(Re)Design Considerations for Scalable Large-File Content Distribution

Brian Biskeborn, Michael Golightly*, KyoungSoo Park, and Vivek S. Pai
Department of Computer Science
Princeton University

Abstract

The CoBlitz system was designed to provide efficient large file transfer in a managed infrastructure environment. It uses a content distribution network (CDN) coupled with a swarm-style chunk distribution system to reduce the bandwidth required at origin servers. With 6 months of operation, we have been able to observe its behavior in typical usage, and glean information on how it could be redesigned to better suit its target audience.

At its heart, this paper describes what happens when a plausible conceptual design meets the harsh realities of life on the Internet. We describe our experiences improving CoBlitz's performance via a range of techniques, including measurement-based feedback, heuristic changes, and new algorithms. In the process, we triple CoBlitz's performance, and we reduce the load it places on origin servers by a factor of five. In addition to improving performance for CoBlitz's users, we believe that our experiences will also be beneficial to other researchers working on large-file transfer and content distribution networks.

1 Introduction

Content distribution networks (CDNs) use distributed sets of HTTP proxies to serve and cache popular web content. They increase perceived web browsing speed by caching content at the edges of the network (close to end users), but they also provide a high degree of reliability when asked to serve very popular pages (by spreading load across many proxies instead of concentrating it on a single origin server). Large files such as movie trailers and free software ISO images are another popular form of content on the Internet, and they can place great strains on servers and network connections. CoBlitz is a service which layers efficient large-file distribution capabilities on top of the PlanetLab-based CoDeeN [15] CDN.

In a CDN such as CoDeeN, which runs on shared hosts owned by many different companies and educational institutions, the network infrastructure is very heterogeneous. Sites display a wide range of Internet connectivity, with available bandwidths ranging from a few hundred Kbps to almost 100 Mbps. In addition to this nonuniformity, the Internet itself is a hostile environment: TCP can take care of packet loss, but occasional congestion on high-capacity paths can slow data transfers to a crawl. In a system like CoBlitz, where large file requests are spread over numerous Web proxies, a few very slow downloads can have a significant impact on the overall download speed.

In this work, we describe the optimizations made to CoBlitz to improve its performance in the nonuniform environment of PlanetLab. Our changes have produced a significant increase in throughput, thus providing benefits for both public users and researchers. Futhermore, we think the lessons learned from CoBlitz apply, at least in part, to any project which aims to distribute content quickly and efficiently.

2 Background

From a user's perspective, CoBlitz provides a simple mechanism for distributing large files, by simply prefacing their URLs with a CoBlitz-enabling host name and port number. From an internal design perspective, CoBlitz is virtually the same as the CoDeploy system [9], which was designed to provide file synchronization across PlanetLab. Both systems use the same infrastructure for transferring data, which is layered on top of the CoDeeN content distribution network [15].

To briefly summarize CoDeeN's organization, each node operates independently, from peer selection to forwarding logic. Nodes periodically exchange heartbeats, which carry local node health information. The timings of these heartbeats allows nodes to determine network health as well as node overload. Nodes independently select peers using these heartbeats. All nodes also act as caches, and use the Highest Random Weight (HRW) algorithm [13] to determine which peer should receive requests that cannot be satisfied from the node's local cache. If a forwarded request cannot be satisfied from the cache, the peer contacts the origin server to fetch the object, instead of forwarding it yet again.

When a large file is requested from CoBlitz, it generates a stream of requests for chunks of the file, and passes these requests to CoDeeN. These requests are spread across the CoDeeN nodes, which will either serve them from cache, or will forward them to the origin server. Replies from the origin server are cached at the CoDeeN nodes, and are returned to the original requestor. The details of the process are explained in our earlier work [9].

Our initial expectation for these systems was that CoDeploy would be used by PlanetLab researchers to

*Current contact: UC Irvine Computer Science Department. Work performed while a summer intern at Princeton

deploy and synchronize their experiments across nodes, while CoBlitz would be used for distributing content to the public, such as CD-ROM images. What we have found is that CoBlitz HTTP interface is quite simple to use, and can easily be integrated into deployment scripts or other infrastructure. As a result, we have seen individual researchers, other PlanetLab-based services, such as Stork [12] and PLuSH [10], and even our own group using CoBlitz to deploy and update files on PlanetLab.

This change in expected usage is significant for our design decisions, because it affects both the caching behavior as well as the desired goals. If the user population is large and the requested files are spread across a long period of time, high aggregate throughput, possibly at the expense of individual download speed, is desirable. At the same time, even if the number of copies fetched from the origin server is not minimal, the net benefit is still large.

In comparison, if the user population is mostly PlanetLab researchers deploying experiments, then many factors in the usage scenario change: the total number of downloads per file will be on the order of the number of nodes in PlanetLab (currently 583), all downloads may start at nearly the same time, the download latency becomes more important than the aggregate capacity, and extra fetches from the origin server reduces the benefits of the system.

Our goal in this work is to examine CoBlitz's design in light of its user population, and to make the necessary adjustments to improve its behavior in these conditions. At the same time, we want to ensure that the original audience for CoBlitz, non-PlanetLab users, will not be negatively affected. Since CoBlitz and CoDeploy share the same infrastructure, we expect that CoDeploy users will also see a benefit.

3 Observations & Redesign

In this section we discuss CoBlitz's behavior, the origins of the behavior, and what changes we made to address them. We focus on three areas: peering policies, reducing origin load, and reducing latency bottlenecks.

3.1 Peering

Background – When CoBlitz sends a stream of requests for chunks of a file into CoDeeN, these requests are dispersed across that CoDeeN node's peers, so the quality of CoDeeN's peering decisions can affect CoBlitz's performance. When CoDeeN's deployment was expanded from only North American PlanetLab nodes to all PlanetLab nodes, its peering strategy was changed such that each node tries to find the 60 closest peers within a 100ms round-trip time (RTT). The choice of using at most 60 peers was so that a once-per-second heartbeat could cycle through all peers within a minute, without generating too much background traffic. While techniques such as gossip [14] could reduce this traffic, we wanted to keep the pairwise measurements, since we were also interested in link health in addition to node status. Any heartbeat aggregation scheme might miss links between all pairs of peers. The 100ms cutoff was to reduce noticeable lag in interactive settings, such as Web browsing. In parts of the world where nodes could not find 20 peers within 100ms, this cutoff is raised to 200ms and the 20 best peers are selected. To avoid a high rate of change in the peer sets, hysteresis was introduced such that a peer was replaced only if another node showed consistently better RTTs.

Problem – To our surprise, we found that nodes at the same site would often have relatively little overlap between their peer lists, which could then have negative impacts on our consistent hashing behavior. The root of the problem was a high variance in RTT estimates being reinforced by the hysteresis. CoDeeN used application-level UDP "pings" in order to see application response time at remote nodes, and the average of a node's last 4 pings was used to determine its RTT. In most cases, we observed that at least one of the four most recent pings could be significantly higher than the rest, due to scheduling issues, application delays, or other non-network causes. Whereas standard network-level pings rarely show even a 10% range of values over short periods, the application-level pings routinely vary by an order of magnitude. Due to the high RTT variances, nodes were picking a very random subset of the available peers. The hysteresis, which only allowed a peer to be replaced if another was clearly better over several samples, then provided significant inertia for the members of this initial list – nodes not on the list could not maintain stable RTTs long enough to overcome the hysteresis.

Redesign – Switching from an *average* application-level RTT to the *minimum* observed RTT (an approach also used in other systems [3, 5, 11]) and increasing the number of samples yielded significant improvement, with application-level RTTs correlating well with ping time on all functioning nodes. Misbehaving nodes still showed large application-level minimum RTTs, despite having low ping times. The overlap of peer lists for nodes at the same site increasing from roughly half to almost 90%. At the same time, we discovered that many intra-PlanetLab paths had very low latency, and restricting the peer size to 60 was needlessly constrained. We increased this limit to 120 nodes, and issued 2 heartbeats per second. Of the nodes regularly running CoDeeN, two-thirds tend to now have 100 or more peers.

3.2 Reducing Origin Load

Background – When many nodes simultaneously download a large file via CoBlitz, the origin server will receive many requests for each chunk, despite the use of consistent hashing algorithms [13] designed to have multiple nodes direct requests for the same chunk to the same peer. In environments where each node will only download the

file once (such as software installs on PlanetLab), the relative benefit of CoBlitz drops as origin load increases.

Problem – When we originally tested using 130 North American nodes all downloading the same file, each chunk was downloaded by 15 different nodes on average, thereby reducing the benefit of CoBlitz to only 8.6 times that of contacting the origin directly. This problem stemmed from two sources: divergence in the peer lists, and the intentional use of multiple peers. CoBlitz's use of multiple peers per chunk stems from our earlier measurements indicating that it produced throughput benefits for cache hits [9]. However, increasing peer replication is a brute-force approach, and we are interested in determining how to do better from a design standpoint. The peer list divergence issue is more subtle – even if peer lists are mostly similar, even a few differences between the lists can cause a small fraction of requests to be sent to "non-preferred" peers. These peers will still fetch the chunks from the origin servers, since they do not have the chunks. These fetches are the most wasteful, since the peer that gets them will have little re-use for them.

Redesign – To reduce the effects of differing peer lists without requiring explicit peer list exchange between nodes, we make the following observation: with consistent hashing, if a node receives a forwarded request, it can determine whether it concurs that it is the best node to handle the request. In practice, we can determine when a request seems to have been inappropriately forwarded to a node, and then send it to a more suitable peer. To determine whether a request should be forwarded again or not, the receiving node calculates the list of possible peers for this request via consistent hashing, as though it had received it originally. If the node is not one of the top candidates on the list, then it concludes that the request was sent from a node with a differing peer list, and forwards it along. Due to the deterministic order of consistent hashing, this approach is guaranteed to make forward progress and be loop-free. While the worst case is a number of hops linear in the number of peer groups, this case is also exponentially unlikely. Even so, we limit this approach to only one additional hop in the redirection, to avoid forwarding requests across the world and to limit any damage caused by bugs in the forwarding logic. Observations of this scheme in practice indicated that typically 3-7% of all chunks require an extra hop, so restricting it to only one additional hop appears sufficient.

3.3 Addressing Latency Bottlenecks

Background – Much of the latency in downloading a large file stems from a small subset of chunks that require much more time to download than others. The agent on each CoDeeN node that generates the stream of chunk requests is also responsible for timing the responses and retrying any chunks that are taking too long. A closer examination of the slow responses indicates that some peers are much more likely than others to be involved. These nodes result in lower bandwidth for all CoBlitz transfers, even if they may not impact aggregate capacity.

Problem – When many requests begin synchronously, many nodes will simultaneously send requests for the same chunk to the peer(s) handling that chunk, resulting in bursty traffic demands. Nodes with less bandwidth will therefore take longer to satify this bursty traffic, increasing overall latency. While random request arrivals are not as affected, we have a user population that will often check for software updates using `cron` or some other periodic tool, resulting in synchronized request arrival. Though the download agent does issue multiple requests in parallel to reduce the impact of slower chunks, its total download rate is limited by the slowest chunk in the download window. Increasing the window size only increases the buffering requirement, which is unappealing since main memory is a limited resource.

Redesign – We observe that a simple way to reduce latency is to avoid peers that are likely to cause it, rather than relying on the agent to detect slow chunks and retry them. At the same time, improvements in the retry logic of the download agent can help eliminate the remaining latency bottlenecks. We experimented with two approaches to reducing the impact of the slowest nodes – reducing their frequency in the consistent hashing algorithms, and eliminating them entirely from the peering lists. Based on our bandwidth measurements of the various peers, described in Section 4.1, we tested both approaches and decided that avoiding slow peers entirely is preferable to modifying the hashing algorithms to use them. We present a discussion of our modified algorithm, along with its benefits and weaknesses, in Section 3.4. We also opted to make our download agent slightly more aggressive, drawing on the approach used in LoCI [2]. Previously, when we decided a chunk was taking too long to download, we stopped the transfer and started a new one with a different peer. In the majority of cases, no data had begun returning on the slow chunks, so this approach made sense. We modified the download agent to allow the previous transfer to continue, and let the two transfers compete to finish. With this approach, we can be more aggressive about starting the retry process earlier, since any work performed by the current download may still be useful.

3.4 Fractional Highest Random Weight

While the standard approach for handling heterogeneous capacities in consistent hashing has been the use of virtual nodes [7], we are not aware of any existing counterpart for the Highest Random Weight (HRW) [13] hashing scheme used in CoDeeN. Our concern is that using virtual nodes increases the number of items needed in the hashing scheme, and the higher computational cost of HRW (N * logN or N * # replicas versus logN for consistent hashing) makes the resulting computational requirements

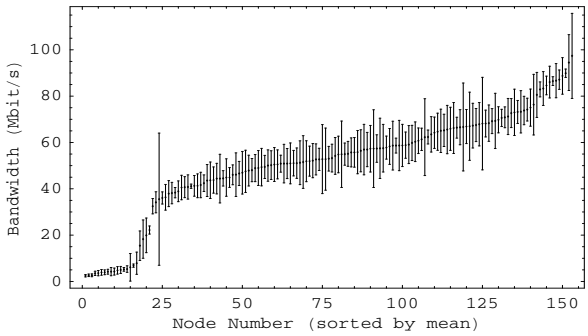

Figure 1: Mean node bandwidths & standard deviations

grow quickly. Our approach, Fractional HRW (F-HRW), does not introduce virtual nodes, and therefore requires only a modest amount of additional computation. HRW consists of three steps to assign a URL to a node: (1) hash the URL with every peer in the list, producing a set of hash values, (2) sort the peers according to these hash values, and (3) select the set of replicas with the highest values.

Our modification to HRW takes the approach of reducing the peer list based on the low-order bits of the hash value, such that peers are still included deterministically, but that their likelihood of being included on a particular HRW list is in proportion to their weight. For each peer, we assume we have a fractional weight in the range of 0 to 1, based on the expected capacity of the peer. In step (1), once we have a hash value for each node, we examine the low order bits (we arbitrarily choose 10 bits, for 1024 values), and only include the peer if the (low bits / $1024 <$ weight). We then sort as normal (or just pick the highest values via linear searches), as would standard HRW. Using the low-order bits to decide which peers to includes ensures that the decision to affect a peer is orthogonal to its rank in the sorted HRW list.

While F-HRW solves the issue of handling weights in HRW-based hashing, we find that it does not reduce latency for synchronized downloads. With F-HRW, the slow nodes do receive fewer requests *overall* versus the faster peers. However, for synchronized workloads, they still receive request bursts in short time frames, making them the download bottlenecks. For workloads where synchronization is not an issue, F-HRW can provide higher aggregate capacity, making it possibly attractive for some CDNs. However, when we examined the total capacity of the slower nodes in PlanetLab, we decided that the extra capacity from F-HRW was less valuable than the reduced latency from eliminating the slow peers entirely.

4 Evaluation

In this section, we describe our measurements of node bandwidths and of the various CoBlitz improvements.

Site (# nodes)	Node Avgs	Site Avg	Fastest
uoregon.edu (3)	2.46 - 2.66	2.59	4.63
cmu.edu (3)	3.50 - 3.95	3.67	5.74
csusb.edu (2)	3.93 - 4.21	4.07	6.76
rice.edu (3)	4.27 - 4.98	4.66	7.88
uconn.edu (2)	4.24 - 6.11	5.15	42.08

Table 1: Worst site bandwidths, measured in Mbps.

Site (# nodes)	Node Avgs	Site Avg	Slowest
neu.edu (2)	94.5 - 97.4	95.9	60.1
pitt.edu (1)	88.7	88.7	57.3
unc.edu (2)	84.6 - 87.1	85.9	66.1
rutgers.edu (2)	83.3 - 86.1	84.7	60.1
duke.edu (3)	80.5 - 89.9	84.2	59.6

Table 2: Best site bandwidths, measured in Mbps.

4.1 Measuring Node Bandwidths

To determine which peers are slow and should be excluded from CoBlitz, we perform continuous monitoring using a simple node bandwidth test. For each "edu" node on PlanetLab (corresponding to North American universities), we select the 10 closest peers, with no more than one peer per site, and synchronously start multiple TCP connections to the node from its peers. We measure the average aggregate bandwidth for a 30 second period, and repeat the test every 4 hours. Tests are run sequentially on the nodes, to avoid cross traffic that would occur with simultaneous tests. The results of 50 tests per node are shown in Figure 1. We show both the average bandwidth for each node, which ranges from 2.5 Mbps to 97.4 Mbps, as well as the standard deviation.

This straightforward testing reveals some interesting information regarding the characteristics of peak node bandwidths across these nodes: per-node bandwidth tends to be stable across time, all nodes at a site tend to be similar, and the disparities are quite large. While some nodes achieve very high bandwidths, we also observe a distinct group of poorly performing nodes that have significantly slower bandwidth speeds than the rest. There is a very large discrepancy between the best and worst sites, as outlined in Tables 1 and 2. We note that these properties are well-suited for our approach – slow nodes can be safely eliminated from consideration as peers via periodic measurements. In the event that fast nodes become slow due to congestion, the retry logic in the download agent can handle the change.

4.2 CoBlitz Improvements

To determine the effect of our redesign on CoBlitz, we measure client download times for both cached and uncached data, using various versions of the software. We isolate the impact of each design change, producing a set of seven different versions of CoBlitz. While these versions are intended to reflect our chronological changes,

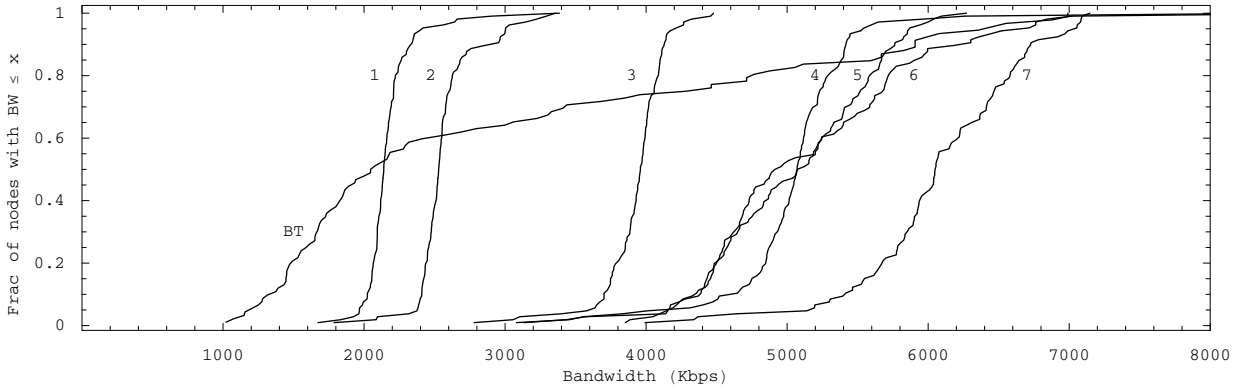

Figure 2: CDFs of mean node bandwidths for all design changes. Line numbers correspond to their entries in Table 3. Lines on the right have better bandwidths than lines on the left.

many of our changes occurred in overlapping steps, rather than a progression through seven distinct versions. In all scenarios, we use approximately 115 clients, running on North American university nodes on PlanetLab. All clients start synchronously, and download a 50 MB file located on a server at Princeton – once when the file is not cached by CoBlitz, and twice when it has already been downloaded once. We repeat each test three times and report average numbers.

Our seven test scenarios incrementally make one change at a time, and so that the final scenario represents the total of all of our modifications. The modifications are as follow: **Original** – CoBlitz as it started, with 60 peers, no exclusion of slow nodes, and the original download agent, **NoSlow** – exclude slow nodes (bandwidth $<$ 20 Mbps) from being peers, **MinRTT** – replaces the use of average RTT values with minimum RTTs, **120Peers** – raises the limit of peers to 120, **RepFactor** – reduces the replication factor from 5 peers per chunk to 2 peers, **MultiHop** – bounces misdirected requests to more suitable peers, **NewAgent** – the more aggressive download agent. The download bandwidth for all clients on the uncached test is shown in Figure 2, and the summary data is shown in Table 3. We also include a run of BitTorrent on the same set of clients, for comparison purposes.

The most obvious change in this data is the increase in mean uncached bandwidth, from 2.1 to 6.1 Mbps, which improves our most common usage scenario. The CDFs show the trends more clearly – the design changes cause a rightward shift in the CDFs, indicating improved performance. The faster strategies also yield a wider spread of node bandwidths, but a wider spread of bandwidths is probably preferable to all nodes doing uniformly poorly. Not shown in the table is the average number of nodes requesting each chunk from the origin server, which starts at 19.0, drops to 11.5 once the number of peers is increased to 120, and drops to 3.8 after the MultiHop strategy is introduced. So, not only is the uncached bandwidth almost three times the original value, but load on the origin server is reduced to one-fifth its original amount.

#	Name	uncached	cached-1	cached-2
1	Original	2.1	5.8	6.6
2	NoSlow	2.5	5.3	6.8
3	MinRTT	3.9	6.7	6.9
4	120Peers	5.0	6.2	6.6
5	RepFactor	5.0	5.5	5.4
6	MultiHop	5.2	5.2	5.6
7	NewAgent	6.1	6.5	6.7
BT	BitTorrent	2.9	–	–

Table 3: Mean bandwidths in Mbps for the various redesign steps, for both uncached and cached downloads. Also included is the value for BitTorrent, for comparison

This behavior also explains the trend in the cached bandwidths – the original numbers for the cached bandwidths are achieved through brute force, where a large number of peers are being contacted for each chunk. The initial reduction in cached bandwidth occurs because chunk downloads times become less predictable as the number of nodes serving each chunk drops. The cached bandwidths are finally restored using the more aggressive download agent, since more of the download delays are avoided by more tightly controlling retry behavior.

Note that our final version completely dominates our original version in all respects – not only is uncached bandwidth higher, but so is bandwidth on the cached tests. All of these improvements are achieved with a reduction of load to the origin server, so we feel confident that performance across other kinds of usage will also be improved. If CoBlitz traffic suddenly shifted toward non-PlanetLab users downloading large files from public Web sites, not only would they receive better performance than our original CoBlitz, but the Web sites would also receive less load.

System	# nodes	Median	Mean
CoBlitz cached	115	6.5	6.7
CoBlitz uncached	115	6.1	6.1
BitTorrent	115	2.0	2.9
Shark	185	1.0	
CoBlitz cached	41	7.3	8.1
CoBlitz uncached	41	7.1	7.4
BulletPrime	41		7.0

Table 4: Bandwidth results (in Mbps) for various systems at specified deployment sizes on PlanetLab. All measurements are for 50MB files, except for Shark, which uses 40MB.

5 Related Work

Due to space considerations, we cannot cover all related work in detail. The most obvious comparable system is BitTorrent [4], and our measurements show that we are twice as fast as it in these scenarios. Since BitTorrent was designed to handle large numbers of clients rather than high per-client performance, our results are not surprising. A more directly-related system is BulletPrime [8], which has been reported to achieve 7 Mbps when run on 41 PlanetLab hosts. In testing under similar conditions, CoBlitz achieved 7.4 Mbps (uncached) and 8.1 Mbps (cached) on average. We could potentially achieve even higher results by using a UDP-based transport protocol like Bullet's, but our current approach is TCP-friendly and is not likely to cause trigger any traffic concerns.

Finally, Shark [1], built on top of Coral [6], also performs a similar kind of file distribution, but uses the filesystem interface instead of HTTP. Shark's performance for transferring a 40MB file across 185 PlanetLab nodes shows a median bandwidth of 0.96 Mbps. Their measurements indicate that the origin server is sending the file 24 times on average in order to satisfy all 185 requests, which suggests that their performance may improve if they use techniques similar to ours to reduce origin server load. The results for all of these systems are shown in Table 4. The missing data for BulletPrime and Shark reflect the lack of information in the publications, or difficulty extracting the data from the provided graphs.

6 Conclusions

In this paper, we have shown how a detailed re-evaluation of several CDN design choices have significantly boosted the performance of CoBlitz. These design choices stemmed from two sources: our observations of our users' behavior, which differed substantially from what we had expected when launching the service, and from observing how our algorithms were behaving in practice, rather than just in theory. We believe we have learned two lessons that are broadly applicable: observe the workload your service receives to see if it can be optimized, and test the assumptions that underly your design once your service is deployed. For researchers working in content distribution networks or related areas, we believe that our experiences in hazards of peer selection and our algorithmic improvements (multi-hop, fractional HRW) may be directly applicable in other environments.

Acknowledgements

We would like to thank the anonymous reviewers for their useful feedback on the paper. This work was supported in part by NSF Grants ANI-0335214 and CNS-0439842 and by Princeton University's Summer Undergraduate Research Experience (PSURE) program.

References

[1] S. Annapureddy, M. J. Freedman, and D. Mazires. Shark: Scaling file servers via cooperative caching. In *2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, May 2005.

[2] 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)*, 2001.

[3] L. Brakmo, S. O'Malley, and L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the SIGCOMM '94 Symposium*, 1994.

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

[5] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the ACM SIGCOMM '04 Conference*, Portland, Oregon, August 2004.

[6] M. J. Freedman, E. Freudenthal, and D. Mazires. Democratizing content publication with coral. In *1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, 2004.

[7] D. Karger, E. Lehman, T. 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 *ACM Symposium on Theory of Computing*, 1997.

[8] D. Kostic, R. Braud, C. Killian, E. Vandekieft, J. W. Anderson, A. C. Snoeren, and A. Vahdat. Maintaining high bandwidth under dynamic network conditions. In *Proceedings of 2005 USENIX Annual Technical Conference*, 2005.

[9] K. Park and V. Pai. Deploying Large File Transfer on an HTTP Content Distribution Network. In *Proceedings of the First Workshop on Real, Large Distributed Systems(WORLDS '04)*, 2004.

[10] PLuSH. http://sysnet.ucsd.edu/projects/plush/.

[11] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[12] Stork. http://www.cs.arizona.edu/stork/.

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

[14] W. Vogels, R. van Renesse, and K. Birman. Using epidemic techniques for building ultra-scalable reliable communications systems. In *Workshop on New visions for Large-Scale Networks: Research and Applications*, Vienna, VA, 2001.

[15] 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.