Michael J. Freedman

353 Serra Mall, Room 288
Stanford, CA 94305-9025
(650) 723-1863

http://www.michaelfreedman.org/
mfreed@scs.stanford.edu
Citizenship: US

Education

New York University .... New York, NY

Ph.D. Candidate in Computer Science .... Expected May 2007
Thesis title: *Harnessing Widespread Cooperation to Democratize Content Distribution*
Visiting **Stanford University**, September 2005–May 2007
Advisor: David Mazières ; GPA: 4.0/4.0

M.S. in Computer Science .... May 2005
Advisor: David Mazières ; GPA: 4.0/4.0

Massachusetts Institute of Technology .... Cambridge, MA

M.Eng. in Electrical Engineering and Computer Science .... June 2002
Thesis title: *A Peer-to-Peer Anonymizing Network Layer*
Advisor: Robert Morris ; GPA: 5.0/5.0

S.B. in Computer Science, Minor in Political Science .... June 2001
Thesis title: *An Anonymous Communications Channel for the Free Haven Project*
Advisor: Ron Rivest ; GPA: 4.9/5.0

Interests

Distributed systems, security, networking, and cryptography

Research

2002–present **Cooperative content distribution.** Conceived and led the Coral Project. Designed and built an Internet-scale, self-organizing web-content distribution network: CoralCDN [11] uses a network of cooperating DNS redirectors and HTTP proxies, backed by a decentralized indexing infrastructure [18], to allow oblivious clients to transparently download content from nearby servers, while avoiding distant or heavily-loaded ones. CoralCDN has been in production use on 300 servers since March 2004, currently receiving about 25 million HTTP requests from over 1 million clients per day, serving several terabytes of data. `http://coralcdn.org/`

With a focus on settings with mutually-distrustful clients, Shark [6] provides a distributed file system that improves scalability and performance through cooperative reads, using Coral's indexing layer to locate files. Yet Shark preserves traditional semantics, manageability, and security. Other research provides integrity guarantees for large files encoded with rateless erasure codes, via a homomorphic hash function that can verify downloaded blocks on-the-fly [10].

Ongoing focus on untrusted settings for CDNs (with C. Aperjis, R. Johari, and D. Mazières), devising incentive-compatible mechanisms that cause nodes to contribute bandwidth for improved quality-of-service. This work uses market-pricing techniques and virtual currency to ensure effective bandwidth usage and network utilization, while still preventing cheating.

2005–present **Anycast.** Designed and built OASIS, a server-selection infrastructure that provides locality- and load-based anycast for replicated Internet services [3] [26]. OASIS tackles the problems of leveraging disparate services to perform (potentially error-prone) network measurement and of scalably managing state information about many services and their participating nodes. OASIS has been in production use since Nov. 2005 and has been adopted by more than a dozen distributed services, handling thousands of replicas. Performed background studies of the geographic locality of IP prefixes [5] and the efficacy of virtual coordinate systems [16]. `http://oasis.coralcdn.org/`

2006–present **IP analytics.** By instrumenting CoralCDN, used active web content to measure and analyze the characteristics of over 7 million clients with respect to "edge technologies" (NATs, proxies, DNS

	and DHCP) [1]. Results quantify how Internet services can use IP addresses to identify clients and enforce access-control decisions. Commercialized historical and real-time techniques for proxy detection and IP geolocation; acquired by Quova, Inc. in Nov. 2006 and currently being tested at large Internet services. `http://illuminati.coralcdn.org/`
2006–present	**Enterprise networks.** Design and implementation contributions to Ethane [2] [25], a backwards-compatible protection and management architecture for enterprise networks. Ethane network switches provide connectivity through on-demand virtual circuits, yet they enforce security policies on a per-flow basis through centrally-managed, atomic, auditable name bindings. Deployment at Stanford since Nov. 2006, serving hundreds of hosts. `http://yuba.stanford.edu/ethane/`
2005–present	**Reliable email.** Designed and implemented the security and privacy protections in Re:, an email acceptance system that leverages social proximity for automated whitelisting [4], using private matching [9]. Recent analysis of privacy for social networks led to more efficient protocols based only on symmetric-key operations (or achieving stronger properties using bilinear maps) [13].
2005–present	**Fault-tolerance groups.** Researched abstractions for the scalable construction of fault-tolerant, distributed systems [14]. Ongoing work with L. Subramanian on partitioning large, dynamic systems into smaller groups, which apply fault-tolerance or reliable communication protocols.
2000–present	**Privacy-preserving protocols.** Developed cryptographic protocols for private matching (PM), which computes the set intersection between two or more parties' inputs [9]. PM uses the properties of homomorphic encryption to privately evaluate a polynomial representation of input sets. Subsequent work led to improved constructions for keyword search (KS) based on oblivious pseudorandom functions [7]. Earlier research included the design and implementation of a prototype system for anonymous cryptographic e-cash (with S. Brands and I. Goldberg), as well as considerations for privacy-enabled digital rights management (DRM) systems [19] [22].
2000–2002	**Anonymity systems.** Designed and implemented Tarzan [12] [20], a peer-to-peer anonymous IP network layer that is strongly resistant to traffic analysis. Helped design Free Haven, a distributed system for the anonymous publishing, storage, and retrieval of information [23] [24] [28].

Positions

3/06–present	**Co-founder** (with Martin Casado). Illuminics Systems, Mountain View, CA.
9/05–present	**Research Assistant.** Stanford University (SCS Group), Stanford, CA.
5/05–8/05	**Research Assistant.** University of California, Berkeley, Berkeley, CA.
9/02–5/05	**Research Assistant.** New York University (SCS Group), New York, NY.
5/03–8/03	**Research Associate.** HP Labs (Trusted Systems Lab), Princeton, NJ.
9/01–6/02	**Research Assistant.** MIT LCS (PDOS Group), Cambridge, MA.
5/01–8/01	**Research Intern.** InterTrust Technologies (STAR Lab), Santa Clara, CA.
6/00–8/00	**Research Intern.** Zero-Knowledge Systems Labs, Montreal, Quebec.
2/99–5/01	**Undergrad Researcher.** MIT LCS (SLS and CIS Groups), Cambridge, MA.
6/99–8/99	**Intern.** Sun Microsystems (HPC Group), Burlington, MA.
6/98–8/98	**Intern.** Cognex Corporation, Natick, MA.
6/96–2/98	**Undergrad Researcher.** MIT Francis Bitter Magnet Lab, Cambridge, MA.

Service

5/03–5/05	**Founder and Organizer.** NYU Systems Reading Group, New York, NY.
2/04–5/05	**Faculty Representative.** NYU Courant Student Organization, New York, NY.
9/01–5/02	**Co-organizer.** MIT Applied Security Reading Group, Cambridge, MA.
9/97–5/02	**President, VP, Winter School Organizer.** MIT Outing Club, Cambridge, MA

Teaching

1/04–5/04	**Teaching Assistant, Lab Instructor.** V22.0480—Computer Networks, NYU.
2/02–5/02	**Teaching Assistant.** 6.033—Computer System Engineering, MIT.
2/01–5/01	**Teaching Assistant.** 6.033—Computer System Engineering, MIT

Advising

Masters	Justin Pettit (Stanford), Robert Soule (NYU), Jeff Borden (NYU)
Undergraduates	Jeffrey Spehar (Stanford), Kevin Shanahan (NYU), Ed Kupershlak (NYU)

Professional activities

Program comm.	WORLDS ’06, UPGRADE-CDN ’06, IRIS Student P2P Workshop ’03
External reviews	NSDI ’07, LATIN ’06, HotNets ’05, EUROCRYPT ’05, Usenix Technical ’05, ISC ’04, CRYPTO ’04, IPDPS ’04, INFOCOM ’04, CCS ’03, SOSP ’03, ISC ’03, PODC ’03, EUROCRYPT ’03, WPES ’02
Journal reviews	ACM Transactions on Computer Systems (TOCS), Journal of Cryptology, Journal of Parallel and Distributed Computing (JPDC), Handbook of Internet Security - P2P Security (Wiley & Sons), Computer Journal

Honors

NDSEG (DoD) Graduate Fellow, 2002-2005
NYU McCracken Fellow, 2002-2006
Henning Biermann Award, NYU Computer Science, 2005 (for outstanding education and service)

Best demo (OASIS), WORLDS 2005.
First paper (highest-ranked), EUROCRYPT 2004 [9].
Award paper, CCS 2002 [12].

Awarded NSF Graduate Fellowship, 2001
Awarded Gordon Wu Fellowship (Princeton), 2001 ; Sterling Prize Fellowship (Yale), 2001
Awarded Graduate Fellowships (U.C.Berkeley, Carnegie-Mellon, UCSD), 2001

Coca-Cola Scholar, 1997-2001 ; Tylenol Scholar, 1997-1999 ; Big 33 Scholar, 1997-1998
Tau Beta Pi, 2000 ; Eta Kappa Nu, 2000 ; Sigma Xi, 2000 ; Order of Omega, 1999
Congressional Award, Silver (1996) and Bronze (1993) medals

Refereed conference publications

[1] Martin Casado and **Michael 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 07)*, Cambridge, MA, April 2007.

[2] Martin Casado, Tal Garfinkle, Aditya Akella, **Michael J. Freedman**, Dan Boneh, Nick McKeown, and Scott Shenker. SANE: A protection architecture for enterprise networks. In *Proc. 15th USENIX Security Symposium*, pages 137–151, Vancouver, BC, August 2006.

[3] **Michael J. Freedman**, Karthik Lakshminarayanan, and David Mazières. OASIS: Anycast for any service. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI 06)*, pages 129–142, San Jose, CA, May 2006.

[4] Scott Garriss, Michael Kaminsky, **Michael J. Freedman**, Brad Karp, David Mazières, and Haifeng Yu. Re: Reliable email. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI 06)*, pages 297–310, San Jose, CA, May 2006.

[5] **Michael J. Freedman**, Mythili Vutukuru, Nick Feamster, and Hari Balakrishnan. Geographic locality of IP prefixes. In *Proc. 5th ACM SIGCOMM Conference on Internet Measurement (IMC 05)*, pages 153–158, Berkeley, CA, October 2005.

[6] Siddhartha Annapureddy, **Michael J. Freedman**, and David Mazières. Shark: Scaling file servers via cooperative caching. In *Proc. 2nd Symposium on Networked Systems Design and Implementation (NSDI 05)*, pages 129–142, Boston, MA, May 2005.

[7] **Michael J. Freedman**, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom function. In *Proc. 2nd Theory of Cryptography Conference (TCC 05)*, pages 303–324, Cambridge, MA, February 2005.

[8] Yevgeniy Dodis, **Michael J. Freedman**, Stanislaw Jarecki, and Shabsi Walfish. Versatile padding schemes for joint signature and encryption. In *Proc. 11th ACM Conference on Computer and Communication Security (CCS 04)*, pages 344–353, Washington, D.C., October 2004.

[9] **Michael J. Freedman**, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology — EUROCRYPT 2004*, pages 1–19, Interlaken, Switzerland, May 2004.

[10] Maxwell Krohn, **Michael J. Freedman**, and David Mazières. On-the-fly verification of rateless erasure codes for efficient content distribution. In *Proc. IEEE Symposium on Security and Privacy*, pages 226–240, Oakland, CA, May 2004.

[11] **Michael J. Freedman**, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *Proc. 1st Symposium on Networked Systems Design and Implementation (NSDI 04)*, pages 239–252, San Francisco, CA, March 2004.

[12] **Michael J. Freedman** and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proc. 9th ACM Conference on Computer and Communications Security (CCS 2002)*, pages 193–206, Washington, D.C., November 2002.

Refereed workshop publications

[13] **Michael J. Freedman** and Antonio Nicolosi. Efficient private techniques for verifying social proximity. In *Proc. 6th International Workshop on Peer-to-Peer Systems (IPTPS 07)*, Bellevue, WA, February 2007.

[14] **Michael J. Freedman**, Ion Stoica, David Mazières, and Scott Shenker. Group therapy for systems: Using link-attestations to manage failures. In *Proc. 5th International Workshop on Peer-to-Peer Systems (IPTPS 06)*, Santa Barbara, CA, February 2006.

[15] **Michael J. Freedman**, Karthik Lakshminarayanan, Sean Rhea, and Ion Stoica. Non-transitive connectivity and DHTs. In *Proc. 2nd Workshop on Real, Large, Distributed Systems (WORLDS 05)*, pages 55–60, San Francisco, CA, December 2005.

[16] Kevin Shanahan and **Michael J. Freedman**. Locality prediction for oblivious clients. In *Proc. 4th International Workshop on Peer-to-Peer Systems (IPTPS 05)*, pages 252–263, Ithaca, NY, February 2005.

[17] Max Krohn and **Michael J. Freedman**. On-the-fly verification of erasure-encoded file transfers (extended abstract). In *Proc. 1st IRIS Student Workshop on Peer-to-Peer Systems*, Cambridge, MA, August 2003.

[18] **Michael J. Freedman** and David Mazières. Sloppy hashing and self-organizing clusters. In *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS 03)*, pages 45–55, Berkeley, CA, February 2003.

[19] Joan Feigenbaum, **Michael J. Freedman**, Tomas Sander, and Adam Shostack. Economic barriers with existing privacy technologies in e-commerce systems. In *Proc. Workshop on Economics and Information Security*, Berkeley, CA, May 2002.

[20] **Michael J. Freedman**, Emil Sit, Josh Cates, and Robert Morris. Introducing Tarzan, a peer-to-peer anonymizing network layer. In *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS 02)*, pages 121–129, Cambridge, MA, March 2002.

[21] **Michael J. Freedman** and Radek Vingralek. Efficient peer-to-peer lookup based on a distributed trie. In *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS 02)*, pages 66–75, Cambridge, MA, March 2002.

[22] Joan Feigenbaum, **Michael J. Freedman**, Tomas Sander, and Adam Shostack. Privacy engineering in digital rights management systems. In *Proc. ACM Workshop in Security and Privacy in Digital Rights Management (DRM 01)*, pages 76–105, Philadelphia, PA, November 2001.

[23] Roger Dingledine, **Michael J. Freedman**, David Hopwood, and David Molnar. A reputation system to increase MIX-net reliability. In *Proc. Information Hiding Workshop (LNCS 2137)*, pages 126–141, Pittsburgh, PA, March 2001.

[24] Roger Dingledine, **Michael J. Freedman**, and David Molnar. The Free Haven Project: Distributed anonymous storage service. In *Proc. Workshop on Design Issues in Anonymity and Unobservability (LNCS 2009)*, pages 67–95, Berkeley, CA, July 2000.

In submission

[25] Martin Casado, **Michael J. Freedman**, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise, 2007.

Unrefereed publications, book chapters

[26] **Michael J. Freedman**. Automating server selection with OASIS. In *;login: The USENIX Magazine*, pages 46–52, October 2006.

[27] Roger Dingledine, **Michael J. Freedman**, David Molnar, and David Parkes. Reputation. In *Digital Government Civic Scenario Workshop*, Cambridge, MA, April 2003.

[28] Roger Dingledine, **Michael J. Freedman**, and David Molnar. *Peer-to-Peer: Harnessing the Power of Disruptive Technology*, chapter Accountability, pages 271–340. O'Reilly, 2001.

[29] Roger Dingledine, **Michael J. Freedman**, and David Molnar. *Peer-to-Peer: Harnessing the Power of Disruptive Technology*, chapter Free Haven, pages 159–190. O'Reilly, 2001.

References

Prof. David Mazières
Stanford University
Computer Science Department
353 Serra Mall, #290
Stanford, CA 94305-9025
(650) 723-8777
`rec@nospam.scs.stanford.edu`

Prof. Frans Kaashoek
Massachusetts Institute of Technology
Stata Center, #32-G992
77 Massachusetts Avenue
Cambridge, MA 02139
(617) 253-7149
`kaashoek@csail.mit.edu`

Prof. Ion Stoica
University of California, Berkeley
RADLab, Room 465
Soda Hall #1776
Berkeley, CA 94720-1776
(510) 643-4007
`istoica@cs.berkeley.edu`

Prof. Nick McKeown
Stanford University
Computer Science Department
353 Serra Mall, #340
Stanford, CA 94305-9025
(650) 725-3641
`nickm@stanford.edu`

Prof. Joan Feigenbaum
Yale University
Computer Science Department
P.O. Box 208285,
New Haven, CT 06520-8285
(203) 432-6432
`joan.feigenbaum@yale.edu`

Stanford, CA, January 31, 2007

Research statement

Michael J. Freedman

My research interests span the areas of distributed systems, security, networking, and cryptography. I particularly enjoy devising technologies that make new functionality broadly available. My work generally tackles systems problems by coupling principled designs with real-world deployments.

A common thread in my research is the extension of systems designed for centralized or trusted entities into decentralized, untrusted, unreliable, or chaotic settings. These scenarios offer significant challenges, yet they are ones ideally suited for academic research: Such problems or architectures do not naturally arise from within industry, even though the techniques often may be applied back into managed environments, *e.g.*, to survive disasters or to operate safely under attack. More than that, open systems encourage further innovation.

I approach these problems through the innovative use of cryptography, algorithms, or abstractions. By leveraging the resulting properties, one can create self-organizing systems out of unreliable nodes, incentivize proper operation, curtail the impact of malicious behavior, or improve manageability to overcome system brittleness.

Such solutions still require solid engineering, always with the end-user in mind. By providing desired functionality, even research systems can attract users, gain traction, and then truly test the system's mettle. Deployed systems provide real data to direct future design decisions, and they can serve as platforms for otherwise intractable experiments. While much research relies solely on simulation and emulation, only at scale can we truly evaluate many systems—learning from their strengths, weaknesses, and emergent properties—and thus discover new research problems and directions.

Cooperative content distribution. My thesis research focuses on making content delivery more widely available by federating large numbers of untrusted or unreliable machines to share localized resources. Content distribution networks (CDNs) are not a new idea, but the architectures of commercial CDNs are tightly bound to centralized control, static deployments, and cost recovery.

My initial system, CoralCDN [1], explores how to build a self-organizing cooperative web CDN using unreliable hosts. Through its scalable distributed index, nodes can record and locate data without overloading any node, regardless of a file's popularity or system dynamics [1, 2]. Decentralized clustering algorithms enable nodes to find nearby data without querying more distant machines.

CoralCDN incorporates a number of engineering mechanisms for sharing resources fairly and preventing abuse—learned through deployment and community feedback—yet the system is inherently open. Simply modify a URL, and the requested content is automatically retrieved and cached by CoralCDN's proxies. As such, it has been widely adopted in often innovative ways: by servers to dynamically offload flash crowds, by browser extensions to recover from server failures, by podcasting and RSS software, and by daily links on Slashdot and other portals. CoralCDN currently handles about 25 million requests daily from over one million clients.

One challenge in designing CoralCDN was how to compel our unmodified clients to use nearby, unloaded proxies. While commercial systems also deploy *anycast* to select servers, their techniques need handle only a single deployment, often comprised of a mere handful of data centers. Ideally, one public infrastructure could provide anycast for many far-flung services, such that the more services that use it, the more accurate its server-selection results and the lower the bandwidth cost per service.

I built a subsequent system, OASIS [3], that does exactly this: OASIS currently provides anycast among thousands of servers from more than a dozen distributed systems, from both the academic and open-source communities. It flexibly supports a variety of interfaces—currently DNS, HTTP, and RPC—with which clients can discover good servers belonging to the requested system. OASIS can do so because it tackles several problems simultaneously: using nodes from participating services to perform network measurement, detecting and disambiguating erroneous results, representing locality stably across time and deployment changes, and scalably managing state information about many services.

This success at building content delivery from unreliable resources raised the question as to whether we could extend this approach to mutually distrustful clients. Shark [4] provides a distributed file system that improves scalability and performance through cooperative reads, using Coral's indexing layer to locate content. Still, Shark preserves traditional semantics and security: End-to-end cryptography ensures that clients need not trust one another.

We also considered security mechanisms for hosts using rateless erasure codes for cooperative large file distribution. Unfortunately, these codes cannot use traditional authenticators (*e.g.*, hash trees) that guarantee the integrity of individual blocks. Therefore, we devised a homomorphic hash function that can be used to verify downloaded blocks on-the-fly, thus preventing malicious participants from polluting the network with garbage [5]. Implementation aspects mattered in this seemingly-theoretical project. The batching of public-key operations was needed to achieve fast verification, while disk-read strategies led to encoding speeds that even exceeded those of hash trees for non-rateless codes. Finally, for preventing pollution in these non-rateless codes, we showed how simple implementation changes could replace others' heavyweight *black-box* mechanisms.

Recently, I have returned to the problem of moving CoralCDN from its current deployment on PlanetLab onto fully untrusted nodes, as CoralCDN's success has led to bandwidth usage that has long saturated PlanetLab's available capacity. As digital signatures can guarantee content integrity, the challenge is ensuring that sufficient capacity exists. Our latest design promotes resource sharing through incentive-compatible mechanisms: Contributing nodes receive better quality-of-service when the system is under-provisioned. The system applies market pricing techniques to efficiently use available bandwidth, but also incorporates network costs to "play friendly" with service providers. Malicious parties cannot cheat as lightweight cryptographic currency accurately tracks nodes' contributions.

While most of my work on cooperative content distribution has focused on leveraging unreliable or untrusted resources, I am not rigid in my approach. Indeed, some of these systems use logically-centralized components, such as the core OASIS infrastructure or, for each file collection in this last system, servers that manage file prices and currency exchange. Rather, I look where it is sensible or economical to leverage available resources—*e.g.*, local bandwidth for CDNs or measurement points for anycast—and architect systems accordingly. Indeed, these same cost arguments are behind industry's increased interest in such architectures, albeit without the same consideration for security.

Securing decentralized systems. When large decentralized systems lack the necessary security mechanisms, things eventually go awry. The Internet's inter-domain routing protocols (BGP) lack source authentication and thus routes have been hijacked, a weakness shared by DNS. Persistent email spam is frustrating, while false positives from spam filters have made email unreliable. Centralized solutions are not the only answer, however.

Tackling the spam false-positive problem, Re: [6] uses proximity in a social network as a basis for auto-whitelisting email. This approach appears promising given our analysis of large email corpora. And by incorporating our cryptographic protocols for private matching [7, 8], Re: ensures that two parties can maintain privacy without third-party intervention.

In a similar vein, websites want to securely identify their users, but ubiquitous client authentication does not exist. Thus, sites often use weaker identifiers such as IP addresses for access-control decisions, even though edge technologies (NATs, proxies, and DHCP) occlude a server's view of its clients. By instrumenting CoralCDN, we used active web content to measure and analyze the characteristics of over 7 million clients; our results help quantify when and how Internet services can use IP addresses and related information to identify clients [9]. (In fact, our techniques for real-time proxy detection and geolocation were acquired by a leading IP analytics company [10].) Here we see how a system, once widely used, can become a vehicle for otherwise infeasible research. Indeed, we are starting to investigate advertisement click fraud using this platform.

Enterprise networks similarly lack comprehensive security "from the ground up." Instead, a bewildering array of mechanisms (firewalls, NATs, and VLANs) have been retrofitted over the years, leading to brittle, inflexible networks. Begun as a clean-slate design [11], Ethane provides a backwards-compatible protection and management architecture for enterprise networks, where switches establish virtual circuits per flow, after using a domain controller to enforce security policies. Because Ethane simplifies so many network management tasks—testing new policies, deploying new appliances or topologies, performing forensics or fault diagnosis, establishing network isolation classes—its architecture empowers innovation and change within networks. I am further interested in extending such techniques to the wider area for managing autonomous systems.

Future work. Given the challenges of securing and managing networked systems, I have begun to think about new ways to simplify this task.

How can we determine when, where, and why performance or persistent faults in distributed systems occur? I intend to explore lightweight distributed tracing to track transactions across hosts and within processes. By tainting network communication and annotating code, we can generate system-wide "call graphs" during run-time. Of particular interest are identifying normal and anomalous system behavior, possibly through machine learning, and building feedback loops for automated reconfiguration. Other approaches to fault monitoring, detection, and diagnosis may be similarly promising. Of course, having deployed systems to test such tools is a critical advantage to experience the vagaries of failures in production environments. (In fact, others have used CoralCDN for exactly this [12].)

What new abstractions can provide better reliability in the face of failures? I am currently thinking about how to partition large systems into smaller groups, which can then apply heavyweight fault-tolerance or detection protocols [13]. (Such partitioning appears necessary for scalability.) While handling malicious parties in dynamic settings presents many difficult problems, the goal remains for better operation on faulty resources.

Finally, what privacy-preserving technologies can promote greater information sharing? Researchers, operators, and end-users can all benefit from greater access to data, whether inter-domain routing policies for traffic engineering, patient records for medical research, census and other polling data for the social sciences, or social information for cooperative filtering [6]. Unfortunately, privacy concerns often limit data availability, leading to suggestions such as private matching [7] for merging terrorist watch lists [14]. Yet current general-purpose cryptographic solutions are too inefficient for large datasets, while statistical methods are often not sound. I am interested in leveraging specific application contexts to build better protocols (as done in [8]), as well as exploring interface and architecture design for privacy-preserving systems.

While technology trends may incrementally improve system performance, new techniques are needed to enhance security, scalability, reliability, and manageability. I tackle these problems by applying methods from cryptography, distributed algorithms, game theory, and other principled sources. But real solutions require real testing: My research will embrace both strong design and engineering components, even as new problems arise over time. This unusual dual approach already has enabled my research systems to provide tens of millions of people with their Internet fix, often in surprising ways. Through such deployments we can discover new problems, encourage further innovation, and ultimately make new functionality broadly available.

References

[1] **M. Freedman**, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. Networked Systems Design and Implementation (NSDI)*, pages 239–252, Mar 2004.

[2] **M. Freedman** and D. Mazières. Sloppy hashing and self-organizing clusters. In *Proc. International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 45–55, Feb 2003.

[3] **M. Freedman**, K. Lakshminarayanan, and D. Mazières. OASIS: Anycast for any service. In *Proc. NSDI*, pages 129–142, May 2006.

[4] S. Annapureddy, **M. Freedman**, and D. Mazières. Shark: Scaling file servers via cooperative caching. In *Proc. NSDI*, pages 129–142, May 2005.

[5] M. Krohn, **M. Freedman**, and D. Mazières. On-the-fly verification of rateless erasure codes for efficient content distribution. In *Proc. IEEE Security and Privacy*, pages 226–240, May 2004.

[6] S. Garriss, M. Kaminsky, **M. Freedman**, B. Karp, D. Mazières, and H. Yu. Re: Reliable email. In *Proc. NSDI*, pages 297–310, May 2006.

[7] **M. Freedman**, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology — EUROCRYPT 2004*, pages 1–19, May 2004.

[8] **M. Freedman** and A. Nicolosi. Efficient private techniques for verifying social proximity. In *Proc. IPTPS*, Feb 2007.

[9] M. Casado and **M. Freedman**. Peering through the shroud: The effect of edge opacity on IP-based client identification. In *Proc. NSDI*, Apr 2007.

[10] Quova. http://www.quova.com/, 2006.

[11] M. Casado, T. Garfinkle, A. Akella, **M. Freedman**, D. Boneh, N. McKeown, and S. Shenker. SANE: A protection architecture for enterprise networks. In *Proc. USENIX Security Symposium*, pages 137–151, Aug 2006.

[12] P. Reynolds, J. Wiener, J. Mogul, M. Aguilera, and A. Vahdat. WAP5: Black-box performance debugging for wide-area systems. In *Proc. WWW*, May 2006.

[13] **M. Freedman**, I. Stoica, D. Mazières, and S. Shenker. Group therapy for systems: Using link-attestations to manage failures. In *Proc. IPTPS*, Feb 2006.

[14] J. Dempsey and P. Rosenzweig. Technologies that can protect privacy as information is shared to combat terrorism. Heritage Foundation Legal Memo #11, May 26 2004.

Teaching statement

Michael J. Freedman

My greatest joy in teaching is helping passionate, hard-working students gain the appropriate tools, knowledge, and skepticism to become independent thinkers and researchers. Given my research interests, this largely translates to sharing my enthusiasm for tackling challenging systems problems and building complete solutions. Designing and building systems requires a broad background in understanding potential approaches, recognizing design tradeoffs, and recalling past successes and failures, much of which can be learned through coursework. But equally critical is the judgment one gains from *doing*: conceptualizing the interplay of various system components, approaches, and often devilish details, and identifying a system's shortcomings through analysis in order to improve it.

My goal, both as an advisor and as a teacher, is to empower students to make their own design decisions and, ultimately, to discover their own interesting problems to tackle. During graduate school, I had the opportunity to supervise research projects for six students, both masters and undergraduates. The challenge was to offer well-defined problems when students needed more supervision, sometimes proposing one or more promising approaches and incremental milestones. Still, I found it important to maintain some vision or open-ended problems that students could work towards.

The students' research experiences helped lead some of them to pursue further graduate education (Robert Soule is now a PhD student at NYU), while it gave others their first experience at writing academic papers (Kevin Shanahan was the first author of a workshop paper on peer-to-peer localization). The most successful outcomes emerged from situations where students ultimately were excited by their research and identified their concrete contributions. Especially motivating were projects that impacted a large audience, *e.g.*, one student built a data collection infrastructure for CoralCDN, knowing that his code would touch data from tens of millions of users. My personal experience has been very similar: My academic highlights from college were the research projects where I played an important role; my worst time was a summer largely spent hacking makefiles written by physicists over two decades.

Beyond supervising independent research, I similarly enjoy teaching students within the classroom setting. I first served as a teaching assistant for the core "Computer System Engineering" course at MIT for two consecutive years. Unfortunately, TAs traditionally only played the role of holding office hours and grading assignments for this course, as faculty taught even recitation sections. Thus, in my second year, I proposed holding an additional weekly small-group tutorial section to help students better learn course concepts and readings—as well as to allow TAs to actually teach—a practice still being done five years later. At NYU, I served as the teaching assistant, lab instructor, and occasional lecturer for the new advanced undergraduate course "Computer Networks," which coupled system programming assignments with academic readings. I also organized and helped teach the MIT Outing Club's month-long winter mountaineering course, which attracted nearly 100 participants. While not academic in nature, the time-intensive experience was gratifying both from my ability to educate others (here, literally, on how to survive) and from deepening my own knowledge in the process. This class, much like project courses, focused on doing, not only on knowing.

Given my research background, I am qualified to teach a variety of courses, including distributed systems, operating or storage systems, security and cryptography, networking, or even software engineering. I am also excited to hold more advanced graduate courses or seminars related to my research areas. I am a strong proponent of project-heavy classes for both advanced undergraduate and graduate students; these go directly towards "hands-on" systems experience and often provide a useful segue into further research.

Finally, I believe that seminars and reading groups play an important role both in staying abreast with the latest research and in learning how to evaluate it critically. At MIT, I helped co-organize an applied security reading group. At NYU, I began a weekly systems seminar and organized it for two years, inviting both outside speakers to present their research and internal volunteers to present others' work. I also served as a student representative at NYU CS faculty meetings, gaining important insight into the concerns and wants of both students and faculty, as well as helping to recruit both new students and faculty to the growing department. The NYU computer science department recognized my contributions with the Henning Biermann award for "outstanding contributions to education and service to the department."

My research statement mentions that I think even academic systems should be user-centric. I am similarly drawn to the eminently "user-centric" nature of teaching. After all, professors ultimately are tasked with producing both research *and* students.

Democratizing content publication with Coral

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

Abstract

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

1 Introduction

The availability of content on the Internet is to a large degree a function of the cost shouldered by the publisher. A well-funded web site can reach huge numbers of people through some combination of load-balanced servers, fast network connections, and commercial content distribution networks (CDNs). Publishers who cannot afford such amenities are limited in the size of audience and type of content they can serve. Moreover, their sites risk sudden overload following publicity, a phenomenon nicknamed the "Slashdot" effect, after a popular web site that periodically links to under-provisioned servers, driving unsustainable levels of traffic to them. Thus, even struggling content providers are often forced to expend significant resources on content distribution.

Fortunately, at least with static content, there is an easy way for popular data to reach many more people than publishers can afford to serve themselves—volunteers can mirror the data on their own servers and networks. Indeed, the Internet has a long history of organizations with good network connectivity mirroring data they consider to be of value. More recently, peer-to-peer file sharing has demonstrated the willingness of even individual broadband users to dedicate upstream bandwidth to redistribute content the users themselves enjoy. Additionally, organizations that mirror popular content reduce their downstream bandwidth utilization and improve the latency for local users accessing the mirror.

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

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

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

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

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

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

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

2 The Coral Content Distribution Network

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

2.1 Usage Models

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

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

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

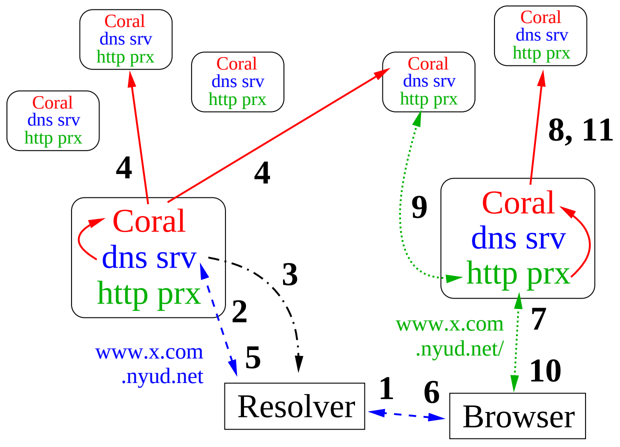


Figure 1: Using CoralCDN, the steps involved in resolving a Coralized URL and returning the corresponding file, per Section 2.2. Rounded boxes represent CoralCDN nodes running Coral, DNS, and HTTP servers. Solid arrows correspond to Coral RPCs, dashed arrows to DNS traffic, dotted-dashed arrows to network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

2.2 System Overview

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

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

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

2.3 The Coral Indexing Abstraction

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

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

Coral provides the following interface to higher-level applications:

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

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

3 Application-Layer Components

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

3.1 The Coral DNS server

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

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

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

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

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

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

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

3.2 The Coral HTTP proxy

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

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

While *CoralProxy* is fetching a web object—either from the origin or from another *CoralProxy*—it inserts a reference to itself in its DSHTs with a time-to-live of 20 seconds. (It will renew this short-lived reference until it completes the download.) Thus, if a flash crowd suddenly fetches a web page, all *CoralProxies*, other than the first simultaneous requests, will naturally form a kind of multicast tree for retrieving the web page. Once any *CoralProxy* obtains the full file, it inserts a much longer-lived reference to itself (*e.g.*, 1 hour). Because the insertion algorithm accounts for TTL, these longer-lived references will overwrite shorter-lived ones, and they can be stored on well-selected nodes even under high insertion load, as later described in Section 4.2.

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

4 Coral: A Hierarchical Indexing System

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

4.1 Coral's Key-Based Routing Layer

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

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

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

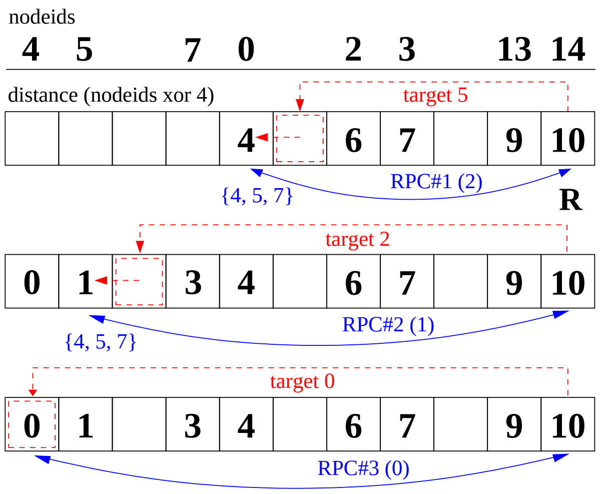


Figure 2: Example of routing operations in a system containing eight nodes with IDs $\{4, 5, 7, 0, 2, 3, 13, 14\}$. In this illustration, node R with $id = 14$ is looking up the node closest to key $k = 4$, and we have sorted the nodes by their distance to k. The top boxed row illustrates XOR distances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

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

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

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

4.2 Sloppy Storage

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

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

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

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

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

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

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

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

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

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

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

Multiple stores of the same key will be spread over multiple nodes. The pointers retrieved by the application are thus distributed among those stored, providing load balancing both *within* Coral and between servers using Coral.

4.3 Hierarchical Operations

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

Figure 3 illustrates Coral's hierarchical routing operations. Each Coral node has the same node ID in all clusters to which it belongs; we can view a node as projecting its presence to the same location in each of its clusters. This structure must be reflected in Coral's basic routing infrastructure, in particular to support switching between a node's distinct DSHTs midway through a lookup.[3]

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

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

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

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

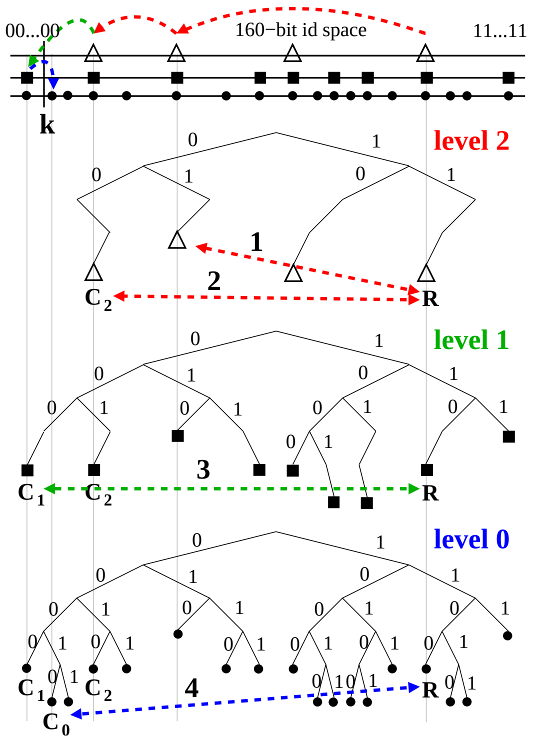


Figure 3: Coral's hierarchical routing structure. Nodes use the same IDs in each of their clusters; higher-level clusters are naturally sparser. Note that a node can be identified in a cluster by its shortest unique ID prefix, *e.g.*, "11" for R in its level-2 cluster; nodes sharing ID prefixes are located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

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

To enable such cluster-aware behavior, the headers of every Coral RPC include the sender's cluster information: the identifier, age, and a size estimate of each of its non-global clusters. The recipient uses this information to demultiplex requests properly, *i.e.*, a recipient should only consider a put and get for those levels on which it shares a cluster with the sender. Additionally, this information drives routing table management: (1) nodes are added or removed from the local cluster-specific routing tables ac-

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

cordingly; (2) cluster information is accumulated to drive cluster management, as described next.

4.4 Joining and Managing Clusters

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

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

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

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

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

Unfortunately, Coral has only rough *approximations* of cluster size, based on its routing-table size. If nearby clusters A and B are of similar sizes, inaccurate estimations could lead to oscillation as nodes flow back-and-forth (although we have not observed such behavior). To perturb an oscillating system into a stable state, Coral employs a preference function δ that shifts every hour. A node selects the larger cluster only if the following holds:

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

where age is the current time minus the cluster's creation time. Otherwise, a node simply selects the cluster with the lower cluster ID.

We use a square wave function for δ that takes a value 0 on an even number of hours and 2 on an odd number. For clusters of disproportionate size, the selection function immediately favors the larger cluster. Otherwise, δ's transition perturbs clusters to a steady state.[4]

In either case, a node that switches clusters still remains in the routing tables of nodes in its old cluster. Thus, old neighbors will still contact it and learn of its new, potentially-better, cluster. This produces an avalanche effect as more and more nodes switch to the larger cluster. This merging of clusters is very beneficial. While a small cluster diameter provides fast lookup, a large cluster capacity increases the hit rate.

5 Implementation

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

6 Evaluation

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

1. CoralCDN dramatically reduces load on servers, solving the "flash crowd" problem.
2. Clustering provides performance gains for popular data, resulting in good client performance.
3. Coral naturally forms suitable clusters.
4. Coral prevents hot spots within its indexing system.

[4]Should clusters of similar size continuously exchange members when δ is zero, as soon as δ transitions, nodes will all flow to the cluster with the lower cluster id. Should the clusters oscillate when $\delta = 2$ (as the estimations "hit" with one around 2^2-times larger), the nodes will all flow to the larger one when δ returns to zero.

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

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

Second, we run an unmodified Apache web server sitting behind a DSL line with 384 Kbit/sec upstream bandwidth, serving 12 different 41KB files, representing groups of three embedded images referenced by four web pages.

Third, we launch client processes on each machine that, after an additional random delay between 0 and 180 seconds for asynchrony, begin making HTTP GET requests to Coralized URLs. Each client generates requests for the group of three files, corresponding to a randomly selected web page, for a period of 30 minutes. While we recognize that web traffic generally has a Zipf distribution, we are attempting merely to simulate a flash crowd to a popular web page with multiple, large, embedded images (*i.e.*, the Slashdot effect). With 166 clients, we are generating 99.6 requests/sec, resulting in a cumulative download rate of approximately $32,800$ Kb/sec. This rate is almost two orders of magnitude greater than the origin web server could handle. Note that this rate was chosen synthetically and in no way suggests a maximum system throughput.

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

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

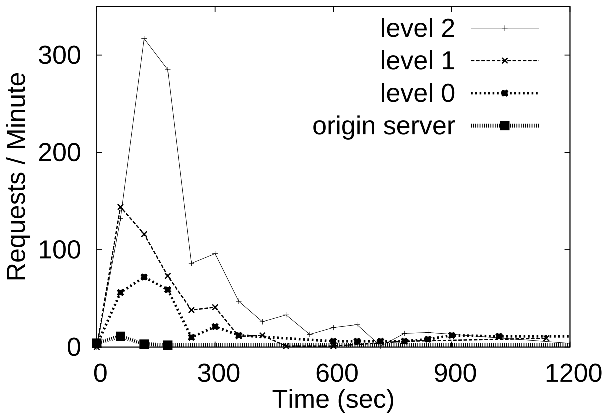


Figure 4: The number of client accesses to *CoralProxies* and the origin HTTP server. *CoralProxy* accesses are reported relative to the cluster level from which data was fetched, and do not include requests handled through local caches.

6.1 Server Load

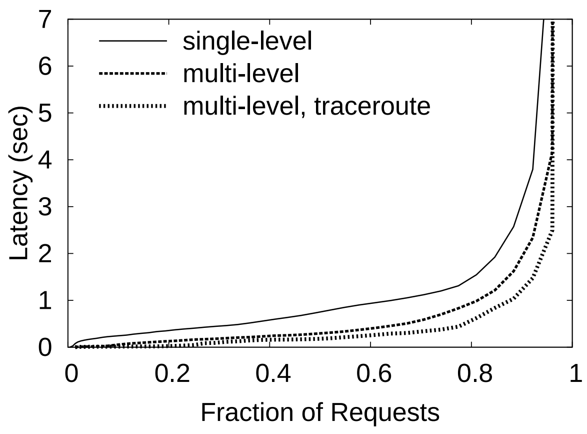
Figure 4 plots the number of requests per minute that could not be handled by a *CoralProxy*'s local cache. During the initial minute, 15 requests hit the origin web server (for 12 unique files). The 3 redundant lookups are due to the simultaneity at which requests are generated; subsequently, requests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

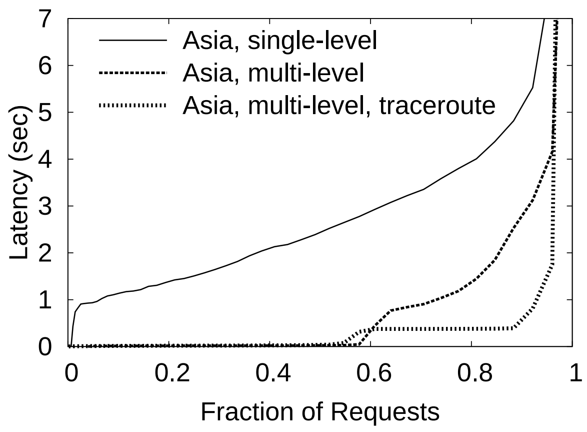
During this first minute, equal numbers of requests were handled by the level-1 and level-2 cluster caches. However, as the files propagated into *CoralProxy* caches, requests quickly were resolved within faster level-2 clusters. Within 8-10 minutes, the files became replicated at nearly every server,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

6.2 Client Latency

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

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





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

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

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

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

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

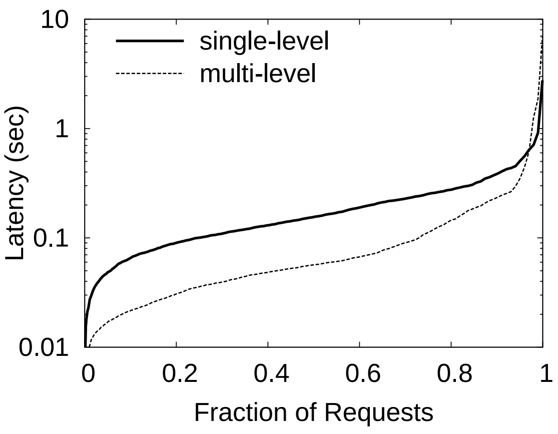


Figure 6: Latencies for proxy to get keys from Coral.

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

6.3 Clustering

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

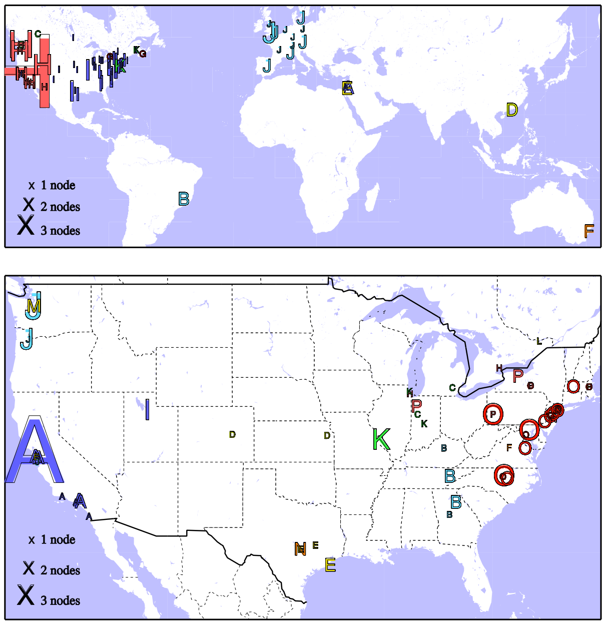


Figure 7: World view of level-1 clusters (60 msec threshold), and United States view of level-2 clusters (20 msec threshold). Each unique, non-singleton cluster is assigned a letter; the size of the letter corresponds to collocated nodes in the same cluster.

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

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

6.4 Load Balancing

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

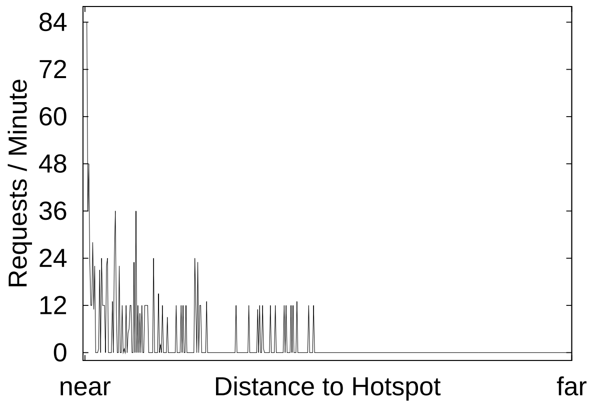


Figure 8: The total number of put RPCs hitting each Coral node per minute, sorted by distance from node ID to target key.

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

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

7 Related work

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

7.1 DHTs and directory services

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

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

Using the network as a directory service, Tapestry [35] and Coral relax the consistency of operations in the network. To put a key, Tapestry routes along fast hops between peers, placing at each peer a pointer back to the sending node, until it reaches the node closest to the key. Nearby nodes routing to the same key are likely to follow similar paths and discover these cached pointers. Coral's flexible clustering provides similar latency-optimized lookup and data placement, and its algorithms prevent multiple stores from forming hot spots. SkipNet also builds a hierarchy of lookup groups, although it explicitly groups nodes by domain name to support organizational disconnect [9].

7.2 Web caching and content distribution

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

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

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

Akamai [1] and other commercial CDNs use DNS redirection to reroute client requests to local clusters of machines, having built detailed maps of the Internet through a combination of BGP feeds and their own measurements, such as traceroutes from numerous vantage points [28]. Then, upon reaching a cluster of collocated machines, hashing schemes [11, 32] map requests to specific machines to increase capacity. These systems require deploying large numbers of highly provisioned servers, and typically result in very good performance (both latency and throughput) for customers.

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

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

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

8 Future Work

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

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

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

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

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

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

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

9 Conclusions

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

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

OASIS: Anycast for Any Service

Michael J. Freedman*‡, Karthik Lakshminarayanan†, David Mazières‡
*New York University, †U.C. Berkeley, ‡Stanford University
http://www.coralcdn.org/oasis/

Abstract

Global anycast, an important building block for many distributed services, faces several challenging requirements. First, anycast response must be fast and accurate. Second, the anycast system must minimize probing to reduce the risk of abuse complaints. Third, the system must scale to many services and provide high availability. Finally, and most importantly, such a system must integrate seamlessly with unmodified client applications. In short, when a new client makes an anycast query for a service, the anycast system must ideally return an accurate reply without performing any probing at all.

This paper presents OASIS, a distributed anycast system that addresses these challenges. Since OASIS is shared across many application services, it amortizes deployment and network measurement costs; yet to facilitate 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. Measurements from a preliminary deployment show that OASIS, surprisingly, provides a significant improvement in the performance that clients experience over state-of-the-art on-demand probing and coordinate systems, while incurring much less network overhead.

1 Introduction

Many Internet services are distributed across a collection of servers that handle client requests. For example, high-volume web sites are typically replicated at multiple locations for performance and availability. Content distribution networks amplify a website's capacity by serving clients through a large network of web proxies. File-sharing and VoIP systems use rendezvous servers to bridge hosts behind NATs.

The performance and cost of such systems depend highly on the servers that clients select. For example, file download times can vary greatly based on the locality and load of the chosen replica. Furthermore, a service provider's costs may depend on the load spikes that the server-selection mechanism produces, as many 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.

More recently, sophisticated techniques for server-selection have been developed. When a legacy client initiates an anycast request, these techniques typically probe the client from a number of vantage points, and then use this information to find the closest server. While efforts, such as virtual coordinate systems [6, 28] and on-demand probing overlays [40, 46], seek to reduce the probing overhead, the savings in overhead comes at the cost of accuracy of the system.

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 have experienced first-hand in a previously deployed system [10]. 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 paper presents OASIS (*O*verlay-based *A*nycast *S*ervice *I*nfra*S*tructure), a shared locality-aware server selection infrastructure. 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 query, "Which server should the client contact?" Selection is primarily optimized for network locality, but also incorporates liveness and load. OASIS can, for instance, be used by CGI scripts to redirect clients to an appropriate web mirror. It can locate servers for IP anycast proxies [2], or it can select distributed SMTP servers in large email services [26].

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

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 1: 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.

techniques that makes it practical to measure the entire Internet in advance. By leveraging the locality of the IP prefixes [12], 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 [34] and has already been adopted by a number of services, including ChunkCast [5], CoralCDN [10], Na Kika [14], OCALA [19], and OpenDHT [37]. Currently, 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.

Experiments from our deployment have shown rather surprisingly that the accuracy of OASIS is competitive with Meridian [46], 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.

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 [11], 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 apriori, 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 [39, 42] 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 [10] 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 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 (§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 (§2.2.2). 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 [36], 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.

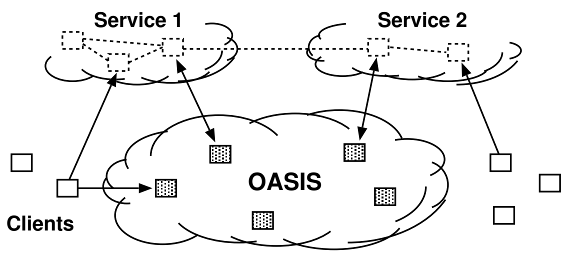


Figure 2: OASIS system overview

2.1 System overview

Figure 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 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 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 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 the case [24].

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,

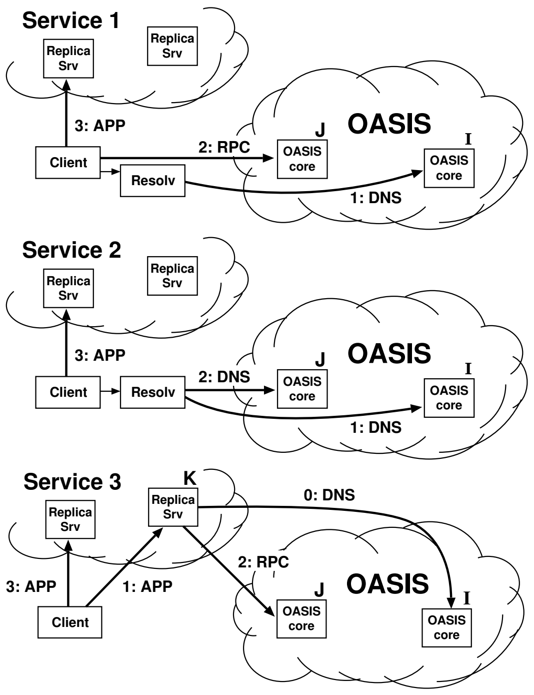


Figure 3: Various methods of using OASIS via its DNS or RPC interfaces, and the steps involved in each anycast request.

then redirects the client. Such an approach does not require that clients be located near their resolvers in order to achieve high accuracy.

This paper largely focuses on DNS redirection, since it is the easiest to integrate with existing applications.

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 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 co-located 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 §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* in-

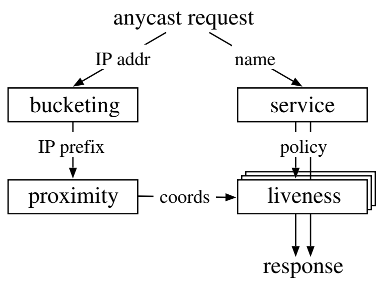


Figure 4: Logical steps to answer an anycast request

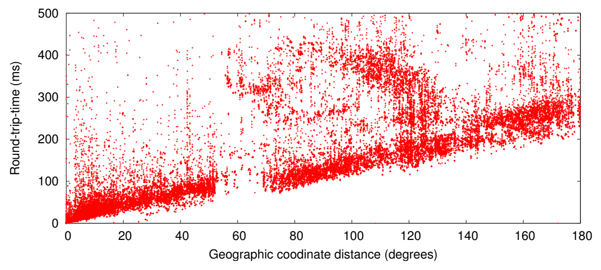


Figure 5: Correlation between round-trip-times and geographic distance across all PlanetLab hosts [43].

cludes all live replicas belonging to the service and their corresponding information (*e.g.*, coordinates, load, and capacity).

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 [23] 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 [12]. For example, 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 [38] as a starting point.[1]

However, some IP prefixes (especially larger prefixes) do not preserve locality [12]. OASIS discovers and adapts to these cases by splitting prefixes that exhibit poor locality precision,[2] an idea originally proposed by IP2Geo [30]. 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 avoid on-demand probing.

2.2.2 Geographic coordinates for location

OASIS takes a two-pronged approach to locate IP prefixes: We first use a direct probing mechanism [46] 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 easily be obtained from a variety of online services [13]. Note that OASIS's shared infrastructure design helps increase the number of vantage points and thus improves its likelihood of having a replica near the prefix.

While geographic coordinates are certainly not optimal predictors of round-trip-times, they work well in practice: The heavy band in Figure 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 [6] and GNP [28] coordinates within OASIS; §5 includes some comparison results.

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 [6, 28] fall short of providing either accuracy or stability [40, 46]. 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

[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 paper.

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

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 [36] 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. 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. OASIS employs a number of tests to detect such situations (see §6). As a final safeguard, however, the core only accepts a prefix-to-coordinate mapping after seeing two consistent measurements from replicas on different networks.

In hindsight, another benefit of geographic coordinates is the ability to couple them with real-time visualization of the network [29], which has helped us identify, debug, and subsequently handle various network peculiarities.

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.

Global membership view. Every OASIS core node maintains a weakly-consistent view of all other nodes in the core, where each node is identified by its IP address, a globally-unique node identifier, and an incarnation number. To avoid $O(n^2)$ probing (where n is the network size), core nodes detect and share failure information cooperatively: every core node probes a random neighbor each time period (3 seconds) and, if it fails to receive a response, gossips its suspicion of failure.

Two techniques suggested by SWIM [7] reduce false failure announcements. First, several intermediates are chosen to probe this target before the initiator announces its suspicion of failure. Intermediaries alleviate the problems caused by non-transitive Internet routing [11]. 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. 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). This approach provides live nodes with sufficient time to respond to and correct false suspicions of failure.

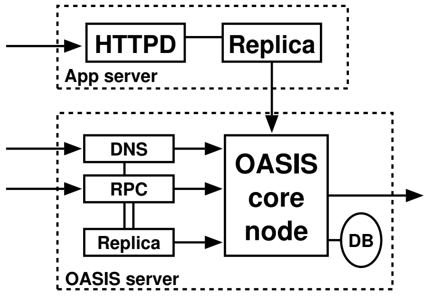


Figure 6: OASIS system components

Implicit in this design is the assumption that nodes are relatively stable; otherwise, the system would incur a high bandwidth cost for failure announcements. Given that OASIS is designed as an *infrastructure service*—to be deployed either by one service provider or a small number of cooperating providers—we believe that this assumption is reasonable.

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, 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 §3.3).

OASIS provides this assignment through consistent hashing [20]. Each node has a random identifier; several nodes with identifiers closest to a key—*e.g.*, the SHA-1 hash of the IP prefix or service name—in the identifier space are assigned the corresponding task. Finding these nodes is easy 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.

Gossiping. OASIS uses gossiping to efficiently disseminate messages—about node failures, service policies, prefix coordinates—throughout the network [7]. Each node maintains a buffer of messages to be piggybacked on other system messages to *random* nodes. Each node gossips each message $O(\log n)$ times for n-node networks; such an epidemic algorithm propagates a message to all nodes in logarithmic time with high probability.[3]

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:

[3]While structured gossiping based on consistent hashing could reduce the bandwidth overhead needed to disseminate a message [3], we use a randomized epidemic scheme for simplicity.

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 6—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.

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

Our implementation differs from [46] 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.

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.

3.1 Managing information

We now describe how OASIS manages the four tables described in §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 responses, and the TTLs of these responses. 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 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 §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 [38], which has approximately 200,000 prefixes. A PATRICIA trie [27] efficiently maps IP addresses to prefixes using longest-prefix matching.

When core nodes modify their bucketing table by splitting or merging prefixes [30], 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, an core node first selects an IP address from within the prefix and issues a probing request to a known replica (or first queries a neighbor to discover one). The selected replica traceroutes the requested IP to find the last routable IP address, performs closest-node discovery using the replica overlay (see §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, 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 §2.2.2).

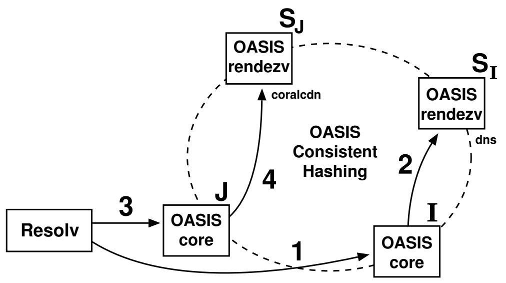


Figure 7: Steps involved in a DNS anycast request to OASIS using rendezvous nodes.

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 fixed as a 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 an 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 [39, 42], 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).

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 7). For simplicity, we limit our discussion to DNS redirection. When a client queries OASIS for the hostname *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:
   (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. §4 gives implementation details.

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 §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, an core node first checks its local liveness table 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 again calculated using coordinate distance, to mirror the same selection criterion used for anycast.) Intuitively, this approach helps prevent situations in which replicas and clients select different co-located nodes and therefore lose the benefit of local information. We analyze the performance benefit of local flooding in §5.1.

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. We do not discuss these further due to space limitations.

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 §5.2.

However, purely load-based metrics cannot be used in conjunction with many of the optimizations that reduce replica lookups to rendezvous nodes (§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.

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 closest-replica requests only from core nodes. It also requires that a replica's overlay neighbors are authorized by OASIS (hence, replicas

```
;; 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 8: Output of `dig` for a hostname using OASIS.

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 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 [25], 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 [27], while the proximity table uses BerkeleyDB [41] for persistent storage.

OASIS's design uses static latitude/longitude coordinates with Meridian overlay probing [46]. For comparison purposes, OASIS also can be configured to use synthetic coordinates using Vivaldi [6] or GNP [28].

RPC and HTTP interfaces. These interfaces take an optional target IP address as input, as opposed to simply using the client's address, in order to support integration of third-party services such as HTTP redirectors (Figure 3). Beyond satisfying normal anycast requests, 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?"[4] In addition to HTML, the HTTP interface supports XML-formatted output for easy visualization using online mapping services [13].

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 signalling by returning CNAME records during core lookups, which map aliases to their *canonical names*.

[4]We plan to support such functionality with DNS TXT records as well, although this has not been implemented yet.

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 [10], which uses the service alias $*.nyud.net$. A resolver R discovers nameservers u, v for $nyud.net$ by querying the root servers for $example.net.nyud.net$.[5] Next, R queries u for this hostname, and is returned a CNAME for $example.net.nyud.net \rightarrow coralcdn.oasis.nyuld.net$ and NS x, y for $coralcdn.oasis.nyuld.net$. In practice, R will 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).

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 8). Thus, when 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. 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*.

5 Evaluation

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

5.1 Wide-area evaluation of OASIS

Experimental setup. We present wide-area measurements on PlanetLab [34] 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.[6]

[5]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 registered as an alias for *coralcdn* in its service policy.

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

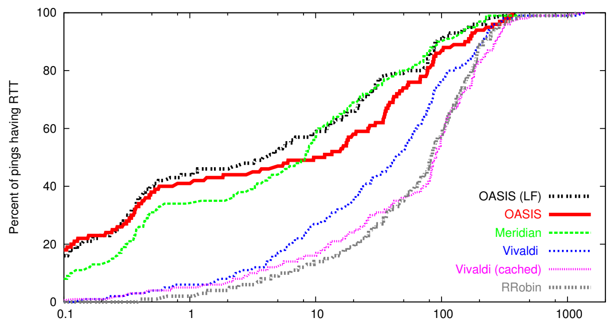


Figure 9: Round trip times (ms)

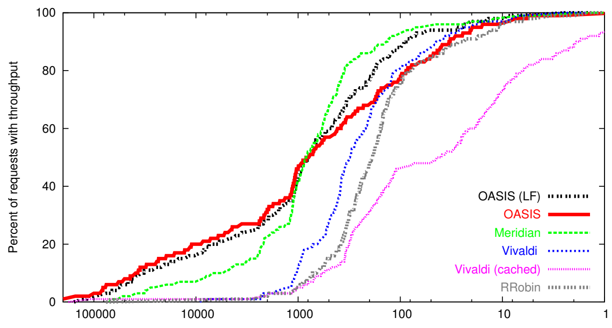


Figure 10: Client-server TCP throughput (KB/s)

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 §3.3). (2) *OASIS* uses only local caching for replicas. (3) *Meridian* (our implementation of [46]) performs on-demand probing by executing closest-replica discovery whenever it receives a request. (4) *Vivaldi* uses 2-dimensional dynamic virtual coordinates [6], 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 for using 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. 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 the following cumulative distribution function (CDF) graphs.

Minimizing RTTs. Figures 9 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.

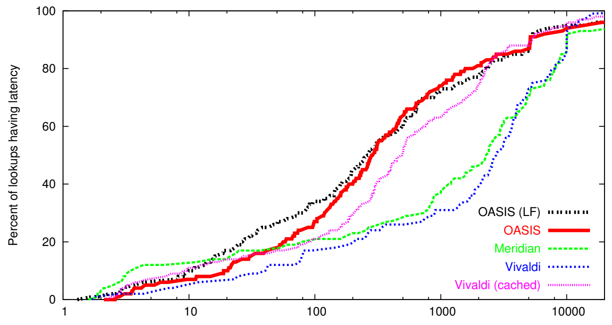


Figure 11: DNS resolution time (ms) for new clients

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 recent work has sought to minimize this instability [8, 33]. 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 (§2.2.2): reprobing by OASIS yields strictly better results, while the accuracy of Meridian queries can vary.

Maximizing throughput. Figure 10 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` [18] in its default configuration (a TCP 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 performance for large data transfers.

DNS resolution time. Figures 11 and 12 evaluate the DNS performance for new clients and for clients already caching their nearby OASIS nameservers, respectively. A request by a new client includes the time to

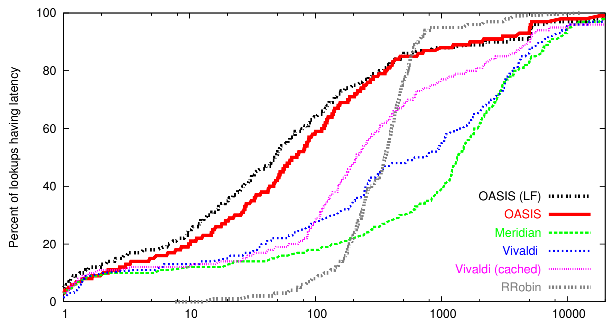


Figure 12: DNS resolution time (ms) for replica lookups

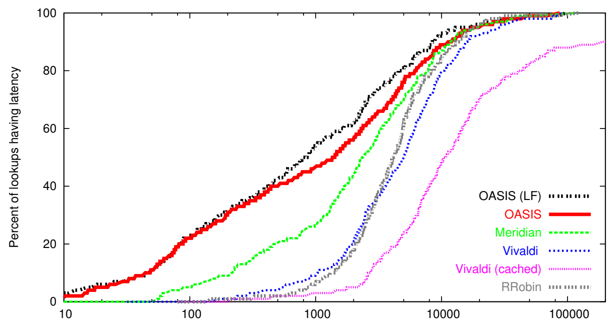


Figure 13: End-to-end download performance (ms)

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), 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 §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 Figure 12.

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*x* faster than that for *Meridian*.[7] 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 13 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 10KB files sequentially. This

[7]A recursive Meridian implementation [46] 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

Table 1: 95th-percentile bandwidth usage (MB)

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 very fast DNS response time with very accurate server selection. Median response-time for OASIS is 290% faster than *Meridian* and 500% faster than simple round-robin systems.

5.2 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 (§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 1MB file, with a 20-second delay between requests. (DNS records have a TTL of 15 seconds.)

Table 1 shows that the webserver with highest bandwidth costs is easily within a factor of two of the least-loaded server. On the other hand, locality-based replica selection creates a traffic spike at a single webserver.

5.3 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* [23]. 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 10 requests per second.

On the other hand, OASIS often uses shorter TTLs to handle replica failover and load balancing. The same datasets showed approximately 100K unique resolvers

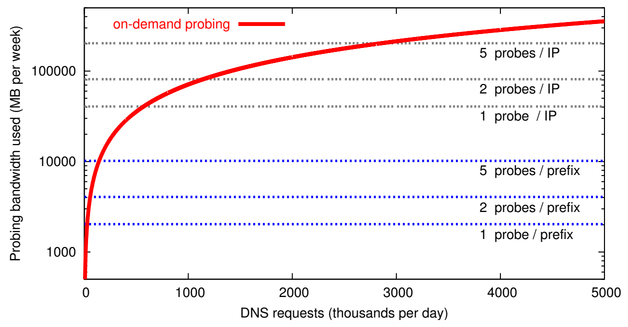


Figure 14: Bandwidth trade-off between on-demand probing, caching IP prefixes (OASIS), and caching IP addresses

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 [10] handles about 20 million HTTP requests from more than one million unique client IPs per day (as of December 2005). To serve this web population, CoralCDN answers 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.

5.4 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, OASIS's approach of probing every IP prefix is not worthwhile. Figure 14 plots the amount of bandwidth used in caching and on-demand anycast systems for a system with 2000 replicas. Following the results of [46], we estimate each closest-replica query to generate about 10.4 KB of network traffic (load grows sub-linearly with the number of replicas).

Figure 14 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 IP addresses, as opposed to 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 [23]. Figure 14 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 12).

Project	Service	Description
ChunkCast [5]	chunkcast	Anycast gateways
CoralCDN [10]	coralcdn	Web proxies
Na Kika [14]	nakika	Web proxies
OASIS	dns	DNS interface
	http	HTTP interface
	rpc	RPC interface
OCALA [19]	ocala	Client IP gateways
	ocalarsp	Server IP gateways
OpenDHT [37]	opendht	Client DHT gateways

Figure 15: Services using OASIS as of March 2006. Services can be accessed using $\langle service \rangle .nyuld.net$.

6 Deployment lessons

OASIS has been deployed on about 250 PlanetLab hosts since November 2005. Figure 15 lists the systems currently using OASIS and a brief description of their service replicas. We 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 §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 ignored.

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 ASes that provide transit for the target, yet filter ICMPs; in such cases, replicas tracerouting the prefix would end up probing the upstream AS.

7 Related work

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 [16]) are based on the triangle-inequality assumption. IDMaps [9] 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 [44], and Iso-bar proposed a two-tier system using landmark-based clustering algorithms [4]. King [15] 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 [28] and Vivaldi [6]) 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 [17, 32], or tracerouting the address space and then mapping router names to locations based on ISP-specific naming conventions [12, 30]. Commercial entities have sought to create exhaustive IP-range mappings [1, 35].

Server selection. IP anycast was proposed as a network-level solution to server selection [22, 31]. 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 [2]; OASIS can serve as a powerful and flexible server-selection backend for such a system.

One of the largest deployed content distribution networks, Akamai [1] 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 [4]. OASIS's task is more difficult than that of commercial CDNs, given its goal of providing anycast for multiple services.

Recent literature has proposed techniques to minimize such exhaustive probing. Meridian [46] (used for DNS redirection by [45]) 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 [21]. 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.

8 Conclusion

OASIS is a global distributed anycast system that allows legacy clients to find nearby or unloaded replicas for distributed services. Two main features distinguish OASIS from prior systems. First, OASIS allows multiple application services to share the anycast service. Second, OASIS avoids on-demand probing when clients initiate requests. Measurements from a preliminary deployment show that OASIS, provides a significant improvement in the performance that clients experience over state-of-the-art on-demand probing and coordinate systems, while incurring much less network overhead.

OASIS's contributions are not merely its individual components, but also the deployed system that is immediately usable by both legacy clients and new services. Publicly deployed on PlanetLab, OASIS has already been adopted by a number of distributed services [5, 10, 14, 19, 37].

Acknowledgments. We thank A. Nicolosi for the keyword analysis of Figure 1. M. Huang, R. Huebsch, and L. Peterson have aided our efforts to run PlanetLab services. We also thank D. Andersen, S. Annapureddy, N. Feamster, J. Li, S. Rhea, I. Stoica, our anonymous reviewers, and our shepherd, S. Gribble, for comments on drafts of this paper. This work was conducted as part of Project IRIS under NSF grant ANI-0225660.

References

[1] Akamai Technologies. http://www.akamai.com/, 2006.

[2] H. Ballani and P. Francis. Towards a global IP anycast service. In *SIGCOMM*, 2005.

[3] M. Castro, M. Costa, and A. Rowstron. Debunking some myths about structured and unstructured overlays. In *NSDI*, May 2005.

[4] Y. Chen, K. H. Lim, R. H. Katz, and C. Overton. On the stability of network distance estimation. *SIGMETRICS Perform. Eval. Rev.*, 30(2):21–30, 2002.

[5] B.-G. Chun, P. Wu, H. Weatherspoon, and J. Kubiatowicz. ChunkCast: An anycast service for large content distribution. In *IPTPS*, Feb. 2006.

[6] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *SIGCOMM*, Aug. 2004.

[7] A. Das, I. Gupta, and A. Motivala. SWIM: Scalable weakly-consistent infection-style process group membership protocol. In *Dependable Systems and Networks*, June 2002.

[8] C. de Launois, S. Uhlig, and O. Bonaventure. A stable and distributed network coordinate system. Technical report, Universite Catholique de Louvain, Dec. 2004.

[9] 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 Trans. on Networking*, Oct. 2001.

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

[11] M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and DHTs. In *WORLDS*, Dec. 2005.

[12] M. J. Freedman, M. Vutukuru, N. Feamster, and H. Balakrishnan. Geographic locality of IP prefixes. In *IMC*, Oct. 2005.

[13] Google Maps. http://maps.google.com/, 2006.

[14] 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 *NSDI*, May 2006.

[15] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary Internet end hosts. In *IMW*, 2001.

[16] J. Guyton and M. Schwartz. Locating nearby copies of replicated Internet servers. In *SIGCOMM*, Aug. 1995.

[17] IP to Lat/Long server, 2005. http://cello.cs.uiuc.edu/cgi-bin/slamm/ip2ll/.

[18] Iperf. Version 1.7.0 – the TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf/, 2005.

[19] D. Joseph, J. Kannan, A. Kubota, K. Lakshminarayanan, I. Stoica, and K. Wehrle. OCALA: An architecture for supporting legacy applications over overlays. In *NSDI*, May 2006.

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

[21] D. R. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *STOC*, 2002.

[22] D. Katabi and J. Wroclawski. A framework for scalable global IP-anycast (GIA). In *SIGCOMM*, Aug. 2000.

[23] K. Keys. Clients of DNS root servers, 2002-08-14. http://www.caida.org/projects/dns-analysis/, 2002.

[24] 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 *USENIX Conference*, June 2002.

[25] D. Mazières. A toolkit for user-level file systems. In *USENIX Conference*, June 2001.

[26] A. Mislove, A. Post, A. Haeberlen, and P. Druschel. Experiences in building and operating a reliable peer-to-peer application. In *EuroSys*, Apr. 2006.

[27] D. Morrison. Practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4), Oct. 1968.

[28] E. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In *INFOCOM*, June 2002.

[29] OASIS. http://www.coralcdn.org/oasis/, 2006.

[30] V. N. Padmanabhan and L. Subramanian. An investigation of geographic mapping techniques for Internet hosts. In *SIGCOMM*, Aug. 2001.

[31] C. Patridge, T. Mendez, and W. Milliken. Host anycasting service. RFC 1546, Network Working Group, Nov. 1993.

[32] D. M. R. Periakaruppan and J. Donohoe. Where in the world is netgeo.caida.org? In *INET*, June 2000.

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

[34] PlanetLab. http://www.planet-lab.org/, 2005.

[35] Quova. http://www.quova.com/, 2006.

[36] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang. BGP routing stability of popular destinations. In *IMW*, Nov. 2002.

[37] 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 *SIGCOMM*, Aug. 2005.

[38] RouteViews. http://www.routeviews.org/, 2006.

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

[40] K. Shanahan and M. J. Freedman. Locality prediction for oblivious clients. In *IPTPS*, Feb. 2005.

[41] Sleepycat. BerkeleyDB v4.2, 2005.

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

[43] J. Stribling. PlanetLab AllPairsPing data, 08-03-2005:11:14:19. http://www.pdos.lcs.mit.edu/ strib/pl_app/, 2005.

[44] W. Theilmann and K. Rothermel. Dynamic distance maps of the Internet. In *IEEE INFOCOM*, Mar 2001.

[45] B. Wong and E. G. Sirer. ClosestNode.com: an open access, scalable, shared geocast service for distributed systems. *SIGOPS OSR*, 40(1), 2006.

[46] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A lightweight network location service without virtual coordinates. In *SIGCOMM*, Aug. 2005.

Efficient Private Matching and Set Intersection

Michael J. Freedman[1*], Kobbi Nissim[2**], and Benny Pinkas[3]

[1] New York University (mfreed@cs.nyu.edu)
[2] Microsoft Research SVC (kobbi@microsoft.com)
[3] HP Labs (benny.pinkas@hp.com)

Abstract. We consider the problem of computing the intersection of private datasets of two parties, where the datasets contain lists of elements taken from a large domain. This problem has many applications for online collaboration. We present protocols, based on the use of homomorphic encryption and balanced hashing, for both semi-honest and malicious environments. For lists of length k, we obtain $O(k)$ communication overhead and $O(k \ln \ln k)$ computation. The protocol for the semi-honest environment is secure in the standard model, while the protocol for the malicious environment is secure in the random oracle model. We also consider the problem of approximating the size of the intersection, show a linear lower-bound for the communication overhead of solving this problem, and provide a suitable secure protocol. Lastly, we investigate other variants of the matching problem, including extending the protocol to the multi-party setting as well as considering the problem of approximate matching.

1 Introduction

This work considers several two-party set-intersection problems and presents corresponding secure protocols. Our protocols enable two parties that each hold a set of inputs – drawn from a *large* domain – to jointly calculate the intersection of their inputs, without leaking any additional information. The set-intersection primitive is quite useful as it is extensively used in computations over databases, *e.g.*, for data mining where the data is vertically partitioned between parties (namely, each party has different attributes referring to the same subjects).

One could envision the usage of efficient set-intersection protocols for online recommendation services, online dating services, medical databases, and many other applications. We are already beginning to see the deployment of such applications using either trusted third parties or plain insecure communication.

Contributions. We study private two-party computation of set intersection, which we also denote as *private matching* (PM):

– Protocols for computing private matching, based on homomorphic encryption and balanced allocations: (i) a protocol secure against semi-honest adversaries; and (ii) a protocol, in the random oracle model, secure against

[*] Research partially done while the author was visiting HP Labs.
[**] Research done while the author was at NEC Labs.

malicious adversaries.[4] Their overhead for input lists of length k is $O(k)$ communication and $O(k \ln \ln k)$ computation, with small constant factors. These protocols are more efficient than previous solutions to this problem.

– Variants of the private matching protocol that (i) compute the intersection *size*, (ii) decide whether the intersection size is greater than a threshold, or (iii) compute some other function of the intersection set.
– We consider private approximation protocols for the intersection size (similar to the private approximation of the Hamming distance by [10]). A simple reduction from the communication lower-bound on disjointness shows that this problem cannot have a sublinear *worst-case* communication overhead. We show a sampling-based private approximation protocol that achieves instance-optimal communication.
– We extend the protocol for set intersection to a multi-party setting.
– We introduce the problem of secure approximate (or "fuzzy") matching and search, and we present protocols for several simple instances.

2 Background and Related Work

Private equality tests (PET). A simpler form of private matching is where each of the two datasets has a single element from a domain of size N. A circuit computing this function has $O(\log N)$ gates, and therefore can be securely evaluated with this overhead. Specialized protocols for this function were also suggested in [9, 18, 17], and they essentially have the same overhead. A solution in [3] provides fairness in addition to security.

A circuit-based solution for computing PM of datasets of length k requires $O(k^2 \log N)$ communication and $O(k \log N)$ oblivious transfers. Another trivial construction compares all combinations of items from the two datasets using k^2 instantiations of a PET protocol (which itself has $O(\log N)$ overhead). The computation of this comparison can be reduced to $O(k \log N)$, while retaining the $O(k^2 \log N)$ communication overhead [18]. There are additional constructions that solve the private matching problem at the cost of only $O(k)$ exponentiations [12, 8]. However, these constructions were only analyzed in the random oracle model, against semi-honest parties.

Disjointness and set intersection. Protocols for computing (or deciding) the intersection of two sets have been researched both in the general context of communication complexity and in the context of secure protocols. Much attention has been given to evaluating the communication complexity of the disjointness problem, where the two parties in the protocol hold subsets a and b of $\{1, \ldots, N\}$. The disjointness function $\textsc{Disj}(a, b)$ is defined to be 1 if the sets a, b have an empty intersection. It is well known that $R_\epsilon(\textsc{Disj}) = \Theta(N)$ [14, 22]. An immediate implication is that computing $|a \cap b|$ requires $\Theta(N)$ communication. Therefore, even without taking privacy into consideration, the communication complexity of private matching is at least proportional to the input size.

[4] For malicious clients, we present a protocol that is secure in the standard model.

One may try and get around the high communication complexity of computing the intersection size by approximating it. In the context of secure protocols, this may lead to a sublinear *private approximation* protocol for intersection size.[5] If one settles for an approximation up to additive error ϵN (for constant ϵ), it is easy to see that very efficient protocols exist, namely $O(\log N)$ bits in the private randomness model [16, Example 5.5]. However, if we require *multiplicative* error (*e.g.*, an (ϵ, δ)-approximation), we show a simple reduction from disjointness that proves that a lower-bound of $\Omega(N)$ communication bits is necessary for any such approximation protocol. See Section 6 for details.

3 Preliminaries

3.1 Private matching (PM)

A **private matching** (PM) scheme is a two-party protocol between a client (chooser) $\mathcal{C}$ and a server (sender) $\mathcal{S}$. $\mathcal{C}$'s input is a set of inputs of size $k_{\mathcal{C}}$, drawn from some domain of size N; $\mathcal{S}$'s input is a set of size $k_{\mathcal{S}}$ drawn from the same domain. At the conclusion of the protocol, $\mathcal{C}$ learns which specific inputs are shared by both $\mathcal{C}$ and $\mathcal{S}$. That is, if $\mathcal{C}$ inputs $X = \{x_1, \ldots, x_{k_\mathcal{C}}\}$ and $\mathcal{S}$ inputs $Y = \{y_1, \ldots, y_{k_\mathcal{S}}\}$, $\mathcal{C}$ learns $X \cap Y$: $\{x_u | \exists v, x_u = y_v\} \leftarrow \mathsf{PM}(X, Y)$.

PM **Variants.** Some variants of the private matching protocol include the following. (i) Private cardinality matching (PM_C) allows $\mathcal{C}$ to learn *how many* inputs it shares with $\mathcal{S}$. That is, $\mathcal{C}$ learns $|X \cap Y|$: $|\mathsf{PM}| \leftarrow \mathsf{PM}_C(X, Y)$. (ii) Private threshold matching (PM_t) provides $\mathcal{C}$ with the answer to the *decisional problem* whether $|X \cap Y|$ is greater than some pre-specified threshold t. That is, $1 \leftarrow \mathsf{PM}_t(X, Y)$ if $\mathsf{PM}_C > t$ and 0 otherwise. (iii) Generalizing PM_C and PM_t, one could define arbitrary private-matching protocols that are simple functions of the intersection set, *i.e.*, based on the output of PM or PM_C.

Private Matching and Oblivious Transfer. We show a simple reduction from oblivious transfer (OT) to private matching. The OT protocol we design is a 1-out-of-2 bit-transfer protocol in the semi-honest case. The sender's input contains two bits b_0, b_1. The chooser's input is a bit σ. At the end of the protocol the chooser learns b_σ and nothing else, while the sender learns nothing.

First, the parties generate their respective PM inputs: The sender generates a list of two strings, $\{0|b_0, 1|b_1\}$, and the chooser generates the list $\{\sigma|0, \sigma|1\}$. Then, they run the PM protocol, at the end of which the chooser learns $\sigma|b_\sigma$. It follows by the results of Impagliazzo and Rudich [13] that there is no black-box reduction of private matching from one-way functions.

Since the reduction is used to show an impossibility result, it is sufficient to show it for the simplest form of OT, as we did above. We note that if one actually wants to build an OT protocol from a PM primitive, it is possible to directly construct a 1-out-of-N bit transfer protocol. In addition, the `PM-Semi-Honest` protocol we describe supports OT of strings.

[5] Informally, a private approximation is an approximation that does not leak information that is not computable given the exact value. See the definition in [10].

3.2 Adversary models

This paper considers both semi-honest and malicious adversaries. Due to space constraints, we only provide the intuition and informal definitions of these models. The reader is referred to [11] for the full definitions.

Semi-honest adversaries. In this model, both parties are assumed to act according to their prescribed actions in the protocol. The security definition is straightforward, particularly as in our case where only one party ($\mathcal{C}$) learns an output. We follow [18] and divide the requirements into (i) protecting the client and (ii) protecting the sender.
***The client's security – indistinguishability*:** Given that the server $\mathcal{S}$ gets no output from the protocol, the definition of $\mathcal{C}$'s privacy requires simply that the server cannot distinguish between cases in which the client has different inputs.
***The server's security – comparison to the ideal model*:** The definition ensures that the client does not get more or different information than the output of the function. This is formalized by considering an ideal implementation where a trusted third party (TTP) gets the inputs of the two parties and outputs the defined function. We require that in the real implementation of the protocol—that is, one without a TTP—the client $\mathcal{C}$ does not learn different information than in the ideal implementation.

Malicious adversaries. In this model, an adversary may behave arbitrarily. In particular, we cannot hope to avoid parties (i) refusing to participate in the protocol, (ii) substituting an input with an arbitrary value, and (iii) prematurely aborting the protocol. The standard security definition (see, *e.g.*, [11]) captures both the correctness and privacy issues of the protocol and is limited to the case in which only one party obtains an output. Informally, the definition is based on a comparison to the ideal model with a TTP, where a corrupt party may give arbitrary input to the TTP. The definition also is limited to the case where at least one of the parties is honest: if $\mathcal{C}$ (resp. $\mathcal{S}$) is honest, then for any strategy that $\mathcal{S}$ (resp. $\mathcal{C}$) can play in the real execution, there is a strategy that it could play in the ideal model, such that the real execution is computationally indistinguishable from execution in the ideal model. We note that main challenge in ensuring security is enforcing the protocol's correctness, rather than its privacy.

3.3 Cryptographic primitives – Homomorphic encryption schemes

Our constructions use a semantically-secure public-key encryption scheme that preserves the group homomorphism of addition and allows multiplication by a constant. This property is obtained by Paillier's cryptosystem [20] and subsequent constructions [21, 7]. That is, it supports the following operations that can be performed without knowledge of the private key: (i) Given two encryptions $\mathsf{Enc}(m_1)$ and $\mathsf{Enc}(m_2)$, we can efficiently compute $\mathsf{Enc}(m_1+m_2)$. (ii) Given some constant c belonging to the same group, we can compute $\mathsf{Enc}(cm)$. We will use the following corollary of these two properties: Given encryptions of the coefficients $a_0, \ldots, a_k$ of a polynomial P of degree k, and knowledge of a plaintext value y, it is possible to compute an encryption of $P(y)$.[6]

4 The Semi-Honest Case

4.1 Private Matching for set intersection (PM)

The protocol follows the following basic structure. $\mathcal{C}$ defines a polynomial P whose roots are her inputs:

$$P(y) = (x_1 - y)(x_2 - y) \ldots (x_{k_\mathcal{C}} - y) = \sum_{u=0}^{k_\mathcal{C}} \alpha_u y^u$$

She sends to $\mathcal{S}$ homomorphic encryptions of the coefficients of this polynomial. $\mathcal{S}$ uses the homomorphic properties of the encryption system to evaluate the polynomial at each of his inputs. He then multiplies each result by a fresh random number r to get an intermediate result, and he adds to it an encryption of the value of his input, *i.e.*, $\mathcal{S}$ computes $\mathsf{Enc}(r \cdot P(y) + y)$. Therefore, for each of the elements in the intersection of the two parties' inputs, the result of this computation is the value of the corresponding element, whereas for all other values the result is random.[7] See Protocol `PM-Semi-Honest`.[8]

4.2 Efficiently evaluating the polynomial

As the computational overhead of exponentiations dominates that of other operations, we evaluate the computational overhead of the protocol by counting exponentiations. Equivalently, we count the number of *multiplications* of the homomorphically-encrypted values by constants (in Step 2(a)), as these multiplications are actually implemented as exponentiations.

Given the encrypted coefficients $\mathsf{Enc}(\alpha_u)$ of a polynomial P, a naive computation of $\mathsf{Enc}(P(y))$ as $\mathsf{Enc}(\sum_{u=0}^{k_\mathcal{C}} y^u \alpha_u)$ results in an overhead of $O(k_\mathcal{C})$ exponentiations, and hence in a total of $O(k_\mathcal{C} k_\mathcal{S})$ exponentiations for `PM-Semi-Honest`.

The computational overhead can be reduced by noting that the input domain is typically much smaller than the modulus used by the encryption scheme.

[6] We neglect technicalities that are needed to make sure the resulting ciphertext hides the sequence of homomorphic operations that led to it. This may be achieved, *e.g.*, by multiplying the result by a random encryption of 1.

[7] This construction can be considered a generalization of the oblivious transfer protocols of [19, 1, 17]. In those, a client retrieving item i sends to the server a predicate which is 0 if and only if $i = j$ where $j \in [N]$.

[8] It is sufficient for Step 3 of the protocol that $\mathcal{C}$ is able to decide whether some ciphertext corresponds to $x \in X$ (*i.e.*, decryption is not necessary). This weaker property is of use if, for example, one uses the El Gamal encryption scheme and encodes an element x by g^x (to allow the homomorphic properties under addition). This may prevent $rP(y) + y$ from being recovered in the decryption process, yet it is easy for $\mathcal{C}$ to decide whether $rP(y) + y = x$. The Paillier [20] homomorphic encryption scheme recovers $rP(y) + y$.

Protocol `PM-Semi-Honest`

INPUT: $\mathcal{C}$'s input is a set $X = \{x_1, \ldots, x_{k_\mathcal{C}}\}$, $\mathcal{S}$'s input is a set $Y = \{y_1, \ldots, y_{k_\mathcal{S}}\}$. The elements in the input sets are taken from a domain of size N.

1. $\mathcal{C}$ performs the following:
   (a) She chooses the secret-key parameters for a semantically-secure homomorphic encryption scheme, and publishes its public keys and parameters. The plaintexts are in a field that contains representations of the N elements of the input domain, but is exponentially larger.
   (b) She uses interpolation to compute the coefficients of the polynomial $P(y) = \Sigma_{u=0}^{k_\mathcal{C}} \alpha_u y^u$ of degree $k_\mathcal{C}$ with roots $\{x_1, \ldots, x_{k_\mathcal{C}}\}$.
   (c) She encrypts each of the $(k_\mathcal{C} + 1)$ coefficients by the semantically-secure homomorphic encryption scheme and sends to $\mathcal{S}$ the resulting set of ciphertexts, $\{\mathsf{Enc}(\alpha_0), \ldots, \mathsf{Enc}(\alpha_{k_\mathcal{C}})\}$.
2. $\mathcal{S}$ performs the following for every $y \in Y$,
   (a) He uses the homomorphic properties to evaluate the encrypted polynomial at y. That is, he computes $\mathsf{Enc}(P(y)) = \mathsf{Enc}(\Sigma_{u=0}^{k_\mathcal{C}} \alpha_u y^u)$. See Section 4.2.
   (b) He chooses a random value r and computes $\mathsf{Enc}(rP(y) + y)$. (One can also encrypt some additional payload data p_y by computing $\mathsf{Enc}(rP(y) + (y|p_y))$. $\mathcal{C}$ obtains p_y iff y is in the intersection.)

   He randomly permutes this set of $k_\mathcal{S}$ ciphertexts and sends the result back to the client $\mathcal{C}$.
3. $\mathcal{C}$ decrypts all $k_\mathcal{S}$ ciphertexts received. She locally outputs all values $x \in X$ for which there is a corresponding decrypted value .

Hence one may encode the values x, y as numbers in the smaller domain. In addition, Horner's rule can be used to evaluate the polynomial more efficiently by eliminating large exponents. This yields a significant (large constant factor) reduction in the overhead.

We achieve a more significant reduction of the overhead by allowing the client to use multiple low-degree polynomials and then allocating input values to polynomials by hashing. This results in reducing the computational overhead to $O(k_\mathcal{C} + k_\mathcal{S} \ln \ln k_\mathcal{C})$ exponentiations. Details follow.

Exponents from a small domain. Let λ be the security parameter of the encryption scheme (*e.g.*, λ is the modulus size). A typical choice is $\lambda = 1024$ or larger. Yet, the input sets are usually of size $\ll 2^\lambda$ and may be mapped into a small domain—of length $n \approx 2\log(\max(kc, ks))$ bits—using pairwise-independent hashing, which induces only a small collision probability. The server should compute $\mathsf{Enc}(P(y))$, where y is n bits long.

Using Horner's rule. We get our first overhead reduction by applying Horner's rule: $P(y) = \alpha_0 + \alpha_1 y + \alpha_2 y^2 + \cdots + \alpha_{k_\mathcal{C}} y^{k_\mathcal{C}}$ is evaluated "from the inside out" as $\alpha_0 + y(\alpha_1 + y(\alpha_2 + y(\alpha_3 + \cdots y(\alpha_{k_\mathcal{C}-1} + y\alpha_{k_\mathcal{C}})\cdots)))$. One multiplies each intermediate result by a *short* y, compared with y^i in the naive evaluation, which results in $k_\mathcal{C}$ *short* exponentiations.

When using the "text book" algorithm for computing exponentiations, the computational overhead is linear in the length of the exponent. Therefore, Horner's rule improves this overhead by a factor of λ/n (which is about 50 for $k_\mathcal{C}, k_\mathcal{S} \approx$

1000). The gain is substantial even when fine-tunes exponentiation algorithms—such as Montgomery's method or Karatsuba's technique—are used.

Using hashing for bucket allocation. The protocol's main computational overhead results from the server computing polynomials of degree $k_{\mathcal{C}}$. We now reduce the degree of these polynomials. For that, we define a process that throws the client's elements into B bins, such that each bin contains at most M elements.

The client now defines a polynomial of degree M for each bin: All items mapped to the bin by some function h are defined to be roots of the polynomial. In addition, the client adds the root $x = 0$ to the polynomial, with multiplicity which sets the total degree of the polynomial to M. That is, if h maps ℓ items to the bin, the client first defines a polynomial whose roots are these ℓ items, and then multiplies it by $x^{M-\ell}$. (We assume that 0 is not a valid input.) The process results in B polynomials, all of them of degree M, that have a total of $k_{\mathcal{C}}$ non-zero roots.

$\mathcal{C}$ sends to $\mathcal{S}$ the encrypted coefficients of the polynomials, and the mapping from elements to bins.[9] For every $y \in Y$, $\mathcal{S}$ finds the bins into which y could be mapped and evaluates the polynomial of those bins. He proceeds as before and responds to $\mathcal{C}$ with the encryptions $rP(y) + y$ for every possible bin allocation for all y.

Throwing elements into bins – balanced allocations. We take the mapping from elements to bins to be a random hash function h with a range of size B, chosen by the client. Our goal is to reduce M, the upper bound on the number of items in a bin. It is well known that if the hash function h maps each item to a random bin, then with high probability (over the selection of h), each bin contains at most $k_{\mathcal{C}}/B + O(\sqrt{(k_{\mathcal{C}}/B)\log B} + \log B)$ elements. A better allocation is obtained using the balanced allocation hashing by Azar et al. [2]. The function h now chooses *two* distinct bins for each item, and the item is mapped into the bin which is *less occupied* at the time of placement. In the resulting protocol, the server uses h to locate the two bins into which y might have been mapped, evaluates both polynomials, and returns the two answers to $\mathcal{C}$.

Theorem 1.1 of [2] shows that the maximum load of a bin is now exponentially smaller: with $1 - o(1)$ probability, the maximum number of items mapped to a bin is $M = (1 + o(1)) \ln\ln B / \ln 2 + \Theta(k_{\mathcal{C}}/B)$. Setting $B = k_{\mathcal{C}}/\ln\ln k_{\mathcal{C}}$, we get $M = O(\ln\ln k_{\mathcal{C}})$.

A note on correctness and on constants. One may worry about the case that $\mathcal{C}$ is unlucky in her choice of h such that more than M items are mapped to some bin. The bound of [2] only guarantees that this happens with probability $o(1)$. However, Broder and Mitzenmacher [4] have shown that asymptotically, when we map n items into n bins, the number of bins with i or more items falls approximately like $2^{-2.6^i}$. This means that a bound of $M = 5$ suffices with probability 10^{-58}. Furthermore, if the hashing searches for the emptiest in three bins, then $M = 3$ suffices with probability of about 10^{-33}. The authors also

[9] For our purposes, it is sufficient that the mapping is selected pseudo-randomly, either jointly or by either party.

provide experimental results that confirm the asymptotic bound for the case of $n = 32,000$. We conclude that we can bound $\ln \ln k_{\mathcal{C}}$ by a small constant in our estimates of the overhead. Simple experimentation can provide finer bounds.

Efficiency. The communication overhead, and the computation overhead of the client, are equal to the total number of coefficients of the polynomials. This number, given by $B \cdot M$, is $O(k_{\mathcal{C}})$ if $B = k_{\mathcal{C}} / \ln \ln k_{\mathcal{C}}$. If $k \leq 2^{24}$, then using $B = k_{\mathcal{C}}$ bins implies that the communication overhead is at most 4 times that of the protocol that does not use hashing.

The server computes, for each item in his input, M exponentiations with a small exponent, and one exponentiation with a full-length exponent (for computing $r \cdot P(y)$). Expressing this overhead in terms of full-length exponentiations yields an overhead of $O(k_{\mathcal{S}} + k_{\mathcal{S}} \frac{\ln \ln k_{\mathcal{C}} \cdot n}{\lambda})$ for $B = k_{\mathcal{C}} / \ln \ln k_{\mathcal{C}}$. In practice, the overhead of the exponentiations with a small exponent has little effect on the total overhead, which is dominated by $k_{\mathcal{S}}$ full-length exponentiations.

4.3 Security of `PM-Semi-Honest`

We state the claims of security for PM in the semi-honest model.

Lemma 1 (Correctness). *Protocol* `PM-Semi-Honest` *evaluates the* PM *function with high probability.*

(The proof is based on the fact that the client receives an encryption of y for $y \in X \cap Y$, and an encryption of a random value otherwise.)

Lemma 2 ($\mathcal{C}$'s privacy is preserved). *If the encryption scheme is semantically secure, then the views of $\mathcal{S}$ for any two inputs of $\mathcal{C}$ are indistinguishable.*

(The proof uses the fact that the only information that $\mathcal{S}$ receives consists of semantically-secure encryptions.)

Lemma 3 ($\mathcal{S}$'s privacy is preserved). *For every client $\mathcal{C}^*$ that operates in the real model, there is a client $\mathcal{C}$ operating in the ideal model, such that for every input Y of $\mathcal{S}$, the views of the parties $\mathcal{C}, \mathcal{S}$ in the ideal model is indistinguishable from the views of $\mathcal{C}^*, \mathcal{S}$ in the real model.*

(The proof defines a polynomial whose coefficients are the plaintexts of the encryptions sent by $\mathcal{C}^*$ to $\mathcal{S}$. The $k_{\mathcal{C}}$ roots of this polynomial are the inputs that $\mathcal{C}$ sends to the trusted third party in the ideal implementation.)

Security of the hashing-based protocol. Informally, the hashing-based protocol preserves $\mathcal{C}$'s privacy since (i) $\mathcal{S}$ still receives semantically-secure encryptions, and (ii) the key is chosen independently of $\mathcal{C}$'s input. Thus, neither the key nor h reveal any information about X to $\mathcal{S}$. The protocol preserves $\mathcal{S}$'s privacy since the total number of non-zero roots of the polynomials is $k_{\mathcal{C}}$.

4.4 Variant: Private Matching for set cardinality (PM_C)

In a protocol for private cardinality matching, $\mathcal{C}$ should learn the cardinality of $X \cap Y$, but not the actual elements of this set. $\mathcal{S}$ needs only slightly change his behavior from that in Protocol `PM-Semi-Honest` to enable this functionality. Instead of encoding y in Step 2(b), $\mathcal{S}$ now only encodes some "special" string, such as a string of 0's, *i.e.*, $\mathcal{S}$ computes $\mathsf{Enc}(rP(y) + 0^+)$. In Step 3 of the protocol, $\mathcal{C}$ counts the number of ciphertexts received from $\mathcal{S}$ that decrypt to the string 0^+ and locally outputs this number c. The proof of security for this protocol trivially follows from that of `PM-Semi-Honest`.

4.5 Variants: Private Matching for cardinality threshold (PM_t) and other functions

In a protocol for private threshold matching, $\mathcal{C}$ should only learn whether $c = |X \cap Y| > t$. To enable this functionality, we change `PM-Semi-Honest` as follows. (i) In Step 2(b), $\mathcal{S}$ encodes random numbers instead of y in PM (or 0^+ in PM_C). That is, he computes $\mathsf{Enc}(rP(y)+r_y)$, for random r_y. (ii) Following the basic PM protocol, $\mathcal{C}$ and $\mathcal{S}$ engage in a secure circuit evaluation protocol. The circuit takes as input $k_{\mathcal{S}}$ values from each party: $\mathcal{C}$'s input is the ordered set of plaintexts she recovers in Step 3 of the PM protocol. $\mathcal{S}$'s input is the list of random payloads he chooses in Step 2(b), in the same order he sends them. The circuit first computes the equality of these inputs bit-by-bit, which requires $k_{\mathcal{S}}\lambda'$ gates, where λ' is a statistical security parameter. Then, the circuit computes a threshold function on the results of the $k_{\mathcal{S}}$ comparisons.

Hence, the threshold protocol has the initial overhead of a PM protocol plus the overhead of a secure circuit evaluation protocol. Note, however, that the overhead of circuit evaluation is not based on the input domain of size N. Rather, it first needs to compute equality on the input set of size $k_{\mathcal{S}}$, then compute some simple function of the size of the *intersection set*. In fact, this protocol can be used to compute any function of the intersection set, *e.g.*, check if c within some range, not merely the threshold problem.

5 Security against Malicious Parties

We describe modifications to our PM protocol in order to provide security in the malicious adversary model. Our protocols are based on protocol `PM-Semi-Honest`, optimized with the balanced allocation hashing.

We first deal with malicious clients and then with malicious servers. Finally, we combine these two protocols to achieve a protocol in which either party may behave adversarially. We take this non-standard approach as: (i) It provides conceptual clarity as to the security concerns for each party; (ii) These protocols may prove useful in varying trust situations, *e.g.*, one might trust a server but not the myriad clients; and (iii) The client protocol is secure in the standard model, while the server protocol is analyzed in the random oracle model.

Protocol `PM-Malicious-Client`

INPUT: $\mathcal{C}$ has input X of size $k_{\mathcal{C}}$, and $\mathcal{S}$ has input Y of size $k_{\mathcal{S}}$, as before.

1. $\mathcal{C}$ performs the following:
   (a) She chooses a key for a pseudo-random function that realizes the balanced allocation hash function h, and she sends it to $\mathcal{S}$.
   (b) She chooses a key s for a pseudo-random function F and gives each item x in her input X a new pseudo-identity, $F_s(G(x))$, where G is a collision-resistant hash function.
   (c) For each of her polynomials, $\mathcal{C}$ first sets roots to the pseudo-identities of such inputs that were mapped to the corresponding bin. Then, she adds a sufficient number of 0 roots to set the polynomial's degree to M.
   (d) She repeats Steps (b),(c) for L times to generate L copies, using a different key s for F in each iteration.
2. $\mathcal{S}$ asks $\mathcal{C}$ to open $L/2$ of the copies.
3. $\mathcal{C}$ opens the encryptions of the coefficients of the polynomials for these $L/2$ copies to $\mathcal{S}$, but does not reveal the associated keys s. Additionally, $\mathcal{C}$ sends the keys s used in the unopened $L/2$ copies.
4. $\mathcal{S}$ verifies that the each opened copy contains $k_{\mathcal{C}}$ roots. If this verification fails, $\mathcal{S}$ halts. Otherwise, $\mathcal{S}$ uses the additional $L/2$ keys he receives, along with the hash function G, to generate the pseudo-identities of his inputs. He runs the protocol for each of the polynomials. However, for an input y, rather than encoding y as the payload for each polynomial, he encodes $L/2$ random values whose exclusive-or is y.
5. $\mathcal{C}$ receives the results, organized as a list of $k_{\mathcal{S}}$ sets of size $L/2$. She decrypts them, computes the exclusive-or of each set, and compares it to her input.

5.1 Malicious clients

To ensure security against a malicious client $\mathcal{C}$, it must be shown that for any possible client behavior in the real model, there is an input of size $k_{\mathcal{C}}$ that the client provides to the TTP in the ideal model, such that his view in the real protocol is efficiently simulatable from his view in the ideal model.

We first describe a simple solution for the implementation that does not use hashing. We showed in Lemma 3 that if a value y is not a root of the polynomial sent by the client, the client cannot distinguish whether this item is in the server's input. Thus, we have to take care of the possibility that $\mathcal{C}$ sends the encryption of a polynomial with more than $k_{\mathcal{C}}$ roots. This can only happen if all the encrypted coefficients are zero (P's degree is indeterminate). We therefore modify the protocol to require that at least one coefficient is non-zero – in Step 1(b) of Protocol `PM-Semi-Honest`, $\mathcal{C}$ generates the coefficients of P with α_0 set to 1, then sends encryptions of the other coefficients to $\mathcal{S}$.

In the protocol that uses hashing, $\mathcal{C}$ sends encryptions of the coefficients of B polynomials (one per bin), each of degree M. The server must ensure that the *total* number of roots (different than 0) of these polynomials is $k_{\mathcal{C}}$. For that we use a cut-and-choose method, as shown in Protocol `PM-Malicious-Client`. With overhead L times that of the original protocol, we get error probability that is exponentially small in L.

Proof. (sketch) In our given cut-and-choose protocol, note that $\mathcal{C}$ learns about an item iff it is a root of all the $L/2$ copies evaluated by $\mathcal{S}$. Therefore, to learn about more than $k_{\mathcal{C}}$ items, she must have $L/2$ copies such that each has more than $k_{\mathcal{C}}$ roots. The probability that all such polynomials are not checked by $\mathcal{S}$ is exponentially small in L. This argument can be used to show that, for every adversarial $\mathcal{C}*$ whose success probability is not exponentially small, there is a corresponding $\mathcal{C}$ in the ideal model whose input contains at most $k_{\mathcal{C}}$ items.[10]

5.2 Malicious servers

Protocol `PM-Semi-Honest` of Section 4 enables a malicious server to attack the protocol *correctness*.[11] He can play tricks like encrypting the value $r \cdot (P(y) + P(y')) + y''$ in Step 2(b), so that $\mathcal{C}$ concludes that y'' is in the intersection set iff both y and y' are X. This behavior does not correspond to the definition of PM in the ideal model. Intuitively, this problem arises from $\mathcal{S}$ using two "inputs" in the protocol execution for input y—a value for the polynomial evaluation, and a value used as a payload—whereas $\mathcal{S}$ has a single input in the ideal model.[12]

We show how to modify Protocol `PM-Semi-Honest` to gain security against malicious servers. The protocol based on balanced allocations may be modified similarly. Intuitively, we force the server to run according to its prescribed procedure. Our construction, `PM-Malicious-Server`, is in the random oracle model.

The server's privacy is preserved as in `PM-Semi-Honest`: The pair (e, h) is indistinguishable from random whenever $P(y) \neq 0$. The following lemma shows that the client security is preserved under malicious server behavior.

Lemma 4 (Security for the client). *For every server $\mathcal{S}^*$ that operates in the real model, there is a server $\mathcal{S}$ operating in the ideal model, such that the views of the parties $\mathcal{C}, \mathcal{S}$ in the ideal model is computationally indistinguishable from the views of $\mathcal{C}, \mathcal{S}^*$ in the real model.*

Proof. (sketch) We describe how $\mathcal{S}$ works.

1. $\mathcal{S}$ generates a secret-key/public-key pair for the homomorphic encryption scheme, chooses a random polynomial $P(y)$ of degree $k_{\mathcal{C}}$ and gives $\mathcal{S}^*$ his encrypted coefficients. Note that $\mathcal{S}^*$ does not distinguish the encryption of $P(y)$ from the encryption of any other degree $k_{\mathcal{C}}$ polynomial.
2. $\mathcal{S}$ records all the calls $\mathcal{S}^*$ makes to the random oracles H_1, H_2. Let $\hat{S}$ be the set of input values to H_1 and $\hat{Y}$ be the set of y input values to H_2.

[10] In the proof, the pseudo-random function F hides from $\mathcal{S}$ the identities of the values corresponding to the roots of the opened polynomials. The collision-resistant hash function G prevents $\mathcal{C}$ from setting a root to which $\mathcal{S}$ maps two probable inputs.

[11] He cannot affect $\mathcal{C}$'s privacy as all the information $\mathcal{C}$ sends is encrypted via a semantically-secure encryption scheme.

[12] Actually, the number of "inputs" is much higher, as $\mathcal{S}$ needs to be consistent in using the same y for all the steps of the polynomial-evaluation procedure.

Protocol `PM-Malicious-Server`

INPUT: $\mathcal{C}$ has input X of size $k_{\mathcal{C}}$, and $\mathcal{S}$ has input Y of size $k_{\mathcal{S}}$, as before.
RANDOM ORACLES: H_1, H_2.

1. $\mathcal{C}$ performs the following:
   (a) She chooses a secret-key/public-key pair for the homomorphic encryption scheme, and sends the public-key to $\mathcal{S}$.
   (b) She generates the coefficients of a degree $k_{\mathcal{C}}$ polynomial P whose roots are the values in X. She sends to $\mathcal{S}$ the encrypted coefficients of P.
2. $\mathcal{S}$ performs the following for every $y \in Y$,
   (a) He chooses a random s and computes $r = H_1(s)$. We use r to "derandomize" the rest of $\mathcal{S}$'s computation for y, and we assume that it is of sufficient length.
   (b) He uses the homomorphic properties of the encryption scheme to compute $(e, h) \leftarrow (\mathsf{Enc}(r' \cdot P(y) + s), H_2(r'', y))$. In this computation, r is parsed to supply r', r'' and all the randomness needed in the computation.

   $\mathcal{S}$ randomly permutes this set of $k_{\mathcal{S}}$ pairs and sends it to $\mathcal{C}$.
3. $\mathcal{C}$ decrypts all the $k_{\mathcal{S}}$ pairs she received. She performs the following operations for every pair (e, h),
   (a) She decrypts e to get $\hat{s}$ and computes $\hat{r} = H_1(\hat{s})$.
   (b) She checks whether, for some $x \in X$, the pair (e, h) is consistent with x and $\hat{s}$. That is, whether the server yields (e, h) using her encrypted coefficients on $y \leftarrow x$ and randomness $\hat{r}$. If so, she puts x in the intersection set.

3. For every output pair (e, h) of $\mathcal{S}^*$, $\mathcal{S}$ checks whether it agrees with some $\hat{s} \in \hat{S}$ and $\hat{y} \in \hat{Y}$. We call such a pair a consistent pair. That is, $\mathcal{S}$ checks that (i) e is a ciphertext resulting from applying the server's prescribed computation using the encrypted coefficients, the value $\hat{y}$, and randomness r'; and (ii) $h = H_2(r'', \hat{y})$, where r', r'' and the randomness in the computation are determined by $H_1(\hat{s})$. If such consistency does occur, $\mathcal{S}$ sets $y = \hat{y}$, otherwise it sets $y = \perp$.
4. $\mathcal{S}$ sends the values y it computed to the TTP, and $\mathcal{S}$ outputs the same output as $\mathcal{S}^*$ in the real model.

It is easy, given the view of $\mathcal{S}^*$, to decide whether a pair is consistent. As $\mathcal{S}^*$ cannot distinguish the input fed to it by $\mathcal{S}$ from the input it receives from $\mathcal{C}$ in the real execution, we get that S^*'s distributions on consistent and inconsistent pairs, when run by the simulator and in the real execution, are indistinguishable.

Whenever (e, h) forms an inconsistent pair, giving an invalid symbol $\perp$ as input to the TTP does not affect its outcome. Let (e, h) be a consistent pair, and let y be the value that is used in its construction. In the real execution, $y \in X$ would result in adding y to the intersection set, and this similarly would happen in the simulation. The event that, in the real execution, an element $x \neq y$ would be added to the intersection set occurs with negligible probability.

We get that the views of the parties $\mathcal{C}, \mathcal{S}$ in the ideal model is computationally indistinguishable from the views of $\mathcal{C}, \mathcal{S}^*$ in the real model, as required.

5.3 Handling both malicious clients and servers

We briefly describe how to *combine* these two schemes yield a PM protocol fully secure in the malicious model. We leave the detailed description to the full version of this paper.

$\mathcal{C}$ generates B bins as before; for each bin B_i, she generates a polynomial of degree M with $P(z) = 0$, where $z \in B_i$ if it is (1) mapped to B_i by our hashing scheme (for $z = F_s(G(x))$ for $x \in X$) or (2) added as needed to yield M items. The latter should be set outside the range of F_s. For each polynomial, $\mathcal{C}$ prepares L copies and sends their commitments to $\mathcal{S}$.

Next, $\mathcal{S}$ opens the encryptions of $L/2$ copies and verifies them. If verification succeeds, $\mathcal{S}$ opens the F_s used in the other $L/2$ copies. He chooses a random s, splits it into $L/2$ shares, and then acts as in `PM-Malicious-Server`, albeit using the random shares as payload, $H_1(s)$ as randomness, and appending $H_2(r'', y)$.

Finally, $\mathcal{C}$ receives a list of the unopened $L/2$ copies. For each, she computes candidates for s's shares and recovers s from them. She uses a procedure similar to `PM-Malicious-Server` to check the consistency of the these $L/2$ shares.

6 Approximating Intersection

In this section, we focus on a problem related to private matching: set intersection and its approximation. Assume $\mathcal{C}$ and $\mathcal{S}$ hold strings X and Y respectively, where $|X| = |Y| = N$. Define $\textsc{Intersect}(X, Y) = |\{i : X_i = Y_i\}|$. Equivalently, $\textsc{Intersect}(X, Y)$ is the scalar product of X, Y. Let $0 < \epsilon, \delta$ be constants. An (ϵ, δ)-approximation protocol for intersection yields, on inputs X, Y, a value $\hat{\alpha}$ such that $\Pr[(1-\epsilon)\alpha < \hat{\alpha} < (1+\epsilon)\alpha] \geq 1-\delta$ where $\alpha = |X \cap Y|$. The probability is taken over the randomness used in the protocol.

A lower bound. Let $0 < \eta \leq N$. It is easy to see that an (ϵ, δ)-approximation may be used for distinguishing the cases $|X \cap Y| \leq \eta$ and $|X \cap Y| \geq \eta(1+\epsilon)^2$, as (with probability $1-\delta$) its output is less than $\eta(1+\epsilon)$ in the former case and greater than $\eta(1+\epsilon)$ in the latter.

A protocol that distinguishes $|X \cap Y| \leq \eta$ and $|X \cap Y| \geq \eta(1+\epsilon)$ may be used for deciding disjointness, as defined in Section 2. Given inputs a, b of length m for $\textsc{Disj}$, $\mathcal{C}$ sets her input to be $X = 1^\eta | a^{(2\epsilon+\epsilon^2)\eta}$ (*i.e.*, η ones followed by $(2\epsilon + \epsilon^2)\eta$ copies of a). Similarly, $\mathcal{S}$ sets $Y = 1^\eta | b^{(2\epsilon+\epsilon^2)\eta}$. The length of these new inputs is $N = |X| = |Y| = \eta + (2\epsilon + \epsilon^2)\eta m$ bits. Note that if a, b are disjoint, then $|X \cap Y| = \eta$; otherwise, $|X \cap Y| \geq \eta(1+\epsilon)^2$. Hence, for constant ϵ, it follows that the randomized communication complexity of distinguishing the two cases is at least $\Omega(m) = \Omega(N/\eta)$. By setting η to a constant, we get that the randomized communication complexity of an (ϵ, δ) approximation for $\textsc{Intersect}$ is $\Theta(N)$.

A private approximation protocol for intersection. We describe a protocol for the semi-honest case. Informally, a protocol realizes a private approximation to a function $f(X, Y)$ if it computes an approximation to $f(X, Y)$ and does not leak any information that is not efficiently computable from $f(X, Y)$. This

Protocol `Private-Sample-`B

INPUT: $\mathcal{C}$ and $\mathcal{S}$ hold N-bit strings X, Y, respectively.

1. $\mathcal{C}$ picks a random mask $m_{\mathcal{C}} \in_R \{0,1\}$ and shift amount $r_{\mathcal{C}} \in_R [N]$. She computes the N-bit string $X' = (X \ll r_{\mathcal{C}}) \oplus m_{\mathcal{C}}$ (*i.e.*, she shifts X cyclicly $r_{\mathcal{C}}$ positions and XORs every location in the resulting string with $m_{\mathcal{C}}$). Similarly, $\mathcal{S}$ picks $m_{\mathcal{S}}, r_{\mathcal{S}}$ and computes $Y' = (Y << r_{\mathcal{S}}) \oplus m_{\mathcal{S}}$.
2. $\mathcal{C}$ and $\mathcal{S}$ invoke two $\binom{N}{1}$-OT protocols where $\mathcal{C}$ retrieves $s_{\mathcal{C}} = Y'_{r_{\mathcal{C}}}$ and $\mathcal{S}$ retrieves $s_{\mathcal{S}} = X'_{r_{\mathcal{S}}}$.
3. $\mathcal{C}$ computes $T_{00} = B(m_{\mathcal{C}}, s_{\mathcal{S}}), T_{01} = B(m_{\mathcal{C}}, s_{\mathcal{S}} \oplus 1), T_{10} = B(m_{\mathcal{C}} \oplus 1, s_{\mathcal{S}}), T_{11} = B(m_{\mathcal{C}} \oplus 1, s_{\mathcal{S}} \oplus 1)$.
4. $\mathcal{C}$ and $\mathcal{S}$ invoke a $\binom{4}{1}$-OT protocol where $\mathcal{S}$ retrieves $T_{m_{\mathcal{S}}, s_{\mathcal{S}}}$. $\mathcal{S}$ sends $T_{m_{\mathcal{S}}, s_{\mathcal{S}}}$ back to $\mathcal{C}$.

is formulated by the requirement that each party should be able to simulate her view given her input and $f(X,Y)$. We refer the reader to [10] for the formal definition.

Our building block – protocol `Private-Sample-`B – is a simple generalization of the private sampler of [10]. `Private-Sample-`B samples a random location ℓ and checks if a predicate B holds on (X_ℓ, Y_ℓ). The location ℓ is shared by $\mathcal{C}$ and $\mathcal{S}$ as $\ell = r_{\mathcal{C}} + r_{\mathcal{S}} \pmod N$, with each party holding one of the random shares $r_{\mathcal{C}}, r_{\mathcal{S}}$ at the end of Step 1. Step 2 results in $\mathcal{C}$ and $\mathcal{S}$ holding random shares of $X_\ell = m_{\mathcal{C}} \oplus s_{\mathcal{S}}$ and $Y_\ell = m_{\mathcal{S}} \oplus s_{\mathcal{C}}$. Finally, both parties learn $B(m_{\mathcal{C}} \oplus s_{\mathcal{C}}, m_{\mathcal{S}} \oplus s_{\mathcal{S}}) = B(X_\ell, Y_\ell)$.

It is easy to see that the views of $\mathcal{C}$ and $\mathcal{S}$ in Protocol `Private-Sample-`B are simulatable given $v = |\{i : B(X_i, Y_y)\}|$. It follows that any approximation based on the outcome of the protocol is a *private approximation* for v.

The communication costs of `Private-Sample-`B are dominated by the cost of the $\binom{N}{1}$-OT protocol in use. Naor and Pinkas [19] showed how to combine a $\binom{N}{1}$-OT protocol with any computational PIR scheme, under the DDH assumption. Combining this result with PIR scheme of Cachin et al. [5] (or of Kiayias and Yung [15]) results in $\lambda\ \mathsf{polylog}(N)$ communication, for security parameter λ.

Our protocol `Intersect-Approx` repeatedly invokes `Private-Sample-`B with $B(\alpha,\beta) = \alpha \wedge \beta$, for a maximum of M invocations. We call an invocation *positive* if it concludes with B evaluated as 1. If T invocations occur in $t < M$ rounds, the protocol outputs T/t and halts. Otherwise (after M invocations) the protocol outputs 0 and halts.

The random variable t is the sum of T independent geometric random variables. Hence, $\mathbf{E}[t] = T/p$ and $\mathbf{Var}[t] = T(1-p)/p^2$, where $p = v/N$. Using the Chebyshev Inequality, we get that $\Pr\left[|t - T/p| \geq \beta T/p\right] \leq \left(T\frac{1-p}{p^2}\right) \Big/ \left((\beta\frac{T}{p})^2\right) \leq \frac{1}{\beta^2 T}$. Let $\beta = \frac{\epsilon}{1+\epsilon}$, taking $T = \frac{2}{\beta^2\delta}$ ensures that, if T positive invocations occur, then the protocol's output is within $(1-\epsilon)\frac{v}{N}$ and $(1+\epsilon)\frac{v}{N}$, except for $\delta/2$ probability. To complete the protocol, we set $M = N(\ln\delta + 1)$ so that if $v \neq 0$, the probability of not having T positive invocations is at most $\delta/2$.

Note that the number of rounds in protocol Intersect-Approx is not fixed, and depends on the exact intersection size v. The protocol is optimal in the sense that it matches the lower-bound for distinguishing inputs with intersection size k from inputs with intersection size $k(1+\epsilon)$ in an expected $O(N/k)$ invocations of Private-Sample-B.

Caveat. As the number of rounds in our protocol is a function of its outcome, an observer that only counts the number of rounds in the protocol, or the time it takes to run it, may estimate its outcome. The problem is inherent in our security definitions—both for semi-honest and malicious parties—as they only take into account the parties that "formally" participate in the protocol (unlike, *e.g.*, in universal composability [6]). In particular, these definitions allow for any information that is learned by all the participating parties to be sent in the clear. While it may be that creating secure channels for the protocol (*e.g.*, using encryption) prevents this leakage in many cases, this is not a sufficient measure in general nor specifically for our protocol (as one must hide the communication length of Intersect-Approx).

7 The Multi-Party Case

We briefly discuss computing the intersection in a multi-party environment. Assume that there are n parties, $P_1, \ldots, P_n$, with corresponding lists of inputs $X_1, \ldots, X_n$; w.l.o.g., we assume each list contains k inputs. The parties compute the intersection of *all* n lists. We only sketch a protocol for semi-honest parties, starting with a basic protocol that is secure with respect to client parties $P_1, \ldots, P_{n-1}$ and then modifying it get security with respect to all parties.

A strawman protocol. Let client parties $P_1, \ldots, P_{n-1}$ each generate a polynomial encoding their input, as for Protocol PM-Semi-Honest in the two-party case. Each client uses her own public key and sends the encrypted polynomials to P_n, which we refer to as the *leader*. This naming of parties as *clients* and the *leader* is done for conceptual clarity.

For each item y in his list, leader P_n prepares $(n-1)$ random shares that XOR to y. He then evaluates the $(n-1)$ polynomials he received, encoding the lth share of y as the payload of the evaluation of the lth polynomial. Finally, he publishes a shuffled list of $(n-1)$-tuples. Each tuple contains the encryptions that the leader obtained while evaluating the polynomials on input y, for every y in his input set. Note that every tuple contains exactly one entry encrypted with the key of client P_l, for $1 \leq l \leq n-1$.

To obtain the outcome, each client P_l decrypts the entries that are encrypted with her public key and publishes them. If XOR-ing the decrypted values results in y, then y is in the intersection.

Achieving security with respect to semi-honest parties. This strawman approach is flawed. The leader P_n generates the shares that the clients decrypt. Hence, he may recognize, for values y in his set but not in the intersection, which clients also hold y: these clients, and only these clients, would publish the right shares. We can fix this problem by letting each client generate k sets of random shares that XOR to zero (one set for each of the leader's inputs). Then, each client encrypts one share from each set to every other client. Finally, the clients publish the XOR of the original share from the leader with the new shares from other clients. If y is in the intersection set, then the XOR of all published values for each of the leader's k inputs is still y, otherwise it looks random to any coalition. More concretely, the protocol for semi-honest parties is as follows.

1. A client party P_i, for $1 \leq i \leq n-1$, operates as in the two-party case. She generates a polynomial Q_i of degree k encoding her inputs, and generates homomorphic encryptions of the coefficients (with her own public key). P_i also chooses k sets of $n-1$ random numbers, call these $\{s^i_{j,1}, \ldots, s^i_{j,n-1}\}^k_{j=1}$. We can view this as a matrix with k rows and $(n-1)$ columns: Each column corresponds to the values given to party P_l; each row corresponds to the random numbers generated for one of the leader's inputs. This matrix is chosen such that the XOR of each row sums to zero, *i.e.*, for $j = 1 \ldots k$, $\bigoplus_{l=1}^{n-1} s^i_{j,l} = 0$. For each column l, she encrypts the corresponding shares using the public key of client P_l. She sends all her encrypted data to a public bulletin board (or just to the leader who acts in such a capacity).
2. For each item y in his list X_n (the rows), leader P_n prepares $(n-1)$ random shares $\sigma_{y,l}$ (one for each column), where $\bigoplus_{l=1}^{n-1} \sigma_{y,l} = y$. Then, for each of the k elements of the matrix column representing client P_l, he computes the encryption of $(r_{y,l} \cdot Q_l(y) + \sigma_{y,l})$ using P_l's public key and a fresh random number $r_{y,l}$. In total, the leader generates k tuples of $(n-1)$ items each. He randomly permutes the order of the tuples and publishes the resulting data.
3. Each client P_l decrypts the n entries that are encrypted with her public key: namely, the lth column generated by P_n (of k elements) and the $(n-1)$ lth columns generated by clients (each also of k elements). P_l computes the XOR of each row in the resulting matrix: $(\bigoplus_{i=1}^{n-1} s^i_{j,l}) \oplus \sigma_{j,l}$. She publishes these k results.
4. Each P_i checks if the XOR of the $(n-1)$ published results for each row is equal to a value y in her input: If this is the case, $\bigoplus_{l=1}^{n-1} \left((\bigoplus_{i=1}^{n-1} s^i_{j,l}) \oplus \sigma_{j,l} \right) = y$, and she concludes that y is in the intersection.

Intuitively, the values output by each client (Step 3) appear random to the leader, so he cannot differentiate between the output from clients with y in their input and those without, as he could in the strawman proposal.

Note that the communication involves two rounds in which $P_1, \ldots P_{n-1}$ submit data, and a round where P_n submits data. This is preferable to protocols consisting of many rounds with n^2 communication. The computation overhead of P_n can be improved by using the hashing-to-bins method of Section 4.2.

8 Fuzzy Matching and Fuzzy Search

In many applications, database entries are not always accurate or full (*e.g.*, due to errors, omissions, or inconsistent spellings of names). In these cases, it would be useful to have a private matching algorithm that reports a match even if two entries are only *similar.*

We let each database entry be a list of T attributes, and consider $X = (x_1, \ldots, x_T)$ and $Y = (y_1, \ldots, y_T)$ similar if they agree on (at least) $t < T$ attributes. One variant is fuzzy search, where the client specifies a list of attributes and asks for all the database entries that agree with at least t of the attributes. This may be achieved by a simple modification of our basic `PM-Semi-Honest` protocol, by letting the server reply with the encryptions of $r_i \cdot P_i(y_i) + s_i$, where t shares of $s_1, \ldots, s_T$ are necessary and sufficient for recovering Y. This fuzzy search scheme may be used to compare two "databases" each containing just one element comprised of many attributes.

The protocol may be modified to privately compute fuzzy matching in larger databases, *e.g.*, when a match is announced if entries agree on t out of T attributes. In this section, we present a scheme, in the semi-honest model, that considers a simple form of this fuzzy private matching problem.

A 2-out-of-3 fuzzy matching protocol A client $\mathcal{C}$ has $k_\mathcal{C}$ 3-tuples $X_1, \ldots, X_{k_\mathcal{C}}$. Let P_1, P_2, P_3 be polynomials, such that P_j is used to encode the jth element of the three tuple, X_i^j, for $1 \leq i \leq k_\mathcal{C}$. For all i, let $\mathcal{C}$ choose a new random value R_i and set $R_i = P_1(X_i^1) = P_2(X_i^2) = P_2(X_i^3)$. In general, the degree of each such polynomial is $k_\mathcal{C}$, and therefore, two non-equal polynomials can match in at most $k_\mathcal{C}$ positions. $\mathcal{C}$ sends (P_1, P_2, P_3) to $\mathcal{S}$ as encrypted coefficients, as earlier. The server $\mathcal{S}$, for every three-tuple Y_i in his database of size $k_\mathcal{S}$, responds to $\mathcal{C}$ in a manner similar to Protocol `PM-Semi-Honest`: He computes the encrypted values $r(P_1(Y_i^1) - P_2(Y_i^2)) + Y_i$, $r'(P_1(Y_i^2) - P_3(Y_i^3)) + Y_i$, and $r''(P_1(Y_i^1) - P_3(Y_i^3)) + Y_i$. If two elements in Y_i are the same as those in X_i, the client receives Y_i in one of the entries.

We leave as open problems the design of more efficient fuzzy matching protocols (without incurring a $\binom{T}{t}$ factor in the communication complexity) and of protocols secure in the malicious model.

References

1. Bill Aiello, Yuval Ishai, and Omer Reingold. Priced oblivious transfer: How to sell digital goods. In *Advances in Cryptology—EUROCRYPT 2001*, Innsbruck, Austria, May 2001.
2. Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, 1999.
3. Fabrice Boudot, Berry Schoenmakers, and Jacques Traore. A fair and efficient solution to the socialist millionaires' problem. *Discrete Applied Mathematics*, 111(1-2):23–036, 2001.
4. Andrei Z. Broder and Michael Mitzenmacher. Using multiple hash functions to improve ip lookups. In *IEEE INFOCOM'01*, pages 1454–1463, Anchorage, Alaska, April 2001.
5. Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *Advances in Cryptology—EUROCRYPT '99*, pages 402–414, Prague, Czech Republic, May 1999.

6. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, Nevada, October 2001.
7. Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *4th International Workshop on Practice and Theory in Public Key Cryptosystems (PKC 2001)*, pages 13–15, Cheju Island, Korea, February 2001.
8. Alexandre Evfimievski, Johannes Gehrke, and Ramakrishnan Srikant. Limiting privacy breaches in privacy preserving data mining. In *Proc. 22nd ACM Symposium on Principles of Database Systems (PODS 2003)*, pages 211–222, San Diego, CA, June 2003.
9. Ronald Fagin, Moni Naor, and Peter Winkler. Comparing information without leaking it. *Communications of the ACM*, 39(5):77–85, 1996.
10. Joan Feigenbaum, Yuval Ishai, Tal Malkin, Kobbi Nissim, Martin Strauss, and Rebecca N. Wright. Secure multiparty computation of approximations. In *Automata Languages and Programming: 27th International Colloquim (ICALP 2001)*, pages 927–938, Crete, Greece, July 2001.
11. Oded Goldreich. Secure multi-party computation. In *Available at Theory of Cryptography Library,* `http://philby.ucsb.edu/cryptolib/BOOKS`, 1999.
12. Bernardo A. Huberman, Matt Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *Proc. ACM Conference on Electronic Commerce*, pages 78–86, Denver, Colorado, November 1999.
13. Russell Impagliazzo and Steven Rudich. Limits on the provable consequences of one-way permutations. In *Proc. 21st Annual ACM Symposium on Theory of Computing*, pages 44–61, Seattle, Washington, May 1989.
14. Bala Kalyanasundaram and Georg Schnitger. The probabilistic communication complexity of set intersection. *SIAM J. Discrete Mathematics*, 5(4):545–557, 1992.
15. Aggelos Kiayias and Moti Yung. Secure games with polynomial expressions. In *Automata Languages and Programming: 27th International Colloquim (ICALP 2001)*, pages 939–950, Crete, Greece, July 2001.
16. Eyal Kushilevitz and Noam Nisan. *Communication Complexity.* Cambridge University Press, Cambridge, 1997.
17. Helger Lipmaa. Verifiable homomorphic oblivious transfer and private equality test. In *Advances in Cryptology—ASIACRYPT 2003*, pages 416–433, Taipei, Taiwan, November 2003.
18. Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *Proc. 31st Annual ACM Symposium on Theory of Computing*, pages 245–254, Atlanta, Georgia, May 1999.
19. Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *SIAM Symposium on Discrete Algorithms (SODA)*, pages 448–457, Washington, D.C., January 2001.
20. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology—EUROCRYPT '99*, pages 223–238, Prague, Czech Republic, May 1999.
21. Pascal Paillier. Trapdooring discrete logarithms on elliptic curves over rings. In *Advances in Cryptology—ASIACRYPT 2000*, pages 573–584, Kyoto, Japan, 2000.
22. Alexander A. Razborov. Application of matrix methods to the theory of lower bounds in computational complexity. *Combinatorica*, 10(1):81–93, 1990.

On-the-Fly Verification of Rateless Erasure Codes for Efficient Content Distribution

Maxwell N. Krohn
MIT
krohn@mit.edu

Michael J. Freedman
NYU
mfreed@cs.nyu.edu

David Mazières
NYU
dm@cs.nyu.edu

***Abstract*— The quality of peer-to-peer content distribution can suffer when malicious participants intentionally corrupt content. Some systems using simple block-by-block downloading can verify blocks with traditional cryptographic signatures and hashes, but these techniques do not apply well to more elegant systems that use rateless erasure codes for efficient multicast transfers. This paper presents a practical scheme, based on homomorphic hashing, that enables a downloader to perform on-the-fly verification of erasure-encoded blocks.**

I. Introduction

Peer-to-peer content distribution networks (P2P-CDNs) 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.

Exclusively unicast P2P-CDNs are furthermore extremely wasteful of bandwidth: a small number of files account for a sizable percentage of total transfers. Recent studies indicate that from a university network, KaZaa's 300 top bandwidth-consuming objects can account for 42% of all outbound traffic [1]. Multicast transmission of popular files might drastically reduce the total bandwidth consumed; however, traditional multicast systems would fare poorly in such unstable networks.

Developments in practical erasure codes [2] and *rateless* erasure codes [3], [4], [5] point to elegant solutions for both problems. 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 reenter 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 [6], [7]. More current work considers rateless erasure codes in unicast, multi-source P2P-CDNs [8], [9]. Most recently, SplitStream [10] has explored applying rateless erasure codes to overlapping P2P multicast networks, and Bullet [11] 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 P2P-CDNs cannot make this assumption.

This paper 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 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.

In the remainder of this paper, we will discuss our setting in more detail (Sections II and III), describe our hashing scheme (Section IV), analyze its important properties (Sections V and VI), discuss related works (Section VII), and conclude (Section VIII).

II. Brief Review of Erasure Codes

In this paper, we consider 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

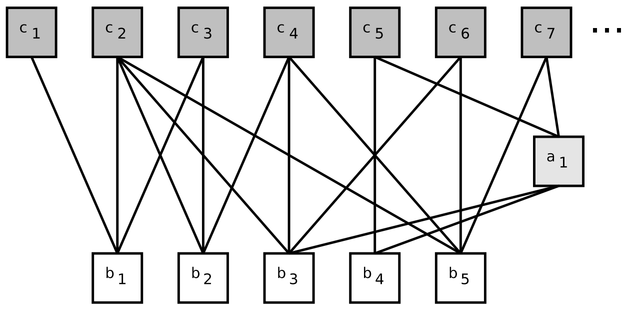


Fig. 1. Example Online encoding of a five-block file. b_i are message blocks, a_1 is an auxiliary block, and c_i are check blocks. Edges represent addition (via XOR). For example, $c_4 = b_2 + b_3 + b_5$, $a_1 = b_3 + b_4$, and $c_7 = a_1 + b_5$.

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 [12], 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 [7], LT Codes [3], Raptor Codes [5] and Online Codes [4]. 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 [9]. 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.

This paper uses Online Codes when considering the specifics of the encoding and decoding processes; however, all three rateless techniques are closely related, and the techniques described are equally applicable to LT and Raptor Codes.

Online Codes. 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 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 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.

III. Threat Model

Deployed P2P-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 SHA1 [13] 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 "down-sampling" 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 P2P-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 P2P-CDN model is vulnerable to a host of different attacks:

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 F.

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 [14].

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. In more interesting P2P-CDNs, trusted publishers might sign file hashes. Consider the case of a Linux vendor using a P2P-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. The general mechanics of reliable filename resolution are beyond the scope this paper; for the most part, 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, P2P-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.

A. *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' [15]. 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 [14]. 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 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

[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).

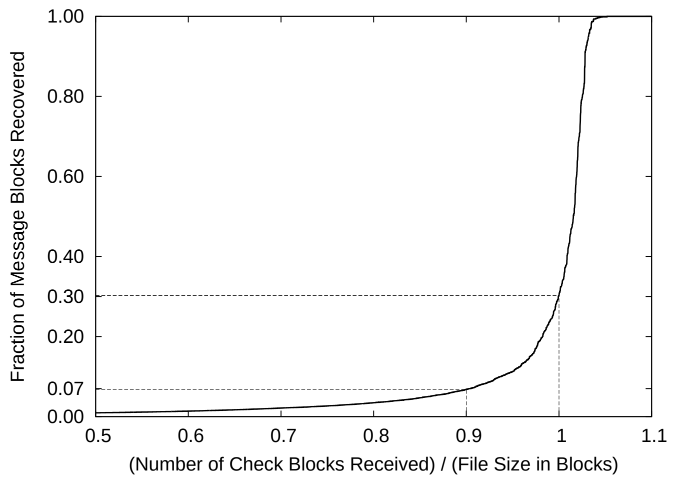


Fig. 2. Number of blocks recoverable as function of number of blocks received. Data collected over 50 random encodings of a 10,000 block file.

only $.068n$ 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.

B. Hashing Check Blocks

Instead of hashing the input to the erasure encoder, publishers might hash its output. If so, the P2P-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 only those check blocks that the publisher has preauthorized. With the benefit of the hash tree taken over all possible check blocks, the receiver can trivially verify check blocks as they arrive. Section V-D 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.

IV. 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 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 file name 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 data sets could travel through such a network.

A. 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 $g^{\mathbf{r}} = (g^{r_1}\ g^{r_2}\ \cdots\ g^{r_m})$.

B. 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 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 Table I.

In decentralized P2P-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 3. 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 SHA1, 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

TABLE I

SYSTEM PARAMETERS AND PROPERTIES

Name	Description	*e.g.*
λ_p	discrete log security parameter	1024
λ_q	discrete log security parameter	257
p	random prime, $\lvert p\rvert = \lambda_p$	
q	random prime, $q \mid (p-1)$, $\lvert q\rvert = \lambda_q$	
β	block size in bits	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	1 GB
k	precoding parameter	3
δ	fraction of unrecoverable message blocks (without the benefit of precoding)	.005
n'	precoded file size, $n' = (1+\delta k)n$	1.015 GB
ϵ	asymptotic encoding overhead	.01
d	average degree of check blocks	~ 8.17

per file F such that $s = \mathsf{SHA1}(N(F))$. Either way, the same parameters G will always be used when hashing file F.

File Representation. As per Table I, 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.

```
Algorithm PickGroup($\lambda_p, \lambda_q, m, s$)
  Seed PRNG $\mathcal{G}$ with $s$.
  do
    $q \leftarrow \mathsf{qGen}(\lambda_q)$
    $p \leftarrow \mathsf{pGen}(q, \lambda_p)$
  while $p = 0$ done
  for $i = 1$ to $m$ do
    do
      $x \leftarrow \mathcal{G}(p-1) + 1$
      $g_i \leftarrow x^{(p-1)/q} \pmod p$
    while $g_i = 1$ done
  done
  return $(p, q, \mathbf{g})$

Algorithm qGen($\lambda_q$)
  do
    $q \leftarrow \mathcal{G}(2^{\lambda_q})$
  while $q$ is not prime done
  return $q$

Algorithm pGen($q, \lambda_p$)
  for $i = 1$ to $4\lambda_p$ do
    $X \leftarrow \mathcal{G}(2^{\lambda_p})$
    $c \leftarrow X \pmod{2q}$
    $p \leftarrow X - c + 1$    // Note $p \equiv 1 \pmod{2q}$
    if $p$ is prime then return $p$
  done
  return 0
```

Fig. 3. 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 [16]. 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 $\{0, \ldots, x-1\}$.

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 [17] (and from Chaum et al. [18]), and it is similar to that used in various incremental hashing schemes (see Section VII). 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:

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

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

$$H_G(F) = (h_G(\mathbf{b}_1)\ h_G(\mathbf{b}_2)\ \cdots\ h_G(\mathbf{b}_n)) \tag{2}$$

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:

$$h_G(\mathbf{c}) = \prod_{i=1}^{n'} h_G(\mathbf{b_i})^{x_i} \quad (3)$$

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 Section II. 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. LT Codes do not involve preprocessing, so the above scheme can be simplified. Raptor Codes involve a two-stage precoding process, and 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.

C. 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:

$$H_G(F) = g^{\mathbf{r}F} \quad (4)$$

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 Section V-B for a more complete running-time analysis. The hasher 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.

D. Computational Efficiency Improvements

We have presented a bare-bones protocol that achieves our security goals but 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 SHA1. 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 [19] 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) compared to the much larger λ_q-bit exponents used in Equation 1.

Batching does open the receiver to small-scale attacks: a receiver accepts a batch worth of check blocks before closing a connection with a malicious sender. 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. Finally, downloaders might tune the batching parameter t based upon their available bandwidth or gradually increase t for each source, so as to bound its overall fraction of bad blocks.

E. 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 SHA1 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:

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

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 $N(F) \rightarrow J_G(F)$, 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 $N(F) \rightarrow J_G(F)$, for some G. He then uses the peer-to-peer routing layer 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, and 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, $\lambda_p/(\beta - \lambda_p) \approx 0.79\%$.

V. Analysis

In this section, we analyze our hashing scheme and report performance numbers of a sample implementation.

A. Correctness

We first claim that homomorphic hashing scheme coupled with the batch verifier given in Section IV-D guarantees correctness. That is, a verifier should always accept the encoded output from an honest sender. Our proof is in Appendix I. Given the proof of this more involved probabilistic verifier, it is easy to see that the naïve verifier is also correct.

B. 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 the $(m\lambda_q/2 + m - 1)\,\mathsf{MultCost}(p)$ 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 4 requires an expected $n\lambda_q/2$ multiplications in $\mathbb{Z}_p^*$. To compute $\mathbf{r}F$ as in Equation 4, mn

```
Algorithm FastMult $((y_1, s_1), \ldots, (y_t, s_t))$
  $y \leftarrow 1$
  for $j = l - 1$ down to $0$ do
    for $i = 1$ to $t$ do
      if $s_i[j] = 1$ then $y \leftarrow y y_i$
    done
    if $l > 0$ then $y \leftarrow y^2$
  done
  return $y$
```

Fig. 4. Algorithm for computing $\prod_{i=1}^{t} y_i^{s_i}$. Each s_i is an l-bit number, and the notation $s_i[j]$ gives the j^{th} bit of s_i, $s_i[0]$ being the least significant bit. This algorithm is presented in [19], although we believe there to be an off-by-one-error in that paper, which we have corrected here.

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 1, with only the benefit of the precomputed squares of the g_i. If we ignore these costs, Global Hash Generation requires a total of $nm\lambda_q\,\mathsf{MultCost}(p)/2$ worth of computation.

Naïve 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 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 Section IV-D. 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 suggest the FastMult algorithm described in Figure 4, 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:

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

C. Microbenchmarks

We implemented a version of these hash primitives using the GNU MP library, version 4.1.2. Table II shows the results of our C++ testing program when run on a 3.0 GHz Pentium 4, with the sample parameters given in Table I and the batching parameters given in Section IV-D. 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 SHA1 and for the Rabin signature scheme with 1024-bit keys [20]. 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. We will call on these benchmarks in the next section.

Although batched verification of hashes is almost an order of magnitude slower than a more conventional hash function such as SHA1, 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 Appendix III for details). Our performance analysis focuses on the per-publisher scheme, which we believe to be better-suited for copyright-friendly distribution of bulk data.

D. Performance Comparison

In Section III-A, we discussed other strategies for on-the-fly verification of check blocks in peer-to-peer networks. 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 SHA1 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 Section IV-E 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 SHA1 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 SHA1 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

[2]FastMult offers no per-block performance improvement for naïve verification, thus we only consider it for fast verification.

TABLE II
MICROBENCHMARKS

Operation on 16 KB block $\mathbf{b}$	time (msec)	throughput (MB/sec)
Per-publisher computation of $h_G(\mathbf{b})$	1.39	11.21
Global computation of $h_G(\mathbf{b})$	420.90	0.037
Naïve 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

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

1) 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/sec (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 than 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 Table III. In the final three columns, we have attempted to provide some concrete realizations of our theoretical bounds. Throughout, 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/sec as we observed on our machine. Under these conditions, an encoder can achieve theoretical encoding throughputs of up to 28.9 MB/sec. We further assume that (4) looking 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 encoding of n/r file blocks. Moreover, in the sign-every-block scheme, 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 Table III.

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

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

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

TABLE III
HASH GENERATION

Scheme	Block Reads	DLog Hashes	SHA1 Hashes	Sigs	Disk (sec)	CPU (sec)	Lower Bound (sec)
Homomorphic Hashing	n	$n\beta/(\beta - \lambda_p)$	1	1	17.69	91.81	91.81
Big-Degree SHA1 Hash Tree	$n^2\beta/(ra)$	0	$(n/r)\beta/(\beta - 160)$	1	566.23	293.96	566.23
Binary SHA1 Hash Tree	$n^2\beta/(ra)$	0	$2n/r$	1	566.23	587.20	587.20
Sign Every Block	$n^2\beta/(ra)$	0	0	n/r	566.23	2076.18	2076.18

TABLE IV
ADDITIONAL STORAGE REQUIREMENTS FOR MIRRORS

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

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 this paper.

2) 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/sec and 51.3 MB/sec, respectively. However, recall that disk throughput and memory limitations combine to bound encoding for both schemes at only 28.9 MB/sec. Moreover, these throughputs are quite large relative to typical network throughput; many P2P-CDN mirror nodes would be happy with T1-rates at 1.5 Mbit/sec.

3) Storage Required on the Mirror: Mirrors participating in P2P-CDNs agree to 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 SHA1 hash tree schemes should retrieve complete hash trees from the publisher and store them to disk, or otherwise must 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. He has no choice but to store all signatures. We summarize these additional storage requirements in Table IV, again assuming a 1 GB input file and an encoding rate of $r = 1/16$.

4) Bandwidth: The bandwidth requirements of the various schemes are given in terms of up-front and per-block costs. These results are considered in Table V. 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 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 problem affects all four schemes equally and can be mitigated by general set-reconciliation algorithms [8] and protocols specific to peer-to-peer settings [9].

The binary SHA1 tree and the sign-every-block scheme allow downloaders to retrieve a file without 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 requests all those signatures not already received. This translates to roughly 11.7% additional overhead when $r = 1/16$.

5) Verification: Table VI summarizes the per-block verification costs of the four schemes. For our homomorphic hashing scheme, we assume batched verification with the parameters given in Section IV-D. The Rabin signature scheme was specially chosen due to its fast verification time, as shown. Surprisingly, verifying a check block using a SHA1 binary tree is more than twice as slow as using our homomorphic hashing protocol, due to the height of the tree.

E. 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 tradeoffs, as it performs well across all categories considered. Homomorphic hashing ranks less favorably when considering verification throughput, but as argued in Section V-C, tuning batch size allows throughput to scale with available bandwidth.

TABLE V

BANDWIDTH

	Up-Front		Per-Block			
Scheme	Predicted (bits)	*e.g.* (MB)	Predicted (bits)	*e.g.* (KB)	Total (GB)	Total w/ Penalty (GB)
Homomorphic Hashing	$\lambda_p n\beta/(\beta-\lambda_p)$	8.06	$\beta+m$	16.06	1.0118	1.0118
Big-Degree SHA1 Hash Tree	$160n\beta/(\beta-160)$	20.02	β	16.00	1.0196	1.0528
Binary SHA1 Hash Tree	0	0	$\beta+160\log_2(n/r)$	16.39	1.0244	1.0578
Sign Every Block	0	0	$\beta+\lambda_\sigma$	16.13	1.0078	1.0407

TABLE VI

PER-BLOCK VERIFICATION PERFORMANCE

Scheme	Batch	SHA1	Rabin	Total (msec)
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

VI. SECURITY

In modern real-world P2P-CDNs, an honest receiver who wishes to obtain the file F from the network communicates almost exclusively with untrusted parties. As mentioned in Section III, a crucial stage of the file transmission protocol—mapping file names to file hashes—is beyond the scope of this paper. In our analysis, we assume that the receiver can reliably resolve $N(F)\rightarrow J_G(F)$ 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.

A. Collision-Resistant Hash Functions

First, we formally define a collision-resistant hash function (CRHF) in the manner of [21]. 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

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

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

Our definition of the hash primitive h per Section IV fits naturally into this definition. In fact, the PickGroup algorithm is a reasonable candidate for the function HGen(). See [21] 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).

B. 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 Section II, 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 the 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 $m, n > 0$, any batch size $t > 1$, let:

$$\begin{aligned} &\mathsf{Adv}^{\text{enc-atk}}_{\mathcal{H},\mathcal{V},m,n,t}(\mathcal{A}) = \\ &\quad \Pr\big[G \leftarrow \mathsf{HGen}(\mathcal{H});\ (F,X,C) \leftarrow \mathcal{A}(G,m,n,t); \\ &\qquad F' \leftarrow \mathcal{P}(F);\ b \leftarrow \mathcal{V}(\mathcal{H}_G(F'),G,X,C) : \\ &\qquad F \text{ is } m\times n \ \wedge\ F' \text{ is } m\times n' \ \wedge\ X \text{ is } n'\times t \\ &\qquad\quad \wedge\ C \text{ is } m\times t \ \wedge\ F'X \neq C \ \wedge\ b = \mathsf{Accept}\big] \end{aligned} \tag{5}$$

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. See Appendix II.

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 naïve 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 Section IV-E follows from an inductive application of Theorem 1.

C. *Future Work and End-To-End Security*

These security guarantees, while necessary, are not sufficient for all multicast settings. In Section III, we proposed both the bogus-encoding attack and the distribution attack. While we have solved the former, one can imagine malicious encoders who 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 Section V-D 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 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 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; therefore we leave the description and analysis of an exact algorithm to future work.

VII. Related Works

Multicast source-authentication is well-studied problem in the recent literature; for a taxonomy of security concerns and some schemes, see [22]. 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 [23] 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 [24] delay consecutive packets into a pool, then form an authentication hash and sign the tree's root. Rohatgi [25] uses reduced-size online/offline k-time signatures instead of hashes. Recent tree-based [26] and graph-based [27] approaches reduce the time/space overheads and are designed for bursty communication and random packet loss. More recent work [28], [29] makes use of *trusted* erasure encoding in order to authenticate blocks, while most schemes, including our own, try 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 used the delayed disclosure of symmetric keys to provide source authentication, including Chueng [30], the Guy Fawkes protocol [31], and more recently TESLA [32], [33], by relying on loose time synchronization between senders and recipients. The recent BiBa [34] 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 [6], [7] 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 P2P-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 [35], [36] and incremental hashing [21], based on RSA and DL constructions respectively, examine commutative hash functions that yield an output independent of the operations' order. Improvements to the schemes' efficiency [37], [38], [39], 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 [40], authenticated prefix aggregation [41], redactable signatures [42], and set-union signatures via accumulators [42]. We use similar techniques to maintain group structure across applications of cryptographic functions, but to different ends. Composing homomorphic signatures with traditional hash functions such as SHA1 [13] would not solve our problem, as the application of the traditional hash function would destroy the group structure we hope to preserve.

VIII. Conclusions

Current peer-to-peer content distribution networks, such as the widely popular file-sharing systems, suffer from unverified downloads. A participant may download an entire file, increasingly in the hundreds of megabytes, before determining that the file is corrupted or mislabeled. Current downloading techniques can use simple cryptographic primitives such as signatures and hash trees to authenticate data. However, these approaches are not efficient for low encoding rates, and are not possible for rateless codes.

To our knowledge, this paper is the first to consider non-interactive, on-the-fly verification of rateless erasure codes. We present a discrete-log-based hash scheme that provides useful homomorphic properties for verifying the integrity of downloaded content. Because recipients can compose hashes just as encoders compose message blocks, they can verify any possible check block. Using batching techniques to improve verification efficiency, we provide implementation results that suggest this scheme is practical for real-world use. A tight reduction proves our scheme secure under standard cryptographic assumptions. We leave formalization of end-to-end security and protection against distribution attacks as interesting open problems.

Acknowledgments

We thank Michael Walfish for first alerting us to the distribution attack. We also thank Petar Maymounkov and Benny Pinkas for helpful discussions, and our shepherd Dan Wallach for his feedback. This research was conducted as part of the IRIS project (`http://project-iris.net/`), supported by the NSF under Cooperative Agreement No. ANI-0225660. Maxwell Krohn is supported by an MIT EECS Fellowship, Michael Freedman by an NDSEG Fellowship, and David Mazières by an Alfred P. Sloan Research Fellowship.

References

[1] 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.

[2] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann, "Practical loss-resilient codes," in *Proc. 29th Annual ACM Symposium on Theory of Computing (STOC)*, El Paso, TX, May 1997.

[3] M. Luby, "LT codes," in *Proc. 43rd Annual Symposium on Foundations of Computer Science (FOCS)*, Vancouver, Canada, Nov. 2002.

[4] P. Maymounkov, "Online codes," NYU, Tech. Rep. 2002-833, Nov. 2002.

[5] A. Shokrollahi, "Raptor codes," Digital Fountain, Inc., Tech. Rep. DF2003-06-001, June 2003.

[6] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A Digital Fountain approach to reliable distribution of bulk data," in *Proc. ACM SIGCOMM '98*, Vancouver, Canada, Sept. 1998.

[7] J. Byers, M. Luby, and M. Mitzenmacher, "Accessing multiple mirror sites in parallel: Using Tornado codes to speed up downloads," in *Proc. IEEE INFOCOM '99*, New York, NY, Mar. 1999.

[8] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed content delivery across adaptive overlay networks," in *Proc. ACM SIGCOMM '02*, Aug. 2002.

[9] P. Maymounkov and D. Mazières, "Rateless codes and big downloads," in *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, Feb. 2003.

[10] 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.

[11] 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.

[12] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *ACM Computer Communication Review*, vol. 27, no. 2, Apr. 1997.

[13] FIPS 180-1, *Secure Hash Standard*, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfi eld, VA, Apr. 1995.

[14] 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.

[15] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology—CRYPTO '87*, Santa Barbara, CA, Aug. 1987.

[16] National Institute of Standards and Technology, "Digital Signature Standard (DSS)," Federal Information Processing Standards Publication 186-2, U.S. Dept. of Commerce/NIST, 2000.

[17] T. P. Pedersen, "Non-interactive and information-theoretic secure verifi able secret sharing," in *Advances in Cryptology—CRYPTO '91*, Santa Barbara, CA, Aug. 1991.

[18] D. Chaum, E. van Heijst, and B. Pfi tzmann, "Cryptographically strong undeniable signatures, unconditionally secure for the signer," in *Advances in Cryptology—CRYPTO '91*, Santa Barbara, CA, Aug. 1991.

[19] M. Bellare, J. Garay, and T. Rabin, "Fast batch verifi cation for modular exponentiation and digital signatures," in *Advances in Cryptology—EUROCRYPT 98*, Helsinki, Finland, May 1998.

[20] M. O. Rabin, "Digitalized signatures and public key functions as intractable as factorization," MIT Laboratory for Computer Science, Tech. Rep. TR-212, Jan. 1979.

[21] 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.

[22] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas, "Multicast security: A taxonomy and some effi cient constructions," in *Proc. IEEE INFOCOM '99*, New York, NY, 1999.

[23] R. Gennaro and P. Rohatgi, "How to sign digital streams," in *Advances in Cryptology—CRYPTO '97*, Santa Barbara, CA, Aug. 1997.

[24] C. K. Wong and S. S. Lam, "Digital signatures for fbws and multicasts," in *Proc. IEEE International Conference on Network Protocols*, Austin, TX, Oct. 1998.

[25] 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.

[26] 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.

[27] S. Miner and J. Staddon, "Graph-based authentication of digital streams," in *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

[28] A. Pannetrat and R. Molva, "Efficient multicast packet authentication," in *Proc. Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2003.
[29] J. M. Park, E. K. P. Chong, and H. J. Siegel, "Efficient multicast stream authentication using erasure codes," *ACM Trans. Inf. Syst. Secur.*, vol. 6, no. 2, 2003.
[30] S. Cheung, "An efficient message authentication scheme for link state routing," in *Proc. 13th Annual Computer Security Applications Conference*, San Diego, CA, Dec. 1997.
[31] R. Anderson, F. Bergadano, B. Crispo, J.-H. Lee, C. Manifavas, and R. Needham, "A new family of authentication protocols," *Operating Systems Review*, vol. 32, no. 4, Oct. 1998.
[32] 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.
[33] ——, "Efficient and secure source authentication for multicast," in *Proc. Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2001.
[34] 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.
[35] 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.
[36] 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.
[37] 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.
[38] J. Camenisch and A. Lysyanskaya, "Dynamic accumulators and application to efficient revocation of anonymous credentials," in *Advances in Cryptology—CRYPTO 2002*, Santa Barbara, CA, Aug. 2002.
[39] G. Tsudik and S. Xu, "Accumulating composites and improved group signing," in *Advances in Cryptology—ASIACRYPT-2003*, Taipei, Taiwan, Nov. 2003.
[40] S. Micali and R. Rivest, "Transitive signature schemes," in *Progress in Cryptology — CT-RSA 2002*, San Jose, CA, Feb. 2002.
[41] S. Chari, T. Rabin, and R. Rivest, "An efficient signature scheme for route aggregation," Feb. 2002.
[42] R. Johnson, D. Molnar, D. Song, and D. Wagner, "Homomorphic signature schemes," in *Progress in Cryptology — CT-RSA 2002*, San Jose, CA, Feb. 2002.

Appendix I
Correctness of Batched Verification

Consider the batched verification algorithm given in Section IV-D. To prove it 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 1, rewrite hashes of check blocks in terms of a common generator g:

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

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

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

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

Appendix II
Proof of Theorem 1

We now prove the security of the batched verification scheme by proving Theorem 1 given in Section VI-B. Our proof follows that from [19], 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, 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 5. Step 2 enforces appropriate dimensionality. Step 4 enforces the requirements that $\langle X, C\rangle$ not be a legal encoding, given in Equation 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 Section IV-D must have output Accept. Using the same manipulations as those given in Appendix I, we take the fact that $\mathcal{V}$ accepted to mean that:

$$g^{\mathbf{r}F'X\mathbf{s}} \equiv g^{\mathbf{r}C\mathbf{s}} \bmod p \tag{6}$$

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 $\mathbf{r}F'X\mathbf{s} \equiv \mathbf{r}C\mathbf{s} \bmod q$, and rearranging,

$$\mathbf{r}\,(F'X - C)\,\mathbf{s} \equiv 0 \bmod q. \tag{7}$$

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:

$$\mathbf{r}\left(F'X - C\right) \not\equiv \mathbf{0} \bmod q \tag{8}$$

For convenience, let the $1 \times t$ row vector $\mathbf{u} = \mathbf{r}(F'X - C)$. Equation 8 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 7 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:

$$s_1 \equiv -\left(u_1^{-1}\right) \sum_{j=2}^{t} u_j s_j \bmod q \tag{9}$$

Referring to Step 1 of verifier $\mathcal{V}$, s_1 was selected at random from 2^l possible values; consequently, the probability of its having the particular value in Equation 9 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. ∎

Appendix III
k-ary exponentiation

In order to speed up global hash generation, one can make an exponential space-for-time tradeoff, using k-ary exponentiation. That is, we can speed up each exponentiation by a factor of $x/2$ while costing a factor of $(2^x - 1)/x$ in core memory. For simplicity, assume that $x | (\lambda_q - 1)$:

1) For $1 \leq i \leq m$, for $0 < j < 2^x$, for $0 \leq k < (\lambda_q - 1)/x$, precompute $g_i^{j2^{kx}}$. Store each value in an array A under the index $A[i][j][k]$.
2) To compute $g_i^{z_i}$, write z_i in base 2^x:

$$z_i = a_0 + a_1 2^x + a_2 2^{2x} + \cdots + a_{(\lambda_q - 1)/x - 1} 2^{\lambda_q - x}$$

Let $K = \{k \mid a_k \neq 0\}$. Then compute the product:

$$g_i^{z_i} = \prod_{k \in K} A[i][a_k][k]$$

The storage requirement for the table A is $m(2^x - 1)(\lambda_q - 1)\lambda_p/x$ bits, which is exponential in x. Disregarding the one-time precomputation in Step 1, the computation of z_i in Step 2 costs $(\lambda_q - 1)\,\mathsf{MultCost}(p)/x$. Compared to the conventional iterative-squaring technique, this method achieves a factor of $x/2$ speed-up.

Setting $x = 8$, the size of the tables $|A| = 510$ MB, and we can hash a 1 GB file with global parameters in less than 2 hours (of course hashing is much faster in the per-publisher model).