Deploying Large File Transfer on an HTTP Content Distribution Network

KyoungSoo Park and Vivek S. Pai
Department of Computer Science
Princeton University

Abstract

While HTTP-based Content Distribution Networks have been successfully used to serve Web users, several design and operational issues have prevented them from being used for the scalable and efficient transfer of large files. We show that with a small number of changes, supporting large file transfer can be efficiently handled on existing HTTP CDNs, without modifying client or server software. This approach not only reduces infrastructure and the need for resource provisioning between services, but can also improve reliability by leveraging the stability of the existing CDN to deploy this new service. We show that this approach can be implemented at low cost, and provides efficient transfers even under heavy load.

1 Introduction

HTTP-based Content Distribution Networks (CDNs) have successfully been used to serve Web pages, offloading origin servers and reducing download latency. These systems, such as those from Akamai [2], generally consist of a distributed set of caching Web proxies and a set of request redirectors. The proxies are responsible for caching requested objects so that they can be served from the edge of the network instead of contacting the origin server. The redirectors, often modified DNS servers, send clients to appropriate CDN nodes, using a number of factors, such as server load, request locality (presence of object in the cache), and proximity (network latency to the node).

Given the scale, coverage, and reliability of CDN systems, it is tempting to use them as the base to launch new services that build on the existing software infrastructure. Doing so can reduce the duplication of infrastructure, and reduce or eliminate decisions about resource provisioning on the nodes. For example, some current approaches to distributing different types of content (Web objects, streaming media) often require running multiple separate servers, so decisions about how much memory and disk to devote to each one must be made in advance, instead of being automatically adjusted dynamically based on the workload. We consider the issue of distribution of large files (100's of MB and above), which has recently been the focus of much academic and development interest [3, 5, 6, 7, 10].

With Web content being heavily skewed to small files and whole-file access, the design decisions that optimize CDN behavior for the average case also make serving large files unattractive. However, these large files can include movie trailers and software CD (ISO) images, making them ideal candidates for CDNs. With main memory in the 1GB range and average Web objects in the 10KB range, a standard proxy used as a CDN node can easily cache 10K-100K objects completely in main memory. While disk caches can be much larger, the slower disk access times drive designers to aim for high main-memory cache hit rates. In this environment, serving files in the 100+MB range can cause thousands of regular files to be evicted from main memory, reducing the overall effectiveness of the CDN node.

This work focuses on deploying a practical large file transfer service that can efficiently leverage HTTP CDN infrastructure, without requiring any changes to the client or server infrastructure. Such a service would allow CDN developers an attractive option for handling this content, and may allow content providers new flexibility in serving these requests.

The rest of this paper is organized as follows: in Section 2, we provide some background and explain the impediments in current CDNs. We then discuss our approach and its implementation, CoDeploy, in Section 3. Finally, we evaluate our performance in Section 4, discuss related work in Section 5, and conclude in Section 6.

2 Background

Large files tax the CDN node's most valuable resource, physical memory. While over-committing other resources (disk, network, CPU) result in linear slowdowns, accessing disk-based virtual memory is much worse than physical memory. Among the problems large files present for content distribution networks are:

Cache hit rates – with current approaches, large files would require much higher hit rates to compensate the CDN/proxy for all the small files they displace in main memory. As a result, proxies generally “tunnel” such files, fetching them from the origin site and not caching them. Some CDNs provide a storage service [9] for unburdening the content providers and enhancing the robustness of delivery. While this approach does offload bandwidth from the origin server, it does not scale the storage of the CDN due to the use of whole-file caching.

Buffer consumption – while multiple client requests for the same uncached file can often be merged and served with one download from the origin server, doing so for large files requires the proxy to buffer all data not received by the slowest client, potentially approaching the entire file size. This buffer either pollutes the cache and consumes memory, or causes disk access and eviction. Throttling the fastest client to reduce the buffer size only makes the CDN slow and increases the number of simultaneous connections.

Serving large files using an HTTP-based infrastructure has received very little attention, and the only published related effort of which we are aware is FastReplica [6], which fragments large files and uses all-to-all communication to push them within a CDN. Its focus is on pushing the whole file to all CDN nodes, and does not affect the whole-file caching and service model used by the nodes. Our focus is breaking away from the whole-file caching model to something more amenable to large file handling.

Other work in large file handling uses custom overlay protocols for peer-to-peer transfers. The most heavily deployed, BitTorrent [7], implements a peer-to-peer network with centralized tracking of available clients. Academic initiatives include (a) LoCI [3], which builds a stateful, filesystem-like network infrastructure, (b) Bullet [10], which uses a self-organizing overlay mesh, and (c) SplitStream [5], which has a superimposed collection of multicast trees on a peer-to-peer overlay. To the best of our knowledge, all of these approaches use (or will use, when deployed) custom protocols for data transfers.

While these designs report promising levels of success within their intended scenarios, they are not readily applicable to our problem, since they require content preparation at the content provider, and custom software at the receivers. Any non-HTTP protocols imply a higher barrier to deployment and decisions regarding resource provisioning. We are interested in efficient handling of large file transfers within HTTP in a way that minimizes fetches from the content providers, reduces the amount of space consumption within the CDN, requires no modification to the clients, and is completely demand-driven rather than requiring advanced preparation.

3 Approach and Implementation

Our approach to handling large files is to treat them as a large number of small files, which can then be spread across the aggregate memory of the CDN as needed. To ensure that this approach is as unobtrusive as possible to clients, servers, and even the CDN’s own proxies, the dynamic fragmentation and reassembly of these small files will have to be performed inside the CDN, on demand.

To hide this behavior from clients, each CDN node has an agent that is responsible for accepting requests for large files and converting them into a series of requests for pieces of the file. To specify pieces, we use HTTP/1.1’s widely-supported byte-range feature [8], which is normally used by browsers to resume interrupted downloads. After these individual requests are injected into the CDN, the results are reassembled by the agent and passed to the actual client. For simplicity, this agent can occupy a different port number than the CDN’s proxy.

Making the stream of requests CDN/proxy-friendly requires modifying the ingress and egress points of the CDN. Since CDNs use URLs for their redirection decisions, simply using the same URL for all byte-range requests would send all requests along the same path, defeating our purpose. Likewise, we are not aware of any HTTP proxy that is capable of caching disjoint pieces of a file. To avoid these problems, we have the CDN agent affix the range information to the URL itself, so that the CDN believes all of the requests are for different files. On egress from the CDN, these requests are normalized – the URL is reverted to the original, and the range information is added as standard headers. The response from the origin server is converted from a 206 code (partial file) to a 200 (whole file), and the headers indicating what range is served are converted into the standard content-length headers.

Since the transfer appears to the CDN as a large number of small files, it can use its normal routing, replication, and caching policies. These cached pieces can then be used to serve future requests. If a node experiences cache pressure, it can evict as many pieces as needed, instead of evicting one large file. Similarly, the addition/departure of nodes will only cause pieces to be fetched from the origin, instead of the whole file. The only difference to the server is that instead of seeing one large file request from one proxy, it sees many byte-range requests from many proxies. Except for the connection and header traffic, no extra bytes are shipped by the server. One artifact of this approach will be an increase in the number of requests to the server, so more entries will appear in the access log.

We have added this support into the CoDeeN content distribution network [18], which runs on 120 PlanetLab nodes in North America. CoDeeN itself is implemented

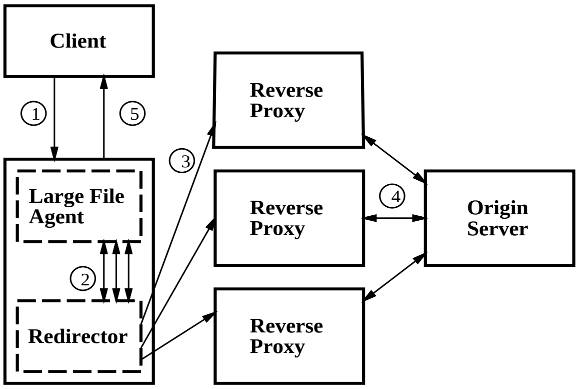

Figure 1: Processing steps for large-file handling - 1. the client sends the request to the agent, 2. the agent generates a series of requests for the URL with special suffixes and sends them to the redirector on the same CDN node, 3. those requests are spread across the CDN as normally occurs, 4. on egress from the CDN (assuming cache misses), the URLs are de-mangled, and the response header is mangled to mimic a regular response, 5. as the data returns to the agent, it sends it back to the client in order, appearing as a single large response

as a module on top of a programmable proxy, using a C API that allows customization of the proxy's data handling [11]. Given a URL, CoDeeN chooses a replica proxy using the Highest Random Weight(HRW) hashing scheme [15], which preserves the uniform distribution of the contents. To avoid hot spots for a popular piece of a file, CoDeeN replicates each piece to k different locations with a configurable number k .

The overall development effort of this system, called CoDeploy, has been relatively light, mostly because the agent can leave all routing, replication, and caching decisions to the CDN. The support in the proxy module for large files is approximately 50 lines of code, mostly dealing with header parsing, insertion and deletion. The bulk of the effort is in the client agent, which we have implemented as a separate program that resides on every CDN node. It consists of less than 500 source lines, excluding comments. However, in that code, it initiates parallel fetches of file pieces, uses non-blocking code to download them in parallel, restarts any slow/failed downloads, and performs other sanity checks to ensure that the large file is being received properly.

The most complicated part of the agent is the restarting of slow downloads. It has been our experience that while most nodes perform "well enough" most of the time, their instantaneous performance can differ dramatically, and some mechanism is needed to prevent a few slow transfers from becoming a bottleneck. To automatically adapt to the speed of the network or the origin server, we keep an exponentially-weighted moving average (EWMA) of the most recent chunk download times per connection, and time out a download when it takes more than a few times the average. When this occurs, the URL is slightly modified by appending a letter to the start position. This letter is meaningless, since the start position is interpreted as an integer, but it causes the CDN to direct the request to a different node than the original. We choose an original timeout of five times the average, and double this value for consecutively failed downloads. The multiple is reset to five once a download succeeds, and the new average includes the recent chunk, increasing the effective future timeout value.

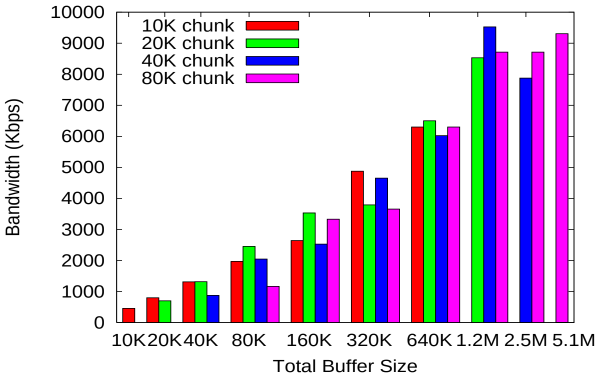

Figure 2: Bandwidth versus total buffer size – buffer size is the chunk size times the # of chunks fetched in parallel

4 Evaluation

We briefly describe some of our early results with CoDeploy, focusing on the performance of our large file support in various scenarios. We also make a number of observations regarding the implementation of our system, and possible future paths for exploration.

To understand the impact of parallelism and chunk size, we vary these parameters on a lightly-loaded node. The client repeatedly requests the file, and the nodes are configured to either cache at the reverse proxies or not cache at all. These results are shown in Figure 2, with the configurations sorted by the product of # of chunks and chunk size. The general trend indicates that performance improves as the total buffer size increases. We do see that larger chunk sizes show some performance loss compared to the smaller ones when the total buffer is small, but this effect is secondary to the correlation with total buffer size. As a general compromise between performance and memory consumption, we choose values of 10 parallel fetches of 60KB each by default.

Our competitive evaluation focuses on the performance of single downloads and multiple simultaneous downloads using different configurations. All tests are

Node	Ping	Direct	Aggr	CoDeploy	
				First	Cached
Rutgers	5.6	18617	14123	3624	4501
U Maryland	5.5	12046	14123	4095	5535
U Notre Dame	35.1	8359	25599	4179	7728
U Michigan	46.9	5688	10239	4708	6205
U Nebraska	51.4	2786	4266	4551	5119
U Utah	66.8	2904	7585	2512	3900
UBC	74.6	3276	7728	2100	4137
U Washington	79.2	4551	17808	1765	4357
UC Berkeley	81.4	4501	20479	6501	9308
UCLA	84.6	2677	7314	2178	4055

Table 1: Ping times (in ms) and bandwidths (in Kbps) across several nodes for various downloading approaches using only one active client at a time.

performed on the 120 nodes running CoDeeN, which are located mostly at North American educational sites in PlanetLab. For the single download tests, we test consecutively on 10 nodes, whereas for the simultaneous download test, all 120 clients begin at the same time. The origin server is a lightly-loaded PlanetLab node at Princeton running a tuned version of Apache, serving a 50MB file, and is connected to the Internet via two high-bandwidth (45 Mbps) connections and one burstable lower-bandwidth connection. The test scenarios are as follows:

1. Direct – fetches from origin in a single download. This is basically what standard browsers would do. However, we increase the socket buffer sizes(=120KB) from the system defaults to expand the space available to cover the bandwidth-delay product.

2. Aggressive – uses byte-range support and multiple parallel connections to download from the origin, similar to the behavior of "download optimizer" programs. The number of parallel connections and chunk size are the same as CoDeploy.

3. CoDeploy First – The client uses CoDeploy, but the content has not been accessed. It must be loaded from the origin server, and is then cached on the reverse proxies.

4. CoDeploy Cached – Clients use CoDeploy, and the cached content is already on the reverse proxies.

We begin our analysis of these results with the single-client tests, shown in Table 1. The direct downloads show general degradation in bandwidth as the ping times increase, which seems surprising since a 50MB file should be large enough to cause the TCP congestion window to grow large enough to cover the bandwidth-delay product. We see that using the aggressive strategy causes a large increase in some of the bandwidths, suggesting that multiple connections can overcome packet loss and congestion much better than a single connection. When the ping times exceed 40ms, CoDeploy's cached results beat the direct downloads, but we see that CoDeploy's single-client numbers do not beat the aggressive strategy.

	25%	Median	Mean	75%
Direct	651	742	851	1037
Aggressive	588	647	970	745
CoDeploy First	2731	3011	2861	3225
CoDeploy Cached	3938	4995	4948	6023

Table 2: Bandwidths (in Kbps) for various downloading approaches with all 120 clients simultaneously downloading.

The situation is much different when comparing multiple simultaneous downloads, as shown in Table 2. The table shows mean, median, 25^{th} and 75^{th} percentile bandwidths for the 120 nodes. Here, the aggressive strategy performs slightly worse than the direct downloads, in general. With all 120 clients simultaneously downloading, the origin server's bandwidth is the bottleneck, and we are using virtually all of Princeton's outbound connectivity. However, both First and Cached show large improvements, with Cached delivering 4-6 times the bandwidth of the non-CoDeploy cases. The performance gain of First over Direct and Aggressive stems from the fact that with multiple simultaneous downloads, some clients are being satisfied by the reverse proxies after the initial fetches are caused by faster clients.

For CoDeploy, these measurements are encouraging, because our long-term goal is to use CoDeeN to absorb flash crowds for large files, such as software releases. In this scenario, CoDeploy is the clear winner. These experiments suggest that if the client and server have a lightly-loaded, high-bandwidth connection between them, either direct connections or an aggressive downloader are beneficial, while CoDeploy can provide better performance if any of those conditions do not hold. The aggressive downloader is not a scalable solution – if everyone were to use it, the extra load could cause severe problems for the origin server. In fact, when our server was behind a firewall, the connection rate caused the firewall to shut down. CoDeploy, on the other hand, performs reasonably well for single clients and extremely well under heavy load.

4.1 Observations and Challenges

Our most important observation is that our approach to large file handling is practical and can be relatively easily implemented on HTTP-based content distribution networks. The system is relatively straightforward to implement and performs well.

The result that the achieved bandwidth is largely a function of the total buffer size is unexpected, but understandable. The underlying reason for this result is that the buffer acts as a sliding window over the file, with various CDN nodes providing the data. The bottleneck will

	base	1%	2%	5%
Univ Utah	3900	5395	7839	10861
UCLA	4055	4339	4814	4955
Univ Brit Columbia	4137	5598	6711	8359
Univ Washington	4357	5926	7488	10542
Rutgers	4501	4811	4939	5072
Univ Nebraska	5119	6339	7079	8895
Univ Maryland	5535	5648	5767	6056
Univ Michigan	6205	6284	6331	7148
Univ Notre Dame	7728	8522	8932	9558
UC Berkeley	9308	10173	11120	14135

Table 3: Improvements from tail reduction – the base number is the current CoDeploy bandwidth, and the other columns show what the bandwidth if that percentage of the slowest chunks were replaced with the median download time.

be the slowest nodes, and as the buffer size increases, the slow nodes enter the sliding window further ahead of the time their data is needed. We should be able to tailor the window size to saturate the client's bandwidth with the least amount of buffer space needed.

This result implies that we should be able to use some form of deadline scheduling to make requests to CDN nodes in a "just in time" fashion, reducing the amount of memory actually consumed by the sliding window. This approach would be conditioned on the stability of download times per CDN node, and the ability of the agent to know where the CDN will send a particular request.

The challenge to such optimization, however, is the variability of download times that we have experienced. We have experienced some nodes that always respond slower than the rest of the CDN. Requests that might have been sent to these nodes could simply be diverted elsewhere, or started much earlier than others. However, the main problem is that even our "fast" nodes tend to show dramatic slowdowns periodically. In some cases, we have noticed some chunks taking more than 10-20 times as long as others, and this behavior is not deterministic. We believe that these are due to short-term congestion, and the gains from reducing them can be significant.

We can estimate the achievable bandwidth improvement by shrinking the download times of the slowest chunks. For each download, we identify the chunks that had the largest download times, and we replace them with the median download time. Any chunks that had been waiting on this bottleneck are time-shifted to reflect the reduction of the bottleneck, and the overall bandwidth of the new trace is calculated. These results, shown in Table 3, show that even modest improvements in speeding up the slowest chunks can dramatically improve bandwidth.

4.2 Supporting Efficient Push and Fast Synchronization

With the large file support in place, one useful service that is easily built on it is efficiently pushing files to n different locations. This includes the initial deployment of the files as well as updating them as new versions are available. This approach essentially provides a scalable 1-to-many version of $rsync$ [16] while avoiding traffic bottlenecks at the source. The necessary steps for enabling this is following,

1. A user sets up a URL containing the directory to be pushed to the destinations. Destinations may have some, all, or none of the files in this directory.

2. A special program, called $cosync$, produces a script which examines and optionally fetches each file in the user's directory hierarchy via the large file agent using the URL specified in step 1. While most files can be transferred without problems, $cosync$ must take some special steps for some files, as described below:

    (a) Any executable bits in the file's privilege mask are cleared, to avoid having the source webserver try to execute the file as a CGI program.

    (b) Since the HTTP standard specifies that proxies should calculate the cacheable lifetime of a file based on its last modification time, recently-modified files are $touch$ed to have an older age. This step ensures that cached files will not expire before the download is complete.

    (c) The execution bits and the modification time are restored to their previous values after downloading is finished.

3. Cosync copies only the script to the destinations and executes it on each node. The script makes sure to download only the modified files for each individual node. For larger scripts, we can store it with the other files, and push only a small script that first downloads it using the large file support before executing it.

Two issues arise in this scenario. One is how to figure out which files to update, and the other is how to prevent the stale version of a cached object from being downloaded. We solve both problems by using the MD5 [13] checksum of each file. Cosync generates and stores the MD5 checksum of each file in the script – when executing, the script checks for an existing file, checks its MD5 checksum, and downloads it only if needed. Also, whenever contacting the large file agent for requesting a file, the script provides the MD5 checksum of the file in an HTTP header. The agent appends this string to the end of each chunk request URL to avoid using any stale copy in the proxy cache. This approach avoids forcing the proxy to unnecessarily revalidate cached content with the origin server, and provides strong consistency. For initial deployments of content, the per-file checking and downloading is not needed, so we also have the option in cosync that creates a tar file of the tree and downloads it as one large file. We have been running cosync on PlanetLab as an open service for over six months.

5 Related Work

In the streaming media community, the technique of caching small segments rather than a whole file is not new [1, 4, 12]. They further optimize the streaming behavior by prefetching and caching more frequently requested portions of the content. Akamai uses a similar strategy [14, 9]. Our contribution is that we have shown how to do this using HTTP, and only requiring small changes to the CDN itself. We do not need to change the CDN's caching and replication policies.

While comparing our approach with previous work is difficult due to the difference in test environment, we can make some informed conjecture based on our experiences. FastReplica's evaluation includes tests of 4-8 clients, and their per-client bandwidth drops from 5.5 Mbps with 4 clients to 3.6 Mbps with 8 clients [6]. Given that their file is broken into a small number equal-sized pieces, it would appear that the slowest node in the system would be the overall bottleneck. By using a large number of small, fixed-size pieces, CoDeploy can mitigate the effects of slow nodes, either by increasing the size of its "sliding window", or by retrying chunks that are too slow. Comparisons with Bullet [10] are harder, since most of Bullet's evaluation uses ModelNet [17], a controlled emulation environment. Our experience indicates that CoDeploy's performance bottleneck are short time-scale effects, not the predictable bandwidth limits Bullet's evaluation uses. The highest-performance evaluation of Bullet on the live Internet uses a 1.5 Mbps stream with 10 client nodes, which we show we can easily match. No information is given about the highest performance Bullet can achieve in this setup, so we cannot draw conclusions in this regard.

6 Conclusion

We have demonstrated that HTTP-based content distribution networks can be extended to efficiently handle large file distribution, an area that had traditionally been approached using different protocols with separate infrastructure. Our measurements indicate that our prototype outperforms even well-connected origin servers, and our experience suggests that tuning opportunities can be used to increase its performance and reduce its memory usage.

References

[1] S. Acharya and B. Smith. Middleman: A video caching proxy server. In *NOSSDAV'00*, 2000.

[2] Akamai Technologies Inc., 1995. http://www.akamai.com/.

[3] M. Beck, D. Arnold, A. Bassi, F. Berman, H. Casanova, J. Dongarra, T. Moore, G. Obertelli, J. Plank, M. Swany, S. Vadhiyar, and R. Wolski. Logistical computing and internetworking: Middleware for the use of storage in communication. In *3rd Annual International Workshop on Active Middleware Services (AMS)*, San Francisco, August 2001.

[4] E. Bommaiah, K. Guo, M. Hofmann, and S. Paul. Design and implementation of a caching system for streaming media over the internet. In *IEEE Real Time Technology and Applications Symposium*, 2000.

[5] M. Castro, P. Drushcel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth content distribution in a cooperative environment. In *Proceedings of IPTPS'03*, Feb 2003.

[6] L. Cherkasova and J. Lee. FastReplica: Efficient large file distribution within content delivery networks. In *Proceedings of the 4th USITS*, Seattle, WA, March 2003.

[7] B. Cohen. Bittorrent, 2003. http://bitconjurer.org/BitTorrent.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, June 1999.

[9] L. Kontothanassis, R. Sitaraman, J. Wein, D. Hong, R. Kleinberg, B. Mancuso, D. Shaw, and D. Stodolsky. A transport layer for live streaming in a content delivery network. In *Proceedings of the IEEE*, volume 92, pages 1408 – 1419, 2004.

[10] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *Proceedings of the 19th ACM SOSP*, 2003.

[11] V. Pai, A. Cox, V. Pai, and W. Zwaenepoel. A flexible and efficient application programming interface for a customizable proxy cache. In *Proceedings of the 4th USITS*, Seattle, WA, March 2003.

[12] R. Rejaie and J. Kangasharju. Mocha: A quality adaptive multimedia proxy cache for internet streaming. In *NOSSDAV'01*, 2001.

[13] R. Rivest. The MD5 message-digest algorithm. RFC 1321, April 1992.

[14] R. Sitaraman. Streaming content delivery networks. Keynote address in *the 12th International Packetvideo Workshop*, 2002. http://amp.ece.cmu.edu/packetvideo2002/keynote_speakers.htm.

[15] D. Thaler and C. Ravishankar. Using Name-based Mappings to Increase Hit Rates. In *IEEE/ACM Transactions on Networking*, volume 6, 1, 1998.

[16] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, 1999.

[17] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[18] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Proceedings of the USENIX Annual Technical Conference*, 2004.