can only "preview" their file at the very end of the transfer. A receiver may discover that, after dedicating hours or days of bandwidth to a certain file transfer, he was receiving incorrect or useless blocks all along. Most prior work in this area assumes honest senders, but architects of robust, real-world LF-CDNs cannot make this assumption.

This chapter describes a novel construction that lets recipients verify the integrity of check blocks immediately, before consuming large amounts of bandwidth or polluting their download caches. In our scheme, a file F is compressed down to a smaller hash value, $H(F)$, with which the receiver can verify the integrity of any possible check block. Receivers then need only obtain a file's correct hash value to avoid being duped during a transfer. Our function H is based on a discrete-log-based, collision-resistant, homomorphic hash function, which allows receivers to compose hash values in much the same way that encoders compose message blocks. Unlike more obvious constructions, ours is independent of encoding rate and is therefore compatible with rateless erasure codes. It is fast to compute, efficiently verified using probabilistic batch verification, and has provable security under the discrete-log assumption. Furthermore, our implementation results suggest this scheme is practical for real-world use.

Note that there is a parallel between this hash value and Shark's use of a proxy authenticator token (§5.2.3.3). That token plays a similar role—namely, a means to prevent a malicious sender from wasting a receiver's bandwidth—but in a much simpler context. In Shark, a fingerprint of *every* chunk's contents is known *a priori* to each party, retrieved from a Shark file server. On the other hand, as there are an exponential number of *potential* erasure-encoded check blocks in this context, the contents of these blocks are unknown to the receiver. Hence, the receiver cannot simply download all potential hashes from a file publisher.

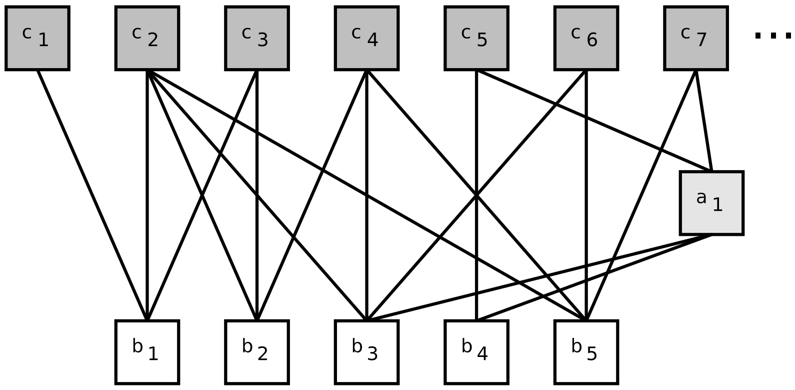


Figure 6.1: *Online encoding of a five-block file.* b_i are message blocks, a_1 is an auxiliary block, and c_i are check blocks. Edges represent addition (via XOR). For example, $c_4 = b_2 + b_3 + b_5$, $a_1 = b_3 + b_4$, and $c_7 = a_1 + b_5$.

6.2 Design

6.2.1 Brief review of erasure codes

This chapter considers the non-streaming transfer of very large files over erasure channels such as the Internet. Typically, a file F is divided into n uniformly-sized blocks, known as *message blocks* (or alternatively, input symbols). Erasure-encoding schemes add redundancy to the original n message blocks, so that receivers can recover from packet drops without explicit packet retransmissions.

Though traditional forward error correction codes such as Reed-Solomon are applicable to erasure channels [158], their decoding times—quadratic in n—make them prohibitively expensive for large files. To this effect, researchers have proposed a class of erasure codes with sub-quadratic decoding times. Examples include Tornado Codes [21], LT Codes [110], Raptor Codes [169] and Online Codes [115]. All four of these schemes output *check blocks* (or alternatively, output symbols) that are simple summations of message blocks. That is, if the file F is composed of message blocks b_1 through b_n, the check block c_1 might be computed as $b_1 + b_2$. The specifics of these linear relationships vary with the scheme.

Tornado Codes, unlike the other three, are fixed-rate. A sender first chooses a rate r and then can generate no more than n/r check blocks. Furthermore, the encoding process grows more expensive as r approaches zero. For multicast and other applications that benefit from lower encoding rates, LT, Raptor, and Online codes are preferable [117]. Unlike Tornado codes, they feature rateless encoders that can generate an enormous sequence of check blocks with state constant in n. LT codes are decodable in time $O(n \ln(n))$, while Tornado, Raptor and Online Codes have linear-time decoders.

We use Online Codes when considering the specifics of the encoding and decoding processes; however, all three rateless techniques are closely related, and the techniques we describe are equally applicable to LT and Raptor Codes.

Online codes [115]. Online Codes consist of three logical components: a precoder, an encoder, and a decoder. A sender initializes the encoding scheme via the precoder, which takes as input a file F with n message blocks and outputs $n\delta k$ *auxiliary blocks*. k is small constant such as 3, and δ, a parameter discussed later, has a value such as .005. The precoder works by adding each message block to k distinct randomly-chosen auxiliary blocks. An auxiliary block is thus the sum of $1/\delta$ message blocks on average. This process need not be random in practice; the connections between the message and auxiliary blocks can be a deterministic function of the input size n, and the parameters k and δ. Finally, the n message blocks and the $n\delta k$ auxiliary blocks are considered together as a composite file F' of size $n' = n(1 + \delta k)$, which is suitable for encoding.

To construct the i^{th} check block, the encoder randomly samples a pre-specified probability distribution for a value d_i, known as the check block's *degree*. The encoder then selects d_i blocks from F' at random, and computes their sum, c_i. The outputted check block is a pair $\langle x_i, c_i \rangle$, where x_i describes which blocks were randomly chosen from F'. In practice, an encoder can compute the degree d_i and the meta-data x_i as the output of a pseudo-random function on input (i, n). It thus suffices to send $\langle i, c_i \rangle$ to the receiving client, who can compute x_i with knowledge of n, the encoding parameters, and access to the same pseudo-random function. See Figure 6.1 for a schematic example of precoding and encoding.

To recover the file, a recipient collects check blocks of the form $\langle x_i, c_i \rangle$. Assume a received block has degree one; that is, it has meta-data x_i of the form $\{j\}$. Then, c_i is simply the j^{th} block of the file F', and it can be marked *recovered*. Once a block is recovered, the decoder subtracts it from the appropriate *unrecovered* check blocks. That is, if the k^{th} check block is such that $j \in x_k$, then b_j is subtracted from c_k, and j is subtracted from x_k. Note that during this subtraction process, other blocks might be recovered. If so, then the decoding algorithm continues iteratively. When the decoder receives blocks whose degree is greater than one, the same type of process applies; that is, all recovered blocks are subtracted from it, which might in turn recover it.

In the encoding process, auxiliary blocks behave like message blocks; in the decoding process, they behave like check blocks. When the decoder recovers an auxiliary block, it then adds it to the pool of unrecovered check blocks. When the decoder recovers a message block, it simply writes the block out to a file in the appropriate location. Decoding terminates once all n message blocks are recovered.

In the absence of the precoding step, the codes are expected to recover $(1-\delta)n$ message blocks from $(1+\epsilon)n$ check blocks, as n becomes large. The auxiliary blocks introduced in the precoding stage help the decoder to recover the final δn blocks. A sender specifies δ and ϵ prior to encoding; they in turn determine the encoder's degree distribution and consequently the number of block operations required to decode.

Online Codes, like the other three schemes, use bitwise exclusive OR (XOR) for both addition and subtraction. We note that although XOR is fast, simple, and compact—*i.e.,* XORing two blocks does not produce carry bits—it is not essential. Any efficiently invertible operation suffices.

6.2.2 Threat model

Deployed LF-CDNs like KaZaa consist of nodes who function simultaneously as publishers, mirrors, and downloaders of content. Nodes transfer content by sending contiguous file chunks over point-to-point links, with few security guarantees. We imagine a similar but more powerful network model.

When a node wishes to publish F, he uses a collision-resistant hash function such as SHA-1 to derive a succinct cryptographic file handle, $H(F)$. He then pushes F into the network and also publicizes the mapping of the file's name $N(F)$ to its key, $H(F)$. Mirrors maintain local copies of the file F and transfer erasure encodings of it to multiple clients simultaneously. As downloaders receive check blocks, they can forward them to other downloaders, harmlessly "downsampling" if constrained by downstream bandwidth. Once a downloader fully recovers F, he generates his own encoding of F, sending "fresh" check blocks to downstream recipients. Meanwhile, erasure codes enable downloaders to collect check blocks concurrently from multiple sources.

This setting differs notably from traditional multicast settings. Here, internal nodes are not mere packet-forwarders but instead are active nodes that produce unique erasure encodings of the files they redistribute.

Unfortunately, in a LF-CDN, one must assume that adversarial parties control arbitrarily many nodes on the network. Hence, mirrors may be frequently malicious.[1] Under these assumptions, the LF-CDN model is vulnerable to a host of different attacks:

1. ***Content Mislabeling.*** A downloader's original lookup mapped $N(F) \rightarrow H(\tilde{F})$. The downloader will then request and receive the file $\tilde{F}$ from the network, even though he expected file F.

2. ***Bogus-Encoding Attacks.*** Mirrors send blocks that are not check blocks of the expected file, with the intent of thwarting the downloader's decoding. This has also been termed a pollution attack [96].

3. ***Distribution Attacks.*** A malicious mirror sends valid check blocks from the encoding of F, but not according to the correct distribution. As a result, the receiver might experience degenerate behavior when trying to decode.

Deployed peer-to-peer networks already suffer from malicious content-mislabeling. A popular file may resolve to dozens of names, only a fraction of which are appropriately named. A number of solutions exist, ranging from simply downloading the most widely replicated name (on the assumption that people will keep the file if it is valid), to more complex reputation-based schemes.

[1]We do not explicitly model adversaries controlling the underlying physical routers or network trunks, although our techniques are also robust against these adversaries, with the obvious limitations (*e.g.,* the adversary can prevent a transfer if he blocks the downloader's network access).

In more interesting LF-CDNs, trusted publishers might sign file hashes. Consider the case of a Linux vendor using a LF-CDN to distribute large binary upgrades. If the vendor distributes its public key in CD-based distributions, clients can verify the vendor's signature of any subsequent upgrade. This is also the context we considered earlier in CoralCDN and Shark, where an authentic "origin" server or publisher is known for each particular file. The general mechanics of reliable filename resolution are beyond the scope of our consideration; we assume that a downloader can retrieve $H(F)$ given $N(F)$ via some out-of-band and trusted lookup.

This work focuses on the bogus-encoding attack. When transferring large files, receivers will talk to many different mirrors, in series and in parallel. At the very least, the receiver should be able to distinguish valid from bogus check blocks at decoding time. One bad block should not ruin hundreds of thousands of valid ones. Moreover, receivers have limited bandwidth and cannot afford to communicate with all possible mirrors on the network simultaneously. They would clearly benefit from a mechanism to detect cheating as it happens, so they can terminate connections to bad servers and seek out honest senders elsewhere on the network.

To protect clients against encoding attacks, LF-CDNs require some form of source verification. That is, downloaders need a way to verify individual check blocks, given a reliable and compact hash of the desired file. Furthermore, this verification must not be interactive; it should work whether or not the original publisher is online. The question becomes, should the original publisher authenticate file blocks before or after they are encoded? We consider both cases.

6.2.2.1 Hashing all input symbols

A publisher wishes to distribute an n-block file F. Assuming Online Codes, he first runs F through a precoder, yielding an n'-block file F'. He then computes a Merkle hash tree of F' [120]. The file's full hash is the entirety of the hash tree, but the publisher uses the hash tree's root for the file's succinct cryptographic handle. To publish, he pushes the file and the file's hash tree into the network, all keyed by the root of the hash tree. Note that although the hash tree is smaller than the original file, its size is still linear in n.

To download F, a client maps $N(F)$ to $H(F)$ as usual, but now $H(F)$ is the root of the file's hash tree. Next, the client retrieves the rest of the hash tree from the network, and is able to verify its consistency with respect to its root. Given this information, he can verify check blocks as the decoding progresses, through use of a "smart decoder." As check blocks of degree one arrive, he can immediately verify them against their corresponding leaf in the hash tree. Similarly, whenever the decoder recovers an input symbol b_j from a check block $\langle x_i, c_i \rangle$ of higher degree, the receiver verifies the recovered block b_j against its hash. If the recovered block verifies properly, then the receiver concludes that $\langle x_i, c_i \rangle$ was generated honestly and hence is *valid*. If not, then the receiver concludes that it is bogus.

In this process, the decoder only XORs check blocks with validated degree-one blocks. Consequently, valid blocks cannot be corrupted during the decoding process. On the other hand, invalid check blocks which are reduced to degree-one blocks are easily identified and discarded. Using this "smart decoder," a receiver can trivially distinguish bogus from valid check blocks and need not worry about the download cache pollution described in [96]. The problem, however, is that a vast majority of these block operations happen at the very end of the decoding process—when almost n check blocks are available to the decoder. Figure 6.2 exhibits the average results for decoding a file of $n = 10,000$ blocks, taken over 50 random Online encodings. According to these experiments, when a receiver has amassed $.9n$ check blocks, he can recover only $.068n$ message blocks; when he has amassed n check blocks, he can recover only $.303n$ message blocks. In practice, a downloader could dedicate days of bandwidth to receiving gigabytes of check blocks, only to find that most are bogus.

6.2.2.2 Hashing check blocks

Instead of hashing the input to the erasure encoder, publishers might hash its output. If so, the LF-CDN is immediately limited to fixed-rate codes. Recall that the publisher is not directly involved in the file's ultimate distribution to clients and therefore cannot be expected to hash and sign check blocks on-the-fly. Thus, the publisher must pre-specify a tractable rate r and "pre-authorize" n/r check blocks. In practice, the publisher might do this by generating n/r check blocks, computing their hash tree, and keying the file by its root. When mirrors distribute the file, they distribute

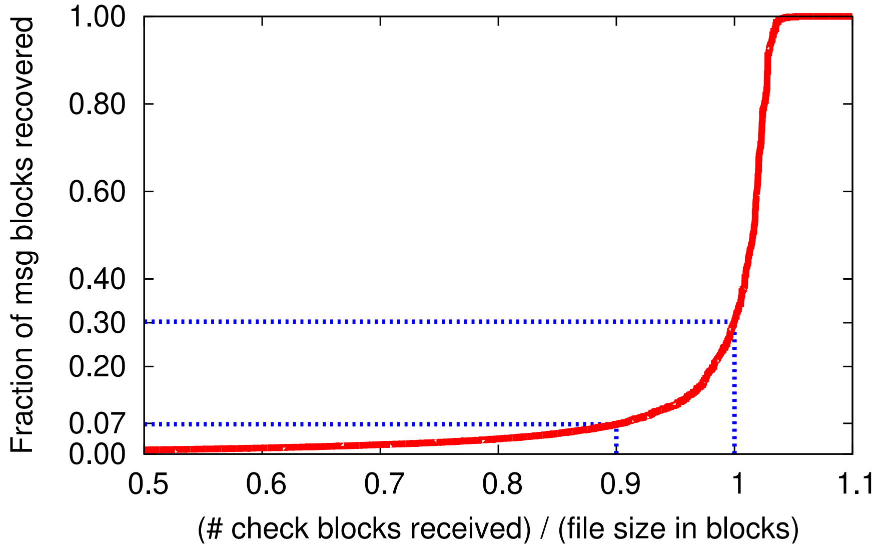


Figure 6.2: Number of blocks recoverable as function of number of blocks received. Data collected over 50 random encodings of a 10,000 block file.

only those check blocks that the publisher has pre-authorized. With the benefit of the hash tree taken over all possible check blocks, the receiver can trivially verify check blocks as they arrive. Section §6.3.2.2 explores this proposal in more detail. We simply observe here that it becomes prohibitively expensive for encoding at low rates, in terms of the publisher's computational resources and the bandwidth required to distribute hashes.

6.2.3 Homomorphic hashing

Our solution combines the advantages of the previous section's two approaches. As in the first scheme, our hashes are reasonably-sized and independent of the encoding rate r. As in the second, they enable receivers to authenticate check blocks on-the-fly.

We propose two possible authentication protocols based on a homomorphic collision-resistant hash function (CRHF). In the *global hashing* model, there is a single way to map F to $H(F)$ by using global parameters. As such, one-time hash generation is slow but well-defined. In the *per-publisher hashing* model, each publisher chooses his own hash parameters, and different publishers will generate different hashes for the same file. We will later show that the per-publishing model enables publishers to generate hashes more efficiently, although the downloader's verification overhead is the same.

In today's file-sharing systems, there may be multiple publishers for the same exact content—*e.g.,* different individuals may rip the same CD—thus these publishers may use global hashing so that all copies look identical to the system. In other environments, content has a single, well-known publisher, and the per-publisher scheme is more appropriate. While the latter might be ill-suited for copyright circumvention, it otherwise is more useful, allowing publishers to sign file hashes and clients to authenticate filename to file-hash mappings. Many Internet users could benefit from cheap, trusted, and efficient distribution of bulk data: anything from Linux binary distributions to large academic datasets to high-definition videos could travel through such a network.

6.2.3.1 Notation and preliminaries

In the following discussion, we will be using scalars, vectors, and matrices defined over modular subgroups of $\mathbb{Z}$. We write scalars in lowercase (*e.g.,* x), vectors in lowercase boldface (*e.g.,* $\mathbf{x}$) and matrices in uppercase (*e.g.,* X). Furthermore, for the matrix X, the j^{th} column is a vector written as $\mathbf{x}_j$, and the ij^{th} cell is a scalar written as x_{ij}. Vectors might be row vectors or column vectors, and we explicitly specify them as such. All additions are assumed to be taken over $\mathbb{Z}_q$, and multiplications and exponentiations are assumed to be taken over $\mathbb{Z}_p$, with q and p selected as described in the next subsection. Finally, we invent one notational convenience concerning vector exponentiation. That is, we define $g^{\mathbf{r}} = \mathbf{g}$ component-wise: if the row vector $\mathbf{r} = (r_1 \; r_2 \; \cdots \; r_m)$, then the row vector $g^{\mathbf{r}} = (g^{r_1} \; g^{r_2} \; \cdots \; g^{r_m})$.

6.2.3.2 Global homomorphic hashing

In global homomorphic hashing, all nodes on the network must agree on hash parameters so that any two nodes independently hashing the same file F should arrive at exactly the same hash. To achieve this goal, all nodes must agree on security parameters λ_p and λ_q. Then, a trusted party globally generates a set of hash parameters $G = (p, q, \mathbf{g})$, where p and q are two large random

Name	Description	*e.g.,*
λ_p	discrete log security parameter (in bits)	1024
λ_q	discrete log security parameter (in bits)	257
p	random prime, $\|p\| = \lambda_p$	
q	random prime, $q\|(p-1)$, $\|q\| = \lambda_q$	
β	block size	16 KB
m	$= \lceil \beta/(\lambda_q - 1) \rceil$ (number of "sub-blocks" per block)	512
$\mathbf{g}$	$1 \times m$ row vector of order q elts in $\mathbb{Z}_p$	
G	hash parameters, given by $(p, q, \mathbf{g})$	
n	original file size (in blocks)	$65,536$
k	precoding parameter	3
δ	fraction of unrecoverable message blocks (without the benefit of precoding)	.005
n'	precoded file size, $n' = (1 + \delta k)n$	$66,519$
ϵ	asymptotic encoding overhead	.01
d	average degree of check blocks	~ 8.17

Figure 6.3: System parameters and properties for securing erasure codes

primes such that $|p| = \lambda_p$, $|q| = \lambda_q$, and $q|(p-1)$. The hash parameter $\mathbf{g}$ is a $1 \times m$ row-vector, composed of random elements of $\mathbb{Z}_p$, all order q. These and other parameters are summarized in Figure 6.3.

In decentralized LF-CDNs, such a trusted party might not exist. Rather, users joining the system should demand "proof" that the group parameters G were generated honestly. In particular, no node should know i, j, x_i, x_j such that $g_i^{x_i} = g_j^{x_j}$, as one that had this knowledge could easily compute hash collisions. The generators might therefore be generated according to the algorithm PickGroup given in Figure 6.4. The input $(\lambda_p, \lambda_q, m, s)$ to the PickGroup algorithm serves as a heuristic proof of authenticity for the output parameters, $G = (p, q, \mathbf{g})$. That is, unless an adversary exploits specific properties of SHA-1, he would have difficulty computing a seed s that yields generators with a known logarithmic relation. In practice, the seed s might be chosen globally, or even chosen per file F such that $s = \mathsf{SHA1}(N(F))$. Either way, the same parameters G will always be used when hashing file F.

```
Algorithm PickGroup(λ_p, λ_q, m, s)
    Seed PRNG 𝒢 with s.
    do
        q ← qGen(λ_q)
        p ← pGen(q, λ_p)
    while p = 0 done
    for i = 1 to m do
        do
            x ← 𝒢(p − 1) + 1
            g_i ← x^((p−1)/q)   (mod p)
        while g_i = 1 done
    done
    return (p, q, g)
```

```
Algorithm qGen(λ_q)
    do
        q ← 𝒢(2^λ_q)
    while q is not prime done
    return q
```

```
Algorithm pGen(q, λ_p)
    for i = 1 to 4λ_p do
        X ← 𝒢(2^λ_p)
        c ← X   (mod 2q)
        p ← X − c + 1        // Note p ≡ 1   (mod 2q)
        if p is prime then return p
    done
    return 0
```

Figure 6.4: *Picking secure generators.* The seed s can serve as an heuristic "proof" that the hash parameters were chosen honestly. This algorithm is based on that given in the NIST Standard [56]. The notation $\mathcal{G}(x)$ should be taken to mean that the pseudo-random number generator $\mathcal{G}$ outputs the next number in its pseudo-random sequence, scaled to the range $\{0, \ldots, x-1\}$.

File representation. As per Figure 6.3, let β be the block size, and let $m = \lceil \beta/(\lambda_q - 1) \rceil$. Consider a file F as an $m \times n$ matrix, whose cells are all elements of $\mathbb{Z}_q$. Our selection of m guarantees that each element is less than $2^{\lambda_q - 1}$, and is therefore less than the prime q. Now, the j^{th} column of F simply corresponds to the j^{th} message block of the file F, which we write $\mathbf{b}_j = (b_{1,j}, \ldots, b_{m,j})$. Thus:

$$F = (\mathbf{b}_1 \, \mathbf{b}_2 \, \cdots \, \mathbf{b_n}) = \begin{pmatrix} b_{1,1} & \cdots & b_{1,n} \\ \vdots & \ddots & \vdots \\ b_{m,1} & \cdots & b_{m,n} \end{pmatrix}$$

We add two blocks by adding their corresponding column-vectors. That is, to combine the i^{th} and j^{th} blocks of the file, we simply compute:

$$\mathbf{b}_i + \mathbf{b}_j = (b_{1,i} + b_{1,j}, \ldots, b_{m,i} + b_{m,j}) \bmod q$$

Precoding. Recall that the precoding stage in Online Codes produces auxiliary blocks that are summations of message blocks, and that the resulting composite file has the original n message blocks and the additional $n\delta k$ auxiliary blocks. The precoder now proceeds as usual, but uses addition over $\mathbb{Z}_q$ instead of the XOR operator.

We can represent this process succinctly with matrix notation. That is, the precoding stage is given by a binary $n \times n'$ matrix, $Y = (I|P)$. The matrix Y is the concatenation of the $n \times n$ identity matrix I, and the $n \times n\delta k$ matrix P that represents the composition of auxiliary blocks. All rows of P sum to k, and its columns sum to $1/\delta$ on average. The precoded file can be computed as $F' = FY$. The first n columns of F' are the message blocks. The remaining $n\delta k$ columns are the auxiliary blocks. For convenience, we refer to auxiliary blocks as $\mathbf{b}_i$, where $n < i \leq n'$.

Encoding. Like precoding, encoding is unchanged save for the addition operation. For each check block, the encoder picks an n'-dimensional bit vector $\mathbf{x}$ and computes $\mathbf{c} = F'\mathbf{x}$. The output $\langle \mathbf{x}, \mathbf{c} \rangle$ fully describes the check block.

Hash generation. To hash a file, a publisher uses a CRHF, secure under the discrete-log assumption. This hash function is a generalized form of the Pederson commitment scheme [138] (and from Chaum *et al.* [35]), and it is similar to that used in various incremental hashing schemes (see §6.4). Recall that a CRHF is informally defined as a function for which finding any two inputs that yield the same output is difficult.

For an arbitrary message block $\mathbf{b}_j$, define its hash with respect to G:

$$h_G(\mathbf{b}_j) = \prod_{i=1}^{m} g_i^{b_{i,j}} \bmod p \tag{6.1}$$

Define the hash of file F as a $1 \times n$ row-vector whose elements are the hashes of its constituent blocks:

$$H_G(F) = (h_G(\mathbf{b}_1)\ h_G(\mathbf{b}_2)\ \cdots\ h_G(\mathbf{b}_n)) \tag{6.2}$$

To convey the complete hash, publishers should transmit both the group parameters and the hash itself: $(G, H_G(F))$. From this construction, it can be seen that each block of the file is β bits, and the hash of each block is λ_p bits. Hence, the hash function H_G reduces the file by a factor of β/λ_p, and therefore $|H_G(F)| = |F|\lambda_p/\beta$.

Hash verification. If a downloader knows $(G, H_G(F))$, he can first compute the hash values for the $n\delta k$ auxiliary blocks. Recall that the precoding matrix Y is a deterministic function of the file size n and the preestablished encoding parameters δ and k. Thus, the receiver computes Y and obtains the hash over the composite file as $H_G(F') = H_G(F) \cdot Y$. The hash of the auxiliary blocks are the last $n\delta k$ cells in this row vector.

To verify whether a given check block $\langle \mathbf{x}, \mathbf{c} \rangle$ satisfies $\mathbf{c} = F'\mathbf{x}$, a receiver verifies that:

$$h_G(\mathbf{c}) = \prod_{i=1}^{n'} h_G(\mathbf{b_i})^{x_i} \tag{6.3}$$

h_G functions here as a *homomorphic hash function*. For any two blocks $\mathbf{b}_i$ and $\mathbf{b}_j$, $h_G(\mathbf{b}_i + \mathbf{b}_j) = h_G(\mathbf{b}_i)h_G(\mathbf{b}_j)$.

Downloaders should monitor the aggregate behavior of mirrors during a transfer. If a downloader detects a number of unverifiable check blocks above a predetermined threshold, he should consider the sender malicious and should terminate the transfer.

Decoding. Decoding proceeds as described in §6.2.1. Of course, XOR is conveniently its own inverse, so implementations of standard Online Codes need not distinguish between addition and subtraction. In our case, we simply use subtraction over $\mathbb{Z}_q$ to reduce check blocks as necessary.

Despite our emphasis on Online Codes in particular, we note that these techniques apply to LT and Raptor codes as well. LT Codes do not involve preprocessing, so the above scheme can be simplified. Raptor Codes involve a two-stage precoding process, and they probably are not compatible with the implicit calculation of auxiliary block hashes described above. In this case, we compute file hashes over the output of the precoder, therefore obtaining slightly larger file hashes.

6.2.3.3 Per-publisher homomorphic hashing

The per-publisher hashing scheme is an optimization of the global hashing scheme just described. In the per-publisher hashing scheme, a given publisher picks group parameters G so that a logarithmic relation among the generators $\mathbf{g}$ is known. The publisher picks q and p as above, but generates $\mathbf{g}$ by picking a random $g \in \mathbb{Z}_p$ of order q, generating a random vector $\mathbf{r}$ whose elements are in $\mathbb{Z}_q$ and then computing $\mathbf{g} = g^{\mathbf{r}}$.

Given the parameters g and $\mathbf{r}$, the publisher can compute file hashes with many fewer modular exponentiations:

$$H_G(F) = g^{\mathbf{r}F} \tag{6.4}$$

The publisher computes the product $\mathbf{r}F$ first, and then performs only one modular exponentiation per file block to obtain the full file hash. See §6.3.2 for a more complete running-time analysis. The publisher must be careful to never reveal g and $\mathbf{r}$; doing so allows an adversary to compute arbitrary collisions for H_G.

Aside from hash parameter generation and hash generation, all aspects of the protocol described above hold for both the per-publisher and the global scheme. A verifier does not distinguish between the two types of hashes, beyond ensuring that the party who generated the parameters is trusted (or the parameters are accompanied by a proof of their correct generation, per Figure 6.4).

6.2.3.4 Computational efficiency improvements

So far, we have presented a bare-bones protocol that achieves our security goals, but it is expensive in terms of bandwidth and computation. The hash function H_G is orders of magnitude slower than a more conventional hash function such as SHA-1. Our goal here is to improve verification performance, so that a downloader can, at the very least, verify hashes as quickly as he can receive them from the network. The bare-bones primitives above imply that a client must essentially recompute the hash of the file $H_G(F)$, but without knowing $\mathbf{r}$.

We use a technique suggested by Bellare, Garay, and Rabin [15] to improve verification performance. Instead of verifying each check block $\mathbf{c}_i$ exactly, we verify them probabilistically and in batches. Each downloader picks a batch size t such as 256 blocks, and a security parameter l such as 32.

The downloader runs a probabilistic batch verifier given by $\mathcal{V}$. The algorithm takes as input the parameter array $(H_G(F'), G, X, C)$. As usual, $H_G(F')$ is the hash of the precoded file F' and G denotes the hash parameters. The $m \times t$ matrix C represents the batch of t check blocks that the downloader received; for convenience, we will write the decomposition $C = (\mathbf{c}_1 \ \cdots \ \mathbf{c}_t)$, where a column $\mathbf{c}_i$ of the matrix represents the i^{th} check block of the batch. The $m \times t$ matrix X is a sparse binary matrix. The cell x_{ij} should be set to 1 if the j^{th} check block contains the message block $\mathbf{b}_i$ and should be 0 otherwise. In other words, the j^{th} column of the matrix X is exactly $\mathbf{x}_j$.

Algorithm $\mathcal{V}(H_G(F'), G, X, C)$

1. Let $s_i \in \{0,1\}^l$ be chosen randomly for $0 < i \leq t$, and let the column vector $\mathbf{s} = (s_1, \ldots, s_t)$.
2. Compute column vector $\mathbf{z} = C\mathbf{s}$.

3. Compute $\gamma_j = \prod_{i=0}^{n'} h_G(\mathbf{b}_i)^{x_{ij}}$ for all $j \in \{1, .., t\}$. Note that if the sender is honest, then $\gamma_j = h_G(\mathbf{c}_j)$.
4. Compute $y' = \prod_{i=1}^{m} g_i^{z_i}$, and $y = \prod_{j=1}^{t} \gamma_j^{s_j}$.
5. Verify that $y' \equiv y \bmod p$.

This algorithm is designed to circumvent the expensive computations of $h_G(\mathbf{c}_i)$ for check blocks in the batch. $\mathcal{V}$ performs an alternative and roughly equivalent computation with the product y in Step 4. The key optimization here is that the exponents s_j are small (l bits) as compared to the much larger λ_q-bit exponents used in Equation 6.1.

Batching does open the receiver to small-scale bandwidth-consumption attacks by a malicious sender: a receiver will accept a batch worth of invalid check blocks before detecting the problem and closing the connection. With our example parameters, each batch is 4 MB. However, a downloader can batch over multiple sources. Only once a batch fails to verify might the downloader attempt per-source batching to determine which source is sending corrupted check blocks (in other words, we can optimize for the common, correct case). Finally, downloaders might tune the batching parameter t based upon their available bandwidth or gradually increase t for each source—based on the assumption that history is a good predictor of behavior—so as to bound its overall fraction of bad blocks.

6.2.3.5 Homomorphic hash trees

As previously noted, hashes $H_G(F)$ are proportional in size to the file F and hence can grow quite large. With our sample hash parameters, an 8 GB file will have a 64 MB hash—a sizable file in and of itself. If a downloader were to use traditional techniques to download such a hash, he would be susceptible to the very same attacks we have set out to thwart, albeit on a smaller scale.

To solve this problem, we construct homomorphic hash trees: treating large hashes themselves as files, and repeatedly hashing until an acceptably small hash is output. We also use a traditional hash function such as SHA-1 to reduce our hashes to standard 20-byte sizes, for convenient indexing at the network and systems levels.

First, pick a parameter L to represent the size of the largest hash that a user might download without the benefit of on-the-fly verification. A reasonable value for L might be 1 MB. Define the following:

$$\begin{aligned}
H_G^0(F) &= F \\
H_G^i(F) &= H_G(H_G^{i-1}(F)) \;\text{ for } i > 0 \\
I_G(F) &= (G, j, H_G^j(F)) \;\text{ for minimal } j \text{ such that } |I_G(F)| < L \\
J_G(F) &= \mathsf{SHA1}(I_G(F))
\end{aligned}$$

That is, $H_G^i(F)$ denotes i recursive applications of H_G. Note that J outputs hashes that are the standard 20 bytes in size. Now, the different components of the system are modified accordingly.

Filename-to-hash mappings. Lookup services map $N(F) \to J_G(F)$, for some G.

File publication. To publish a file F, a publisher must compute the hashes chain of hashes $H_G^1(F), \ldots, H_G^j(F)$, and also the hashes $I_G(F)$ and $J_G(F)$. For $i \in \{0, ..., j-1\}$, the publisher stores $H_G^i(F)$ under the key $(J_G(F), i)$, and he additionally stores $I_G(F)$ under the key $(J_G(F), -)$.

File download. To retrieve a file F, a downloader first performs the name-to-hash lookup $N(F) \to J_G(F)$, for some G. He then uses some key-value indexing layer (§2) to determine a set of sources who serve the file and hashes corresponding to $J_G(F)$. The downloader queries one of the mirrors with the key $(J_G(F), -)$, and expects to receive $I_G(F)$. This transfer can be at most L big. Assuming the hash $J_G(F) = \mathsf{SHA1}(I_G(F))$ correctly verifies, the downloader knows the value j, as well as the j^{th} order hash $H_G^j(F)$. He can then request the next hash in the sequence simultaneously from all of the mirrors who serve F. The downloader queries these servers with the key $(J_G(F), j-1)$, expecting the hash $H_G^{j-1}(F)$ in response. This transfer also can be completed using erasure encoding and our hash verification protocol. The downloader iteratively queries its sources for lower-order hashes until it receives the 0^{th} order hash, or rather, the file itself.

In practice, it would be rare to see a j greater than 3. With our sample hash parameters, the third-order hash of a 512 GB file is a mere 32 KB. However, this scheme can scale to arbitrarily large files. Also note that because each application of the hash function cuts the size of the input by a factor of β/λ_p, the total overhead in hash transmission will be bounded below a small fractional multiple of the original file size, namely:

$$\begin{aligned}\frac{\text{overhead}}{\text{filesize}} &= \sum_{i=1}^{j}\left(\frac{\lambda_p}{\beta}\right)^i < \sum_{i=1}^{\infty}\left(\frac{\lambda_p}{\beta}\right)^i \\ &< \frac{1}{1-\lambda_p/\beta} - 1 = \frac{\lambda_p}{\beta-\lambda_p}\end{aligned}$$

With our example parameters, $\lambda_p/(\beta - \lambda_p) \approx 0.79\%$.

6.3 Evaluation

In this section, we prove the correctness of our protocol, analyze the performance of our hashing scheme, and then prove its security under standard cryptographic assumptions.

6.3.1 Correctness

We now show that homomorphic hashing scheme coupled with the batch verifier given in §6.2.3.4 guarantees correctness. That is, a verifier should always accept the encoded output from an honest sender. Given the proof of this more involved probabilistic verifier, it is easy to see that the naive verifier is also correct.

To prove the batched verification algorithm given in §6.2.3.4 correct—*i.e.,* that correct check blocks will be validated—let us examine an arbitrary hash $(G, H_G(F))$. For notational convenience, we write y and y' computed in Step 4 in terms of an element $g \in \mathbb{Z}_p$ of order q and row vector $\mathbf{r}$ such that $g^{\mathbf{r}} = \mathbf{g} \bmod p$. These elements are guaranteed to exist, even if they cannot be computed efficiently. Thus,

$$y' = \prod_{i=1}^{m} g_i^{z_i} = \prod_{i=1}^{m} g^{r_i z_i} = g^{\sum_{i=1}^{m} z_i r_i} = g^{\mathbf{rz}}$$

By the definition of $\mathbf{z}$ from Step 2, we conclude $y' = g^{\mathbf{r}C\mathbf{s}}$.

Now we examine the other side of the verification, y. Recalling Equation 6.1, rewrite hashes of check blocks in terms of a common generator g:

$$h_G(\mathbf{c}_j) = \prod_{i=1}^{m} g^{r_i c_{i,j}} = g^{\sum_{i=1}^{m} r_i c_{i,j}} = g^{\mathbf{r}\mathbf{c}_j}$$

As noted in Step 3, for an honest sender, $\gamma_j = h_G(\mathbf{c}_j)$. Thus, we can write that $\gamma_j = g^{s_j \mathbf{r}\mathbf{c}_j}$. Combining with the computation of y in Step 4:

$$y = \prod_{j=1}^{t} g^{s_j \mathbf{r}\mathbf{c}_j} = g^{\sum_{j=1}^{t} s_j \mathbf{r}\mathbf{c}_j} = g^{\mathbf{r}C\mathbf{s}}$$

Thus we have that $y' \equiv y \bmod p$, proving the correctness of the validator.

6.3.2 Running-time analysis

In analyzing the running time of our algorithms, we count the number of multiplications over $\mathbb{Z}_p^*$ and $\mathbb{Z}_q$ needed. For instance, a typical exponentiation y^x in $\mathbb{Z}_p^*$ requires $1.5|x|$ multiplications using the "iterative squaring" technique. $|x|$ multiplications are needed to produce a table of values y^{2^z}, for all z such that $1 \leq z < |x|$. Assuming data compression, half of the bits of x on average will be 1, thus requiring $|x|/2$ multiplications of values in the table. In our analysis, we denote $\mathsf{MultCost}(p)$ as the cost of multiplication in $\mathbb{Z}_p^*$, and $\mathsf{MultCost}(q)$ as the cost of multiplication in $\mathbb{Z}_q$.

Note that computations of the form $\prod_{i=1}^{m} g_i^{x_i}$ are computed at various stages of the different hashing protocols. As mentioned above, the precomputation of the $g_i^{2^z}$ requires $m\lambda_q$ multiplications over $\mathbb{Z}_p^*$. But the product itself can be computed in $(m\lambda_q/2)\,\mathsf{MultCost}(p)$ computations—and not $(m\lambda_q/2 + m - 1)\,\mathsf{MultCost}(p)$ as one might expect—by keeping a "running product."

We recognize that certain operations like modular squaring are cheaper than generic modular multiplication. Likewise, multiplying an element of $\mathbb{Z}_q$ by a 32-bit number is less expensive than multiplying two random elements from $\mathbb{Z}_q$. In our analysis, we disregard these optimizations and seek only simplified upper bounds.

Per-publisher hash generation. Publishers first precompute a table g^{2^z} for all z such that $1 \leq z < \lambda_q$. This table can then be used to compute $H_G(F)$ for any file F. Here and throughout this analysis, we can disregard the one-time precomputation, since $n \gg m$. Thus, the n-vector exponentiation in Equation 6.4 requires an expected $n\lambda_q/2$ multiplications in $\mathbb{Z}_p^*$. To compute $\mathbf{r}F$ as in Equation 6.4, mn multiplications are needed in $\mathbb{Z}_q$. The total cost is therefore $mn\,\mathsf{MultCost}(q) + n\lambda_q\,\mathsf{MultCost}(p)/2$.

Global hash generation. Publishers using the global hashing scheme do not know $\mathbf{r}$ and hence must do multiple exponentiations per block. That is, they must explicitly compute the product given in Equation 6.1, with only the benefit of the precomputed squares of the g_i. If we ignore these precomputation costs, global hash generation requires a total of $nm\lambda_q\,\mathsf{MultCost}(p)/2$ worth of computation.

Naive hash verification. Hash verifiers who chose not to gain batching speed-ups perform much the same operations as the global hash generators. That is, they first precompute tables of squares, and then compute the left side of Equation 6.3 for the familiar cost of $m\lambda_q\,\mathsf{MultCost}(p)/2$. The right side of the equation necessitates an average of d multiplications in $\mathbb{Z}_p^*$, where d, we recall, is the average degree of a check block $\mathbf{c}$. Thus, the expected per-block cost is $(m\lambda_q/2 + d)\mathsf{MultCost}(p)$.

Fast hash verification. We refer to the algorithm described in §6.2.3.4. In Step 2, recall that C is a $m \times t$ matrix, and hence the matrix multiplication costs $mt\,\mathsf{MultCost}(q)$. $\mathcal{V}$ determines γ_j in Step 3 with d multiplications over $\mathbb{Z}_p^*$, at a total cost of $td\,\mathsf{MultCost}(p)$. In Step 4, computing y' costs $m\lambda_q/2\,\mathsf{MultCost}(p)$ with access to precomputed tables of the form $g_i^{2^x}$. For y, no such precomputations exist; the bases in this case are γ_j, of which there are more than n. To compute y efficiently, we use the fast multiplication algorithm presented in [15] (and discussed in [104]),

Operation on 16 KB block $\mathbf{b}$	time (ms)	throughput (MB/s)
Per-publisher computation of $h_G(\mathbf{b})$	1.39	11.21
Global computation of $h_G(\mathbf{b})$	420.90	0.037
Naive verification of $h_G(\mathbf{b})$	431.82	0.038
Batched verification of $h_G(\mathbf{b})$	2.05	7.62
$\mathsf{SHA1}(\mathbf{b})$	0.28	56.25
Sign $\mathbf{b}$ with Rabin-1024	1.98	7.89
Verify Rabin-1024 signature of $\mathbf{b}$	0.29	53.88
Receiving $\mathbf{b}$ on a T1	83.33	0.186
Reading $\mathbf{b}$ from disk (sequentially)	0.27	57.87

Figure 6.5: Homomorphic hashing microbenchmarks

which costs $(tl/2 + l - 1)\,\mathsf{MultCost}(p)$.[2] Summing these computations and amortizing over the batch size t yields a per-block cost of:

$$m \cdot \mathsf{MultCost}(q) + \left[d + \frac{l}{2} + \frac{m\lambda_q/2 + l - 1}{t}\right] \cdot \mathsf{MultCost}(p)$$

6.3.2.1 Microbenchmarks

We implemented a version of these hash primitives using the GNU MP library, version 4.1.2. Figure 6.5 shows the results of our C++ testing program when run on a 3.0 GHz Pentium 4, with the sample parameters given in Figure 6.3 and the batching parameters given in §6.2.3.4. On this machine, $\mathsf{MultCost}(p) \approx 6.2\ \mu$secs and $\mathsf{MultCost}(q) \approx 1.0\ \mu$secs. Our results are reported in both cost per block and overall throughput. For comparison, we include similar computations for SHA-1 and for the Rabin signature scheme with 1024-bit keys [148]. We also include disk bandwidth measurements for reading blocks off a Seagate 15K Cheetah SCSI disk drive (in batches of 64), and maximum theoretical packet arrival rate on a T1 connection. We will call on these benchmarks in the next section.

[2]This fast multiplication algorithm offers no per-block performance improvement for naive verification, thus we only consider it for fast verification.

Although batched verification of hashes is almost an order of magnitude slower than a more conventional hash function such as SHA-1, it is still more than an order of magnitude faster than the maximum packet arrival rate on a good Internet connection. Furthermore, by adjusting the batch parameter t, downloaders can upper-bound the amount of *time* they waste receiving bad check blocks. That is, receivers with faster connections can afford to download more potentially bogus check blocks, and can therefore increase t (and thus verification throughput) accordingly.

Our current scheme for global hash generation is rather slow, but publishers with reasonable amounts of RAM can use k-ary exponentiation to achieve a four-fold speedup (see [104]). Our performance analysis focuses on the per-publisher scheme, which we believe to be better-suited for copyright-friendly distribution of bulk data.

6.3.2.2 Performance Comparison

In §6.2.2.1, we discussed other strategies for on-the-fly verification of check blocks in LF-CDNs. We now describe these proposals in greater detail, to examine how our scheme compares in terms of bandwidth, storage, and computational requirements. There are three schemes in particular to consider:

High-degree SHA-1 hash tree. The publisher generates n/r check blocks, and then hashes each one. Since this collection of hashes might be quite large, the publisher uses the recursive scheme described in §6.2.3.5 to reduce it to a manageable size. The publisher distributes the file, keyed by the root of the hash tree. Downloaders first retrieve all nodes in the hash tree and then can verify check blocks as they arrive.

Binary SHA-1 hash tree. As before, the publisher generates n/r check blocks, but then computes a binary hash tree over all check blocks. The publisher keys the file by the root of its hash tree. In this scheme, mirrors need access to the entire hash tree, but clients do not. Rather, when the mirrors send check blocks, they prepend the "authentication path" describing the particular check block's location in the hash tree. If downloaders know the hash tree's root *a priori*, they can, given the correct authentication path, verify that a received check block is one of those intended by the publisher.

Sign every block. A publisher might generate n/r blocks and simply sign every one. The hash of the file is then the SHA-1 of the filename and the publisher's public key. The mirrors must download and store these signatures, prepending them to check blocks before they are sent to remote clients. To retrieve the file, clients first obtain the publisher's public key from the network, and verify this key against the hash of the file. When they arrive from mirrors, the check blocks contain their own signatures and are thus easily verified.

These three schemes require a suitable value of r. For codes with rate r, a file with n message blocks will be expanded into n/r check blocks. For simple lower bounds, assume that any set of n of these check blocks suffices to reconstruct the file. In a multicast scenario, a client essentially collects these blocks at random, and the well-known "coupon collector bound" predicts that he will receive $-(n/r)\ln(1-r)$ check blocks on average before collecting n unique check blocks.[3] Using this bound, we can estimate the expected additional transmission overheads due to repeated check blocks:

r	$-(1/r)\ln(1-r)$
1/2	0.3863
1/4	0.1507
1/8	0.0683
1/16	0.0326
1/32	0.0160

That is, with an encoding rate $r = 1/2$, a receiver expects an additional 39% overhead corresponding to duplicate blocks. In many-to-many transmission schemes, small encoding rates are essential to achieving good bandwidth utilization.

We now present a performance comparison of the three fixed-rate schemes and our homomorphic hashing proposal, focusing on key differences between them: hash generation costs incurred

[3]This asymptotic bound is within a 10^{-5} neighborhood of the exact probability when $n = 2^{16}$.

by the publisher, storage requirements at the mirror, bandwidth utilization between the mirror and downloader, and verification performance.

Hash generation. Fixed-rate schemes such as the three presented above can generate signatures only as fast as they can generate check blocks. Encoding performance depends upon the file's size, but because we wish to generalize our results to very large files, we must assume that the publisher cannot store the entire input file (or output encoding) in main memory. Hence, secondary storage is required.

Our implementation experience with Online Codes has shown that the encoder works most efficiently if it stores the relevant pieces of the encoding graph structure and a fixed number of check blocks in main memory.[4] The encoder can make several sequential passes through the file. With each pass, it adds message blocks from disk into check blocks in memory, as determined by the encoding graph. As the pass is completed, it flushes the completed batch of check blocks to the network, to disk, or to functions that compute hashes or signatures. This technique exploits the fact that sequential reads from disk are much faster than random seeks.

Our current implementation of Online Codes can achieve encoding throughputs of about 21 MB/s (on 1 GB files, using roughly 512 MB of memory). However, to better compare our system against fixed-rate schemes, we will assume that an encoder exists that can achieve the maximum possible throughput. This upper bound is $ae/(\beta n)$, where the file has n blocks, the block size is β, the amount of memory available for storing check blocks is a, and the disk's sequential read throughput is e.

When publishers use fixed-rate schemes to generate hashes, they must first precompute n/r check blocks. Using the encoder described above, this computation requires $n\beta/(ra)$ scans of the entire file. Moreover, each scan of the file involves n block reads, so $n^2\beta/(ra)$ block reads in total are required. Concurrent with these disk reads, the publisher computes hashes and signatures of the check blocks and the implied hash trees if necessary.

The theoretical requirements for all four schemes are summarized in Figure 6.6. In the bottom table, we have attempted to provide some concrete realizations of our theoretical bounds. Through-

[4]With little impact on performance, our implementation also stores auxiliary blocks in memory.

Scheme	Block Reads	DLog Hashes	SHA-1 Hashes	Sigs
Homomorphic Hashing	n	$n\beta/(\beta - \lambda_p)$	1	1
Big-Deg. SHA-1 Tree	$n^2\beta/(ra)$	0	$(n/r)\beta/(\beta - 160)$	1
Binary SHA-1 Tree	$n^2\beta/(ra)$	0	$2n/r$	1
Sign Every Block	$n^2\beta/(ra)$	0	0	n/r

Scheme	Disk (sec)	CPU (sec)	Lower Bound (sec)
Homomorphic Hashing	17.69	91.81	91.81
Big-Deg. SHA-1 Tree	566.23	293.96	566.23
Binary SHA-1 Tree	566.23	587.20	587.20
Sign Every Block	566.23	2076.18	2076.18

Figure 6.6: Comparison of hash generation performance

out, we assume (1) a 1 GB file, broken up into $n = 2^{16}$ blocks, each of size $\beta = 16$ KB, (2) the publisher has $a = 512$ MB of memory for temporary storage of check blocks, and (3) disk throughputs can be sustained at 57.87 MB/s as we observed on our test machine. Under these conditions, an encoder can achieve theoretical encoding throughputs of up to 28.9 MB/s. We further assume that (4), seeking to keep overhead due to redundant check blocks below 5%, the publisher uses an encoding rate of $r = 1/16$, and (5) a publisher can entirely overlap disk I/O and computations and therefore only cares about whichever takes longer. In the right-most column, we present reasonable lower bounds on hash generation performance for the four different schemes.

Despite our best efforts to envision a very fast encoder, the results for the three fixed-rate schemes are poor, largely due to the cost of encoding n/r file blocks. Moreover, in the sign-every-block scheme, the CPU becomes the bottleneck due to the expense of signature computation.

By contrast, the homomorphic hashing scheme can escape excessive disk accesses, because it hashes data before it is encoded. It therefore requires only one scan of the input file to generate the hashes of the message blocks. The publisher's subsequent computation of the higher-level hashes $H^2(F), H^3(F), \ldots$ easily fit into memory. Our prototype can compute a homomorphic hash of a 1 GB file in 123.63 seconds, reasonably close to the lower bound of 91.81 seconds predicted in Figure 6.6.

Of course, performance for the three fixed-rate schemes worsens as r becomes smaller or n

Scheme	Overhead	Storage (MB)
Homomorphic Hash	0.008	8.06
Big-Degree Tree	0.020	20.02
Binary Tree	0.039	40.00
Sign Every Block	0.125	128.00

Figure 6.7: Comparison of additional storage requirements for mirrors

becomes larger. It is possible to ameliorate these problems by raising the block size β or by striping the file into several different files, but these schemes involve various trade-offs that are beyond the scope of our consideration.

Mirror's encoding performance. In theory, the homomorphic hashing scheme renders encoding more computationally expensive because it substitutes XOR block addition for more expensive modular additions. We have measured that our machine computes the exclusive OR of two 16 KB check blocks in 8.5 μsecs. By comparison, our machine requires 37.4 μsecs to sum two blocks with modular arithmetic. The average check-block degree in our implementation of Online Codes is 8.17, so check-block generation on average requires 69.5 μsecs and 305 μsecs under the two types of addition. This translates to CPU-bound throughputs of 224.8 MB/s and 51.3 MB/s, respectively. However, recall that disk throughput and memory limitations combine to bound encoding for both schemes at only 28.9 MB/s. Moreover, these throughputs are orders of magnitude larger than typical upstream network capacities in peer-to-peer deployments.

Storage required on the mirror. Mirrors participating in LF-CDNs donate disk space for content distribution, though usually they mirror files they also use themselves. All four verification schemes require additional storage for hashes and signatures. With homomorphic hashing, the mirror should store the hash that the publisher provides. Regenerating the hash is theoretically possible but computationally expensive. Similarly, mirrors in the two SHA-1 hash-tree schemes should retrieve complete hash trees from the publisher and store them to disk, or they must otherwise dedicate tremendous amounts of disk I/O to generate them on-the-fly. Finally, in the sign-every-block scheme, the mirror does not know the publisher's private key and hence cannot generate signatures.

Scheme	Up-Front Predicted (bits)	Up-Front *e.g.,* (MB)	Per-Block Predicted (bits)	Per-Block *e.g.,* (KB)
Homomorphic Hashing	$\lambda_p n\beta/(\beta-\lambda_p)$	8.06	$\beta+m$	16.06
Big-Deg. SHA-1 Tree	$160n\beta/(\beta-160)$	20.02	β	16.00
Binary SHA-1 Tree	0	0	$\beta+160\log_2(n/r)$	16.39
Sign Every Block	0	0	$\beta+\lambda_\sigma$	16.13

Scheme	Total (GB)	w/ Penalty (GB)
Homomorphic Hashing	1.0118	1.0118
Big-Deg. SHA-1 Tree	1.0196	1.0528
Binary SHA-1 Tree	1.0244	1.0578
Sign Every Block	1.0078	1.0407

Figure 6.8: Comparison of bandwidth utilization

He has no choice but to store all signatures. We summarize these additional storage requirements in Figure 6.7, again assuming a 1 GB input file and an encoding rate of $r=1/16$.

Bandwidth. The bandwidth requirements of the various schemes are given in terms of up-front and per-block costs. These results are considered in Figure 6.8. The new parameter λ_σ describes the size of signatures, which is 1024 bits in our examples. In multicast settings, receivers of fixed-rate codes incur additional overhead due to repeated blocks (reported as "penalty"). At an encoding rate of $r=1/16$, the coupon collector bound predicts about 3.3% overhead. In all four schemes, downloaders also might see duplicate blocks when reconciling partial transfers with other downloaders. That is, if two downloaders participate in the same multicast tree, and then try to exchange check blocks with each other, they will have many blocks in common. This unavoidable (and deployment-specific) problem affects all four schemes equally, and we thus do not include it in our analysis. It can be mitigated, however, by general set-reconciliation algorithms [22] or protocols specific to peer-to-peer settings [117].

The binary SHA-1 tree and the sign-every-block scheme allow downloaders to retrieve a file without the up-front transfer of cryptographic meta-data. Of course, when downloaders become full mirrors, they cannot avoid this cost. In the former scheme, the downloader needs the hash tree in its entirety, adding an additional 3.9% overhead to its total transfer. In the latter, the downloader

Scheme	Batch	SHA-1	Rabin	Total (ms)
Homomorphic Hash	1	0	0	2.05
Big-Degree Tree	0	1	0	0.28
Binary Tree	0	$\log_2(n/r)$	0	5.60
Sign Every Block	0	0	1	0.29

Figure 6.9: Comparison of per-block verification performance

requests all those signatures not already received. This translates to roughly 11.7% additional overhead when $r = 1/16$.

Verification. Figure 6.9 summarizes the per-block verification costs of the four schemes. For our homomorphic hashing scheme, we assume batched verification with the parameters given in §6.2.3.4. The Rabin signature scheme was specially chosen due to its fast verification time, as shown. Surprisingly, verifying a check block using a SHA-1 binary tree is more than twice as slow as using our homomorphic hashing protocol, due to the height of the tree.

6.3.2.3 Discussion

For encoding rates such as $r = 1/16$, each of the three fixed-rate schemes has important strengths and weaknesses. Though the sign-every-block scheme is bandwidth-efficient, requires no up-front hash transfer, and has good verification performance, its hash generation costs are prohibitive and its storage costs are higher. Similarly, though the binary hash tree method has no up-front transfer, its bandwidth, storage and verification costs make it less attractive than hash trees with larger fan-out. The homomorphic hashing scheme entails no such trade-offs, as it performs well across all categories considered. Homomorphic hashing ranks less favorably when considering verification throughput, but as argued in §6.3.2.1, tuning batch size allows throughput to scale with available bandwidth.

6.3.3 Security

In real-world LF-CDNs, an honest receiver who wishes to obtain the file F communicates almost exclusively with untrusted parties. As mentioned in §6.2.2, we do not consider the problem of mapping file names to file hashes; in our analysis, we assume that the receiver can reliably resolve $N(F) \to J_G(F)$ through a trusted, out-of-band channel. We wish to prove, however, that given $J_G(F)$, the downloader can recover F from the network while recognizing certain types of dishonest behavior almost immediately.

6.3.3.1 Collision-resistant hash functions

First, we formally define a collision-resistant hash function (CRHF) in the manner of [13]. Recall that a family of hash functions is given by a pair of PPT algorithms $\mathcal{F} = (\mathsf{HGen}, \mathcal{H})$. HGen denotes a hash generator function, taking an input of security parameters $(\lambda_p, \lambda_q, m)$ and outputting a description of a member of the hash family, G. $\mathcal{H}_G$ will hash inputs of size $m\lambda_q$ to outputs of size λ_p, exactly as we have seen thus far. A hash adversary $\mathcal{A}$ is a probabilistic algorithm that attempts to find collisions for the given function family.

Definition 1 *For any CRHF family $\mathcal{F}$, any probabilistic algorithm $\mathcal{A}$, and security parameter $\lambda = (\lambda_p, \lambda_q, m)$ where $\lambda_q < \lambda_p$ and $m \leq \mathrm{poly}(\lambda_p)$, let*

$$\begin{aligned} \mathsf{Adv}^{\text{col-atk}}_{\mathcal{F},\lambda}(\mathcal{A}) \quad = \quad \Pr[\ \ & G \leftarrow \mathsf{HGen}(\lambda); (x_1, x_2) \leftarrow \mathcal{A}(G) : \\ & \mathcal{H}_G(x_1) = \mathcal{H}_G(x_2) \ \wedge \ x_1 \neq x_2 \quad] \end{aligned}$$

$\mathcal{F}$ is a (τ, ε)-secure hash function family if, for all PPT adversaries $\mathcal{A}$ with time-complexity $\tau(\lambda)$, $\mathsf{Adv}^{\textit{col-atk}}_{\mathcal{F},\lambda}(\mathcal{A}) < \varepsilon(\lambda)$, where $\varepsilon(\lambda)$ is negligible in λ and $\tau(\lambda)$ is polynomial in λ.

Our definition of the hash primitive h per §6.2.3 fits naturally into this definition. In fact, the $\mathsf{PickGroup}$ algorithm is a reasonable candidate for the function $\mathsf{HGen}()$. See [13] for a proof that the function family h_G is collision-resistant hash function, assuming that the discrete log problem is hard over the group parameterized by (λ_p, λ_q).

6.3.3.2 Security of encoding verifiers

We can now define a notion of security against bogus-encoding attacks. For simplicity, we assume erasure codes that have a precoding algorithm $\mathcal{P}$ and an encoder amenable to succinct matrix representation; as discussed in §6.2.1, examples include LT, Raptor, and Online Codes.

As usual, consider an adversary $\mathcal{A}$ against an honest verifier $\mathcal{V}$. The adversary $\mathcal{A}$ succeeds in a bogus-encoding attack if he can convince the verifier $\mathcal{V}$ to accept blocks from "forged" or bogus file encodings. When making the decision of whether or not to accept a given encoding, $\mathcal{V}$ can only access the hash $H_G(F')$ of the precoded file F' he expects. In this definition, the adversary has the power to generate the file F, which is then precoded as normal to obtain F'.

Definition 2 (Secure Encoding Verifier) *For any CRHF $\mathcal{H}$, any probabilistic algorithm $\mathcal{A}$, any honest verifier $\mathcal{V}$, any $m, n > 0$, and any batch size $t > 1$, let:*

$$
\begin{aligned}
\mathsf{Adv}^{\text{enc-atk}}_{\mathcal{H},\mathcal{V},m,n,t}(\mathcal{A}) \quad = \quad \Pr[\quad & G \leftarrow \mathsf{HGen}(\mathcal{H});\ (F, X, C) \leftarrow \mathcal{A}(G, m, n, t); \\
& F' \leftarrow \mathcal{P}(F);\ b \leftarrow \mathcal{V}(\mathcal{H}_G(F'), G, X, C) : \\
& \quad F \textit{ is } m \times n \ \wedge\ F' \textit{ is } m \times n' \ \wedge\ X \textit{ is } n' \times t \\
& \quad \wedge\, C \textit{ is } m \times t \ \wedge\ F'X \neq C \ \wedge\ b = \mathsf{Accept} \quad]
\end{aligned}
$$

The encoding verifier $\mathcal{V}$ is (τ, ε)-secure if, $\forall\ m, n > 0,\ t > 1$ and PPT *adversaries $\mathcal{A}$ with time-complexity $\tau(m, n, t)$, $\mathsf{Adv}^{\text{enc-atk}}_{\mathcal{H},\mathcal{V},m,n,t}(\mathcal{A}) < \varepsilon(m, n, t)$.*

Our definition requires that $\mathcal{V}$ be (τ, ε)-secure for all values of $t > 1$. Thus, a protocol that uses a secure encoding verifier can tune t as desired to trade computational efficiency for communication overhead. From here, we can prove that the batch verification procedure presented previously is secure.

Theorem 1 *Given security parameters l, λ_p, λ_q, batch size t, number of generators m, and the (τ, ε)-secure hash family h generated by $(\lambda_q, \lambda_p, m)$, the batched verification procedure $\mathcal{V}$ given above is a (τ', ε')-secure encoding verifier, where $\tau' = \tau - mt(\mathsf{MultCost}(q) + \mathsf{MultCost}(p))$ and $\varepsilon' = \varepsilon + 2^{-l}$.*

We do not state or prove the corresponding theorem for the naive verifier, but it is straightforward to check that it has equivalent or stronger properties than that of the batch verifier. The security of the recursive hashing scheme outlined in §6.2.3.5 follows from an inductive application of Theorem 1.

Proof of secure encoding verifier. We now prove the security of the batched verification scheme by proving Theorem 1. Our proof follows from that of [15], with some additional complexity due to our multi-dimensional representation of a file.

Consider the hash function family h parameterized by $(\lambda_p, \lambda_q, m)$. For any file size n and batch size $t < n$, consider an arbitrary adversary $\mathcal{A}'$ that (τ', ε')-attacks the encoding verifier $\mathcal{V}$. Based on this adversary, define a CRHF-adversary $\mathcal{A}(G)$ that works as follows:

Algorithm $\mathcal{A}(G)$

1. $(F, X, C) \leftarrow \mathcal{A}'(G, m, n, t)$.
2. If F is not $m \times n$ or X is not $n' \times t$ or C is not $m \times t$, then Fail.
3. $F' \leftarrow \mathcal{P}(F)$.
4. If $F'X = C$, then Fail.
5. If $\mathcal{V}(H_G(F'), G, X, C) =$ Reject, then Fail.
6. If $H_G(F'X) \neq H_G(C)$, then Fail.
7. Find a column j such that $F'\mathbf{x}_j \neq \mathbf{c}_j$. Return $(F'\mathbf{x}_j, \mathbf{c}_j)$.

By our selection of the adversary $\mathcal{A}'$, running it in Step 1 will require time complexity τ' and will succeed in the experiment given in Definition 2 with probability ε'. By construction, $\mathcal{A}$ corresponds naturally to the steps of our definitional experiment in Equation 6.5. Step 2 enforces appropriate dimensionality. Step 4 enforces the requirements that $\langle X, C \rangle$ not be a legal encoding, given in Equation 6.5 by $F'X \neq C$. Step 5 requires that the verifier $\mathcal{V}$ accepts the "forged" input. We can conclude that the Algorithm $\mathcal{A}$ will arrive at Step 6 with probability ε'.

We now argue that $\mathcal{A}$ fails at Step 6 with probability 2^{-l}. To arrive at this step, the verifier $\mathcal{V}$ as defined in §6.2.3.4 must have output Accept. Using the same manipulations as those in §6.3.1, we take the fact that $\mathcal{V}$ accepted to mean that:

$$g^{\mathbf{r}F'X\mathbf{s}} \equiv g^{\mathbf{r}C\mathbf{s}} \bmod p$$

Note that the exponents on both sides of the equation are scalars. Because g has order q, we can say that these exponents are equivalent $\bmod q$; that is, $\mathbf{r}F'X\mathbf{s} \equiv \mathbf{r}C\mathbf{s} \bmod q$., Rearranging, we have that:

$$\mathbf{r}\left(F'X - C\right)\mathbf{s} \equiv 0 \bmod q \tag{6.5}$$

If the algorithm $\mathcal{A}'$ fails at Step 6, then $H_G(F'X) \neq H_G(C)$. Rewriting these row vectors in terms of the g and $\mathbf{r}$, we have that $g^{\mathbf{r}F'X} \not\equiv g^{\mathbf{r}C} \bmod p$. Recalling that g is order q and that exponentiation of a scalar by a row vector is defined component-wise, we can write that $\mathbf{r}F'X \not\equiv \mathbf{r}C \bmod q$, and consequently:

$$\mathbf{r}\left(F'X - C\right) \not\equiv \mathbf{0} \bmod q \tag{6.6}$$

For convenience, let the $1 \times t$ row vector $\mathbf{u} = \mathbf{r}(F'X - C)$. Equation 6.6 gives us that $\mathbf{u} \not\equiv \mathbf{0} \bmod q$; thus some element of $\mathbf{u}$ must be non-zero. For simplicity of notation, say that u_1 is the first non-zero cell, but our analysis would hold for any index. Equation 6.5 gives us that $\mathbf{us} \equiv \mathbf{0} \bmod q$. Since $u_1 \neq 0$, it has a multiplicative inverse, u_1^{-1}, in $\mathbb{Z}_q^*$. Therefore:

$$s_1 \equiv -\left(u_1^{-1}\right)\sum_{j=2}^{t} u_j s_j \bmod q \tag{6.7}$$

Referring to Step 1 of verifier $\mathcal{V}$, s_1 was selected at random from 2^l possible values; consequently, the probability of it having the particular value in Equation 6.7 is at most 2^{-l}. Thus, $\mathcal{A}$ can fail at Step 6 with probability at most 2^{-l}.

Combining our results, we have that algorithm $\mathcal{A}$ will reach Step 7 with probability $\varepsilon' - 2^{-l}$. At this point in the algorithm, $\mathcal{A}$ is assured that $F'X \neq C$, since execution passed Step 4. If we consider this inequality column-wise, we conclude there must be some $j \in \{1, ..., t\}$ such that $F'\mathbf{x}_j \neq \mathbf{c}_j$, where $\mathbf{x}_j$ and $\mathbf{c}_j$ are the j^{th} columns of X and C, respectively. Because Step 6 guarantees that $H_G(F'X) = H_G(C)$ at this point in the algorithm, we can use the definition of H_G to claim that for all j, $h_G(F'\mathbf{x}_j) = h_G(\mathbf{c}_j)$. Thus, $(F'\mathbf{x}_j, \mathbf{c}_j)$ represents a hash collision for the hash function h_G.

Analyzing the time-complexity of $\mathcal{A}$, Step 1 completes with time-complexity τ', the matrix multiplication $F'X$ in Step 4 requires mt multiplications in $\mathbb{Z}_q$, and the hash computations in Step 6 each require $tm/2$ multiplications in $\mathbb{Z}_p^*$, assuming the usual precomputations. Therefore, $\mathcal{A}$ has a time complexity given by $\tau = \tau' + mt(\mathsf{MultCost}(q) + \mathsf{MultCost}(p))$.

Therefore, we have shown that if an adversary $\mathcal{A}'$ exists that is successful in a (τ', ε')-attack against $\mathcal{V}$, then another adversary $\mathcal{A}$ exists that is (τ, ε)-successful in finding collisions for the hash function h, where $\tau' = \tau - mt(\mathsf{MultCost}(q) + \mathsf{MultCost}(p))$ and $\varepsilon = \varepsilon' + 2^{-l}$. This completes the proof of Theorem 1. ∎

6.3.3.3 Discussion of end-to-end security

These security guarantees, while necessary, are not sufficient for all multicast settings. In §6.2.2, we proposed both the bogus-encoding attack and the distribution attack. While we have solved the former, one can imagine malicious encoders that thwart the decoding process through an incorrect distribution of well-formed check blocks. Because Tornado, Raptor, Online, and LT Codes are all based on irregular graphs, their output symbols are not interchangeable. Bad encoders could corrupt degree distributions; they could also purposefully avoid outputting check blocks derived from some particular set of message blocks. Indeed, the homomorphic hashing scheme and the three fixed-rate schemes discussed in §6.3.2.2 are all vulnerable to the distribution attack.

In future work, we hope to satisfy a truly end-to-end definition of security for encoding schemes. For the end-to-end model, we envision an experiment in which the adversary can chose to supply the recipient with either its own check blocks or those from an honest encoder. The X and C parameters of the verifier function now correspond to the entire download history, not just to the most recent batch of blocks. The verifier outputs Reject if it believes it is talking to a malicious encoder, in which case the experiment discards the batch of blocks just received. In the end, the experiment runs the decoder on all retained check blocks after receiving $(1+\epsilon)n' + aB$ total blocks, where a is a constant allowance for wasted bandwidth per bad encoder, and B is the number of times the verifier correctly output Reject after receiving blocks from the adversary. The adversary succeeds if this decoding fails with non-negligible probability.

One approach toward satisfying such a definition might be to require a sender to commit to a pseudo-random sequence determined by a succinct seed, and then to send check blocks whose x_i portions are entirely determined by the pseudo-random sequence. But in the context of non-reliable network transport or multicast "downsampling," a malicious sender can drop particular blocks in the sequence and place the blame on congestion. If, for example, the sender drops all degree-one blocks, or drops all check blocks that mention a particular message block, decoding will never succeed.

A more promising approach involves validating an existing set of check blocks by simulating the receipt of future check blocks. Given an existing set of check blocks:

$$\langle \mathbf{x}_1, \mathbf{c}_1 \rangle, \langle \mathbf{x}_2, \mathbf{c}_2 \rangle, \ldots, \langle \mathbf{x}_Q, \mathbf{c}_Q \rangle$$

the verifier can run the encoder (without the contents of F) to generate a stream of block descriptions $\mathbf{x}_{Q+1}, \mathbf{x}_{Q+2}, \ldots$. If the file would not be recoverable given $\mathbf{c}_{Q+1}, \mathbf{c}_{Q+2}, \ldots$, this is evidence that the distribution of $\mathbf{x}_1, \ldots, \mathbf{x}_Q$ has been skewed. If the file would be recoverable, the verifier can repeat the experiment several times to amplify its confidence in $\mathbf{x}_1, \ldots, \mathbf{x}_Q$. To be of any use, such a verifier can do no more than $O(\log n)$ operations per check block received. Thus simulated streams should be re-used for efficiency, with the effects of the first simulated block $\mathbf{x}_{Q+1}$ replaced by those of the next real block received. The feasibility of efficiently "undoing" encoding remains an open question; we therefore leave the description and analysis of an exact algorithm to future work.

6.4 Related work on multicast security

Multicast source authentication is a well-studied problem in the recent literature; for a taxonomy of security concerns and some schemes, see [25]. Preexisting solutions fall into two broad categories: (1) sharing secret keys among all participants and MACing each block, or (2) using asymmetric cryptography to authenticate each block sent. Unfortunately, the former lacks any source authentication, while the latter is costly with respect to both computation resources and bandwidth.

A number of papers have looked at providing source authentication via public-key cryptography, yet amortizing asymmetric operations over several blocks. Gennaro and Rohatgi [66] propose a protocol for stream signatures, which follows an initial public-key signature with a chain of efficient one-time signatures, although it does not handle block erasures (*e.g.,* from packet loss). Wong and Lam [192] delay consecutive packets into a pool, then form an authentication hash and sign the tree's root. Rohatgi [159] uses reduced-size online/offline k-time signatures instead of hashes. Tree-based [70] and graph-based [122] approaches reduce the time/space overheads and are designed for bursty communication and random packet loss. More recent work [135, 136] makes use of *trusted* erasure encoding in order to authenticate blocks; most schemes, including our own, try instead to authenticate blocks in spite of *untrusted* erasure encoding.

Another body of work is based solely on symmetric-key operations or hash functions for real-time applications. Several protocols use the delayed disclosure of symmetric keys to provide source authentication, including Chueng [38], the Guy Fawkes protocol [6], and TESLA [141, 142], by relying on loose time synchronization between senders and recipients. The BiBa [140] protocol exploits the birthday paradox to generate one-time signatures from k-wise hash collisions. The latter two can withstand arbitrary packet loss; indeed, they were explicitly developed for Digital Fountain's content distribution system [20, 21] to support video-on-demand and other similar applications. Unfortunately, these delayed-disclosure key schemes require that publishers remain online during transmission.

In the traditional settings considered above, the publisher and the encoder are one in the same. In our LF-CDN setting, untrusted mirrors generate the check blocks; moreover, a trusted publisher cannot explicitly authenticate every possible check block, since their number grows exponentially with file size. Thus, a publisher must generate its authentication tokens on the initial message blocks, and we require a hash function that preserves the group structure of the encoding process.

Our basic homomorphic hashing scheme is complementary to existing threads of work that make use of homomorphic group operations. One-way accumulators [11, 16] and incremental hashing [13], based on RSA and discrete-log constructions respectively, examine commutative hash functions that yield an output independent of the operations' order. Improvements to the schemes' efficiency [12, 24, 183], however, largely focus on dynamic or incremental changes to the elements being hashed/authenticated, *e.g.,* the modification of an entry of an authenticated dictionary. More recent work has investigated homomorphic signature schemes for specific applications: undirected transitive signatures [121], authenticated prefix aggregation [34], redactable signatures [91], and set-union signatures via accumulators [91]. We use similar techniques to maintain group structure across applications of the cryptographic function, but to different ends. Composing homomorphic signatures with traditional hash functions such as SHA-1 [55] would not solve our problem, as the application of the traditional hash function would destroy the group structure we hope to preserve.

Chapter 7

Conclusion

7.1 Summary of contributions

This thesis contributes a number of techniques that realize highly-scalable cooperative content distribution by federating large numbers of participants. These mechanisms have been designed, implemented, deployed, and tested in production systems that have served billions of requests from many millions of users over the past three-and-a-half years. More specifically, this thesis has presented and evaluated the following systems and techniques.

- For effective *content discovery*, the Coral indexing layer provides a soft-state key-value store, offering good routing and data locality through a hierarchical structure and self-organizing clusters, yet preventing any hot spots in the indexing layer, even under degenerate loads.

- CoralCDN builds a web content distribution network, using Coral to manage the referencing and discovery of cached content. CoralCDN incorporates a number of mechanisms, learned over its deployment, to provide efficient downloads, robustness to origin failures, bandwidth fairness, and security. We have made CoralCDN freely available (see `http://www.coralcdn.org/`), so that even people with slow connections can publish websites whose capacity grows automatically with popularity.

- For *server selection*, OASIS provides a shared locality- and load-aware anycast infrastructure. OASIS shows how to amortize deployment and network measurement costs over all participants, determine and maintain network locality information in an application-independent way, and manage all tasks and data in a decentralized manner through a key-value index.
- Investigating new approaches for secure cooperation, Shark shows how to scale a distributed read/write file system by utilizing the bandwidth of mutually-distrustful peers. Rather than always fetch data from an origin file server, nodes can locate (via the Coral indexing layer) and read chunks of cached data from each other, even while ensuring traditional data integrity and read authorization properties.
- Finally, also taking a more clean-slate approach for *secure content transmission*, we show how to securely leverage rateless erasure codes when downloading large files. We present a discrete-log-based hash scheme that provides useful homomorphic properties for verifying the integrity of individually-downloaded blocks, while batching techniques make this scheme practical for real-world use.

We have found these mechanisms and systems to be quite useful for building distributed services: Several CDNs employ the Coral indexing layer [8, 59, 73], OASIS is used by about a dozen distributed services for anycast [40, 59, 73, 92, 100, 157, 163, 179, 193], and CoralCDN is in extensive public use. We fully envision that these tools may find similar applicability to a wider range of distributed systems.

7.2 Moving forward: Adding incentives for participation

Most of this thesis has focused on building efficient, scalable, and secure mechanisms for distributing content, leveraging and organizing resources wherever they may be found. What we have not directly addressed, however, is why nodes will choose to contribute resources. Some of these contributions may arise through altruism. Others are likely as a side-effect of nodes participating in the system for their own observed benefit: *e.g.,* a CoralHTTP proxy that serves its local clients for improved availability may contribute upstream bandwidth in return; a client using Shark's cooperative read protocols for better file-system performance may share its own local file cache. These systems, however, have no secure mechanistic way to incentivize nodes to contribute resources: the CoralCDN or Shark proxy can turn off or block its upstream contributions with little downside.

But if our ultimate goal is to make desired content available to everybody, we should have a means to incentivize sufficient system capacity. Indeed, recall that our current deployment of CoralCDN already suffers from an insufficient bandwidth supply; it can serve only 2 TB of the 10 TB requested on a daily basis. While algorithms can ensure that this capacity is fairly allocated across the set of files or origin sites (*e.g.,* as in §3.4.2), a better goal would be to increase the capacity of our system in the first place!

In ongoing work [63], we are examining how to better address the overall welfare of all participants in peer-to-peer systems. Rather than just enforce that peers contribute *some* upstream resources [43, 80, 97, 184], we seek to address *which* files are "most useful" to disseminate, *where* resource congestion is occurring, and *who* is providing useful content to the system.

We believe that the centerpiece of such a system should be a market-based file-exchange mechanism that associates prices to files and resource constraints. At the system's core, users can act as both *buyers* and *sellers*, with their software exchanging virtual currency in each transaction. Buyers specify which files they seek to download, while sellers decide which files they are willing to upload and their maximum aggregate upload rate. But as the system fully specifies the buy-and-sell behavior of clients—*e.g.,* resource allocation and prices for sellers, and budgeting and peer selection for buyers—price mechanisms are largely invisible to users. Yet users remain incentivized to contribute upstream capacity to ensure a sufficient budget for downloading under resource constraints. On the other hand, if resources are not constrained, prices will drop and even non-contributors (freeloaders) can download files, as it is socially efficient for them to do. This design philosophy—viewing freeloading as a concern only in the context of resource constraints—diverges significantly from prior work.

Second, our design also can benefit two commonly-overlooked participants in peer-to-peer content distribution systems: network providers and content publishers. Specifically, we consider a facility for *network operators to set network cost multipliers* for long-haul traffic carriage. When coupled with a hierarchical network model that captures the congestion points in a network and enables parties to price each accordingly, these cause peer-to-peer traffic to have a strong incentive to remain local when possible, or at least traverse wide-area links that yield more efficient network usage. Furthermore, discovering the instantaneous value of content also can *signal content publishers* as to how they should allocate scarce origin server resources across their set of files. By recognizing the interests of all these parties, we can potentially turn peer-to-peer content distribution into a collaboration between service providers and their users, as opposed to the contention that currently exists (*e.g.,* where ISPs block or rate-limit traffic).

Finally, for each collection of files, a *rendezvous service* can allow peers to discover one another, while a *bank* can track each users' virtual wealth from peer-to-peer payments. We envision multiple rendezvous and bank services to exist simultaneously, each run by a content or service provider that seeks to disseminate a *collection* of files, which may range from large multimedia libraries (*e.g.,* iTunes) to a corpus that spans the entire web (as in CoralCDN). Notice, however, that some of the very techniques we describe in this thesis have a role to play in this new system model as well: locality estimation in OASIS is one way with which to determine network costs, while OASIS's infrastructure can also provide the network cost and rendezvous lookup services needed by such a model. Furthermore, the same distributed key-value indexing techniques we present can be used to scale this bank service, *i.e.,* by partitioning users across servers.

Work is still needed to better understand whether our network model accurately captures and prices resource constraints in the system, how monetary policy should be enacted, how content publishers should allocate server resources to ensure some level of quality-of-service for their clients, and, perhaps most importantly, whether these mechanistic incentives will actually lead to the desired behavior in a real deployment. But the general approach is promising, as it potentially offers benefits for all participants—users, network operators, and content publishers—involved in peer-assisted content distribution.

7.3 Concluding remarks

Without sufficient server and network resources, publishers cannot satisfy the content demands of their audiences. Yet, until now, assembling such resources to meet clients' demands is not quite so simple. At best, publishers turn to commercial CDNs that are centrally-managed and statically-provisioned, running on trusted infrastructure. At worst, content remains unavailable to some clients.

This thesis argues for a new approach, for decentralized systems and open infrastructures that enable the scalable and efficient dissemination of content. By coupling content discovery, server selection, and secure content transmission with incentives that help ensure adequate resources—the very mechanisms proposed in this thesis—one can realize the goal of democratized content distribution. Desired content can be made available to everybody, regardless of its publisher's own resources.

Bibliography

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[2] Akamai Technologies. http://www.akamai.com/, 2007.

[3] Akamai Technologies. SureRoute. http://www.akamai.com/dl/feature_sheets/fs_edgesuite_sureroute.pdf, 2007.

[4] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

[5] C. Anderson. The long tail. *Wired Magazine*, 12(10), Oct. 2004.

[6] R. Anderson, F. Bergadano, B. Crispo, J.-H. Lee, C. Manifavas, and R. Needham. A new family of authentication protocols. *Operating Systems Review*, 32(4):9–20, Oct. 1998.

[7] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roseli, and R. Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, Feb. 1996.

[8] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: Scaling file servers via cooperative caching. In *Proc. 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[9] M. Arlitt and T. Jin. A workload characterization study of the 1998 World Cup Web site. *IEEE Network*, 14(3):30–37, May 2000.

[10] H. Ballani and P. Francis. Towards a global IP anycast service. In *Proc. ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.

[11] N. Barić and B. Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *Advances in Cryptology—EUROCRYPT '97*, Konstanz, Germany, May 1997.

[12] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *Advances in Cryptology—EUROCRYPT '97*, Konstanz, Germany, May 1997.

[13] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. In *Advances in Cryptology—CRYPTO '94*, Santa Barbara, CA, Aug. 1994.

[14] M. Bellare, R. Canetti, and H. Krawczyk. Keyed hash functions and message authentication. In *Advances in Cryptology—CRYPTO '96*, Santa Barbara, CA, Aug. 1996.

[15] M. Bellare, J. Garay, and T. Rabin. Fast batch verification for modular exponentiation and digital signatures. In *Advances in Cryptology—EUROCRYPT '98*, Helsinki, Finland, May 1998.

[16] J. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital sinatures. In *Advances in Cryptology—EUROCRYPT '93*, Lofthus, Norway, May 1993.

[17] BitTorrent. http://www.bittorrent.com/, 2007.

[18] R. Braden. RFC 1123: Requirements for Internet hosts—application and support, Oct. 1989.

[19] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. IEEE INFOCOM*, New York, NY, Mar. 1999.

[20] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A Digital Fountain approach to reliable distribution of bulk data. In *Proc. ACM SIGCOMM*, Vancouver, Canada, Sept. 1998.

[21] J. Byers, M. Luby, and M. Mitzenmacher. Accessing multiple mirror sites in parallel: Using Tornado codes to speed up downloads. In *Proc. IEEE INFOCOM*, New York, NY, Mar. 1999.

[22] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. In *Proc. ACM SIGCOMM*, Pittsburgh, PA, Aug. 2002.

[23] B. Callaghan, B. Pawlowski, and P. Staubach. RFC 1813: NFS version 3 protocol specification, June 1995.

[24] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Advances in Cryptology—CRYPTO '02*, Santa Barbara, CA, Aug. 2002.

[25] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and some efficient constructions. In *Proc. IEEE INFOCOM*, New York, NY, Mar. 1999.

[26] M. Casado and M. J. Freedman. Peering through the shroud: The effect of edge opacity on IP-based client identification. In *Proc. 4th Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, Apr. 2007.

[27] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Proc. 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, Oct. 2000.

[28] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in distributed hash tables. In *Proc. Workshop on Future Directions in Distributed Computing (FuDiCo)*, Bertinoro, Italy, June 2002.

[29] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. Technical Report MSR-TR-2003-94, Microsoft Research, Dec. 2003.

[30] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton's Landing, NY, Oct. 2003.

[31] M. Castro, M. Costa, and A. Rowstron. Debunking some myths about structured and unstructured overlays. In *Proc. 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[32] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[33] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical Internet object cache. In *Proc. USENIX Annual Technical Conference*, San Diego, CA, Jan. 1996.

[34] S. Chari, T. Rabin, and R. Rivest. An efficient signature scheme for route aggregation. Manuscript, Feb. 2002.

[35] D. Chaum, E. van Heijst, and B. Pfitzmann. Cryptographically strong undeniable signatures, unconditionally secure for the signer. In *Advances in Cryptology—CRYPTO '91*, Santa Barbara, CA, Aug. 1991.

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

[37] Y. Chen, K. H. Lim, R. H. Katz, and C. Overton. On the stability of network distance estimation. *SIGMETRICS Performance Evaluation Review*, 30(2):21–30, Sept. 2002.

[38] S. Cheung. An efficient message authentication scheme for link state routing. In *Proc. 13th Annual Computer Security Applications Conference*, San Diego, CA, Dec. 1997.

[39] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[40] B.-G. Chun, P. Wu, H. Weatherspoon, and J. Kubiatowicz. ChunkCast: An anycast service for large content distribution. In *Proc. International Workshop on Peer-to-Peer Systems (IPTPS)*, Santa Barbara, CA, Feb. 2006.

[41] Cisco's CDN Approach. http://www.bcr.com/bcrmag/2000/10/p16s1.asp, Oct. 2000.

[42] CoDeeN. http://codeen.cs.princeton.edu/, 2007.

[43] B. Cohen. Incentives build robustness in BitTorrent. In *Proc. Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.

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

[45] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

[46] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proc. ACM SIGCOMM*, Portland, OR, Aug. 2004.

[47] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proc. 1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.

[48] A. Das, I. Gupta, and A. Motivala. SWIM: Scalable weakly-consistent infection-style process group membership protocol. In *Proc. Dependable Systems and Networks*, Washington, DC, June 2002.

[49] C. de Launois, S. Uhlig, and O. Bonaventure. A stable and distributed network coordinate system. Technical report, Universite Catholique de Louvain, Dec. 2004.

[50] Digg. http://www.digg.com/, 2007.

[51] Edge Side Includes. http://www.esi.org/, 2007.

[52] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of condors: Load sharing among workstation clusters. *J. Future Generations of Computer Systems*, 12:53–65, May 1996.

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

[54] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol—HTTP/1.1, June 1999.

[55] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, Apr. 1995.

[56] FIPS 186-2. *Digital Signature Standard (DSS)*. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, 2000.

[57] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A global Internet host distance estimation service. *IEEE/ACM Transactions on Networking*, 9(5): 525–540, Oct. 2001.

[58] M. J. Freedman and D. Mazières. Sloppy hashing and self-organizing clusters. In *Proc. International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, Mar. 2003.

[59] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. 1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.

[60] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and DHTs. In *Proc. USENIX Workshop on Real Large Distributed Systems (WORLDS)*, San Francisco, CA, Dec. 2005.

[61] M. J. Freedman, M. Vutukuru, N. Feamster, and H. Balakrishnan. Geographic locality of IP prefixes. In *Proc. Internet Measurement Conference*, Berkeley, CA, Oct. 2005.

[62] M. J. Freedman, K. Lakshminarayanan, and D. Mazières. OASIS: Anycast for any service. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[63] M. J. Freedman, C. Aperjis, and R. Johari. Prices are right: Aligning incentives for peer-assisted content distribution. Manuscript, Aug. 2007.

[64] S. Gadde, J. Chase, and M. Rabinovich. A taste of crispy squid. In *Proc. Workshop on Internet Server Performance*, Madison, WI, June 1998.

[65] P. Ganesan, K. Gummadi, and H. Garcia-Molina. Canon in G Major: Designing DHTs with hierarchical structure. In *Proc. International Conference on Distributed Computing Systems (ICDCS)*, Tokyo, Japan, Mar. 2004.

[66] R. Gennaro and P. Rohatgi. How to sign digital streams. In *Advances in Cryptology—CRYPTO '97*, Santa Barbara, CA, Aug. 1997.

[67] S. Gerding and J. Stribling. Examining the tradeoffs of structured overlays in a dynamic non-transitive network. http://pdos.csail.mit.edu/~strib/docs/projects /networking_fall2003.pdf, Dec. 2003.

[68] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, Oct. 2003.

[69] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proc. 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, Oct. 1998.

[70] P. Golle and N. Modadugu. Authenticated streamed data in the presernce of random packet loss. In *Proc. Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2001.

[71] Google Maps. http://maps.google.com/, 2007.

[72] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. 12th ACM Symposium on Operating Systems Principles (SOSP)*, Litchfield Park, AZ, Dec. 1989.

[73] R. Grimm, G. Lichtman, N. Michalakis, A. Elliston, A.Kravetz, J. Miller, and S. Raza. Na Kika: Secure service execution and composition in an open edge-side computing network. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[74] B. Gronvall, A. Westerlund, and S. Pink. The design of a multicast-based distributed file system. In *Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, Feb. 1999.

[75] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary Internet end hosts. In *Proc. Internet Measurement Workshop*, San Francisco, CA, Nov. 2001.

[76] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. Wetherall. Improving the reliability of Internet paths with one-hop source routing. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[77] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, Oct. 2003.

[78] K. P. Gummadi, R. Gummadi, S. D. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proc. ACM SIGCOMM*, Karlsruhe, Germany, Aug. 2003.

[79] I. Gupta, K. Birman, P. Linga, A. Demers, and R. V. Renesse. Kelips: building an efficient and stable P2P DHT through increased memory and background overhead. In *Proc. International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, Mar. 2003.

[80] M. Gupta, P. Judge, and M. Ammar. A reputation system for peer-to-peer networks. In *Proc. ACM NOSSDAV*, Monterey, CA, June 2003.

[81] J. Guyton and M. Schwartz. Locating nearby copies of replicated Internet servers. In *Proc. ACM SIGCOMM*, Cambridge, MA, Aug. 1995.

[82] Hadoop. http://lucene.apache.org/hadoop/, 2007.

[83] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. In *Proc. 14th ACM Symposium on Operating Systems Principles (SOSP)*, Asheville, NC, Dec. 1993.

[84] N. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proc. USENIX Symposium on Internet Technologies and Systems (USITS)*, Seattle, WA, Mar. 2003.

[85] 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 Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[86] IBM Tivoli. Storage manager. http://www.ibm.com/tivoli, 2007.

[87] IP to Lat/Long server. http://cello.cs.uiuc.edu/cgi-bin/slamm/ip2ll/, 2005.

[88] Iperf. Version 1.7.0—the TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf/, 2005.

[89] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *Proc. Principles of Distributed Computing (PODC)*, Monterey, CA, July 2002.

[90] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from DNS rebinding attacks. In *Proc. 14th ACM Conference on Computer and Communication Security (CCS)*, Alexandria, VA, Oct. 2007.

[91] R. Johnson, D. Molnar, D. Song, and D. Wagner. Homomorphic signature schemes. In *Progress in Cryptology—CT-RSA 2002*, San Jose, CA, Feb. 2002.

[92] D. Joseph, J. Kannan, A. Kubota, K. Lakshminarayanan, I. Stoica, and K. Wehrle. OCALA: An architecture for supporting legacy applications over overlays. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[93] T. Karagiannis, P. Rodriguez, and K. Papagiannaki. Should Internet service providers fear peer-assisted content distribution? In *Proc. Internet Measurement Conference*, Berkeley, CA, Oct. 2005.

[94] 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 *Proc. ACM Symposium on the Theory of Computing (STOC)*, El Paso, TX, May 1997.

[95] D. R. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *Proc. ACM Symposium on the Theory of Computing (STOC)*, Montreal, Canada, May 2002.

[96] C. Karlof, N. Sastry, Y. Li, A. Perrig, and J. Tygar. Distillation codes and applications to DoS resistant multicast authentication. In *Proc. 11th Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, Feb. 2004.

[97] I. Kash, E. J. Friedman, and J. Y. Halpern. Optimizing scrip systems: Efficiency, crashes, hoarders, and altruists. In *Proc. ACM Conference on Electronic Commerce*, San Diego, CA, June 2007.

[98] D. Katabi and J. Wroclawski. A framework for scalable global IP-anycast (GIA). In *Proc. ACM SIGCOMM*, Stockholm, Sweden, Aug. 2000.

[99] K. Keys. Clients of DNS root servers, 2002-08-14. http://www.caida.org/projects/dns-analysis/, Aug. 2002.

[100] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *Proc. ACM SIGCOMM*, Kyoto, Japan, Aug. 2007.

[101] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton's Landing, NY, Oct. 2003.

[102] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In *Advances in Cryptology—CRYPTO '01*, Santa Barbara, CA, Aug. 2001.

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

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

[105] 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 *Proc. 9th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, Nov. 2000.

[106] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2): 133–169, May 1998.

[107] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proc. 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[108] LimeLight Networks. http://www.limelightnetworks.com/, 2007.

[109] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. *Operating Systems Review*, 25(5):226–238, Oct. 1991.

[110] M. Luby. LT codes. In *Proc. IEEE Symposium on Foundation of Computer Science (FOCS)*, Vancouver, Canada, Nov. 2002.

[111] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann. Practical loss-resilient codes. In *Proc. ACM Symposium on the Theory of Computing (STOC)*, El Paso, TX, May 1997.

[112] R. Malpani, J. Lorch, and D. Berger. Making world wide web caching servers cooperate. In *Proc. World Wide Web Conference*, Darmstadt, Germany, Apr. 1995.

[113] T. Mann, A. D. Birrell, A. Hisgen, C. Jerian, and G. Swart. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems*, 12(2):123–164, May 1994.

[114] Z. M. Mao, C. Cranor, F. Douglis, M. Rabinovich, O. Spatscheck, and J. Wang. A precise and efficient evaluation of the proximity between web clients and their local DNS servers. In *Proc. USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[115] P. Maymounkov. Online codes. Technical Report 2002-833, New York University, Nov. 2002.

[116] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proc. International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, Mar. 2002.

[117] P. Maymounkov and D. Mazières. Rateless codes and big downloads. In *Proc. International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, Feb. 2003.

[118] D. Mazières. A toolkit for user-level file systems. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2001.

[119] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, SC, Dec. 1999.

[120] R. C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology—CRYPTO '87*, Santa Barbara, CA, Aug. 1987.

[121] S. Micali and R. Rivest. Transitive signature schemes. In *Progress in Cryptology—CT-RSA 2002*, San Jose, CA, Feb. 2002.

[122] S. Miner and J. Staddon. Graph-based authentication of digital streams. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

[123] Mirror Image Internet. http://www.mirror-image.com/, 2007.

[124] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 13(3):217–252, Aug. 1997.

[125] D. Morrison. Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, Oct. 1968.

[126] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

[127] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[128] D. Neel. Cogent, Level 3 in standoff over Internet access. TechWeb, Oct. 2005.

[129] E. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In *Proc. IEEE INFOCOM*, New York, NY, June 2002.

[130] OASIS. http://oasis.coralcdn.org/, 2007.

[131] B. M. Oki and B. H. Liskov. Viewstamped replication: a new primary copy method to support highly available distributed systems. In *Proc. Principles of Distributed Computing (PODC)*, New York, NY, 1988.

[132] V. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *Proc. International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, Mar. 2002.

[133] V. N. Padmanabhan and L. Subramanian. An investigation of geographic mapping techniques for Internet hosts. In *Proc. ACM SIGCOMM*, San Diego, CA, Aug. 2001.

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

[135] A. Pannetrat and R. Molva. Efficient multicast packet authentication. In *Proc. Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2003.

[136] J. M. Park, E. K. P. Chong, and H. J. Siegel. Efficient multicast stream authentication using erasure codes. *ACM Transactions on Information and System Security*, 6(2):258–285, May 2003.

[137] C. Patridge, T. Mendez, and W. Milliken. RFC 1546: Host anycasting service, Nov. 1993.

[138] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology—CRYPTO '91*, Santa Barbara, CA, Aug. 1991.

[139] D. M. R. Periakaruppan and J. Donohoe. Where in the world is netgeo.caida.org? In *Proc. Internet Society Conference*, Yokohama, Japan, June 2000.

[140] A. Perrig. The BiBa one-time signature and broadcast authentication protocol. In *Proc. 8th ACM Conference on Computer and Communication Security (CCS)*, Philadelphia, PA, Nov. 2001.

[141] A. Perrig, R. Canetti, D. Song, and D. Tygar. Efficient authentication and signature of multicast streams over lossy channels. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

[142] A. Perrig, R. Canetti, D. Song, and D. Tygar. Efficient and secure source authentication for multicast. In *Proc. Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2001.

[143] G. Pfister and V. A. Norton. "Hot spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, 34(10):943–948, Oct. 1985.

[144] P. Pietzuch, J. Ledlie, and M. Seltzer. Supporting network coordinates on PlanetLab. In *Proc. USENIX Workshop on Real Large Distributed Systems (WORLDS)*, San Francisco, CA, Dec. 2005.

[145] PlanetLab. http://www.planet-lab.org/, 2007.

[146] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proc. 4th Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, Apr. 2007.

[147] Quova. http://www.quova.com/, 2007.

[148] M. O. Rabin. Digitalized signatures and public key functions as intractable as factorization. Technical Report 212, MIT Laboratory for Computer Science, Jan. 1979.

[149] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proc. 1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.

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

[151] P. Reiher, J. T. Page, G. J. Popek, J. Cook, and S. Crocker. Truffles—a secure service for widespread file sharing. In *Proc. PSRG Workshop on Network and Distributed System Security*, San Diego, CA, Feb. 1993.

[152] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. RFC 1918: Address allocation for private Internets, Feb. 1996.

[153] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang. BGP routing stability of popular destinations. In *Proc. Internet Measurement Workshop*, Marseille, France, Nov. 2002.

[154] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proc. USENIX Conference on File and Storage Technologies*, Berkeley, CA, Mar. 2003.

[155] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2004.

[156] S. Rhea, B.-G. Chun, J. Kubiatowicz, and S. Shenker. Fixing the embarrassing slowness of OpenDHT on PlanetLab. In *Proc. USENIX Workshop on Real Large Distributed Systems (WORLDS)*, San Francisco, CA, Dec. 2005.

[157] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proc. ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.

[158] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM Computer Communication Review*, 27(2):24–36, Apr. 1997.

[159] P. Rohatgi. A compact and fast hybrid signature scheme for multicast packet authentication. In *Proc. 6th ACM Conference on Computer and Communication Security (CCS)*, Singapore, Nov. 1999.

[160] RouteViews. http://www.routeviews.org/, 2007.

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

[162] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, Nov. 2001.

[163] S. Roy, H. Pucha, Z. Zhang, Y. C. Hu, and L. Qiu. Overlay node placement: Analysis, algorithms and impact on applications. In *Proc. International Conference on Distributed Computing Systems (ICDCS)*, Toronto, Canada, June 2007.

[164] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proc. USENIX Annual Technical Conference*, Portland, OR, June 1985.

[165] S. Saroui, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An analysis of Internet content delivery systems. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Oct. 2002.

[166] S. Sen and J. Wang. Analyzing peer-to-peer traffic across large networks. In *Proc. Internet Measurement Workshop*, Marseille, France, Nov. 2002.

[167] K. Shanahan and M. J. Freedman. Locality prediction for oblivious clients. In *Proc. International Workshop on Peer-to-Peer Systems (IPTPS)*, Ithaca, NY, Feb. 2005.

[168] A. Sherman, P. Lisiecki, A. Berkheimer, and J. Wein. ACMS: Akamai Configuration Management System. In *Proc. 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[169] A. Shokrollahi. Raptor codes. Technical Report DF2003-06-001, Digital Fountain, Inc., June 2003.

[170] Slashdot. http://www.slashdot.com/, 2007.

[171] Sleepycat. BerkeleyDB v4.2. http://www.sleepycat.com/, 2005.

[172] Y. J. Song, V. Ramasubramanian, and E. Sirer. Optimal resource utilization in content distribution networks. Technical Report 2005-2004, Cornell University, Nov. 2005.

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

[174] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *Proc. International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, Mar. 2002.

[175] A. Stavrou, D. Rubenstein, and S. Sahu. A lightweight, robust P2P system to handle flash crowds. In *Proc. IEEE International Conference on Network Protocols*, Paris, France, Nov. 2002.

[176] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proc. USENIX Annual Technical Conference*, Dallas, TX, Feb. 1988.

[177] 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 Transactions on Networking*, 11(1):17–32, Feb. 2003.

[178] J. Stribling. PlanetLab All Pairs Ping data, 08-03-2005. http://www.pdos.lcs.mit.edu/~strib/pl_app/, 2005.

[179] J. Stribling, J. Li, I. Councill, M. F. Kaashoek, and R. Morris. OverCite: A cooperative digital research library. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

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

[181] W. Theilmann and K. Rothermel. Dynamic distance maps of the Internet. In *Proc. IEEE INFOCOM*, Tel Aviv, Israel, Mar. 2000.

[182] C. Thekkath, T. Mann, and E. Lee. Frangipani: A scalable distributed file system. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint Malo, France, Oct. 1997.

[183] G. Tsudik and S. Xu. Accumulating composites and improved group signing. In *Advances in Cryptology—ASIACRYPT '03*, Taipei, Taiwan, Nov. 2003.

[184] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A secure economic framework for P2P resource sharing. In *Proc. Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.

[185] L. A. Wald and S. Schwarz. 1999 Southern California Seismic Network Bulletin. *Seismological Research Letters*, 71(4):401–422, July/Aug. 2000.

[186] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on CDN robustness. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec 2002.

[187] D. Wessels and K. Claffy. RFC 2186: Internet Cache Protocol (ICP) version 2, Sept. 1997.

[188] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[189] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. Levy. On the scale and performance of cooperative web proxy caching. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, SC, Dec. 1999.

[190] B. Wong and E. G. Sirer. ClosestNode.com: an open access, scalable, shared geocast service for distributed systems. *Operating Systems Review*, 40(1):62–64, Jan. 2006.

[191] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A lightweight network location service without virtual coordinates. In *Proc. ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.

[192] C. K. Wong and S. S. Lam. Digital signatures for flows and multicasts. In *Proc. IEEE International Conference on Network Protocols*, Austin, TX, Oct. 1998.

[193] C. Yoshikawa, F. Annexstein, and K. A. Berman. The Galaxy filesystem toolkit: Providing Windows Explorer access to custom data. Technical Report ECECS-TR-2006-7, U. Cincinnati, July 2006.

[194] H. Yu, D. Zheng, B. Y. Zhao, and W. Zheng. Understanding user behavior in large-scale video-on-demand systems. In *Proc. EuroSys*, Leuven, Belgium, Apr. 2006.

[195] 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*, 22(1):41–53, Jan. 2004.