Scale and Performance in the CoBlitz Large-File Distribution Service

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

Abstract

Scalable distribution of large files has been the area of much research and commercial interest in the past few years. In this paper, we describe the CoBlitz system, which efficiently distributes large files using a content distribution network (CDN) designed for HTTP. As a result, CoBlitz is able to serve large files without requiring any modifications to standard Web servers and clients, making it an interesting option both for end users as well as infrastructure services. Over the 18 months that CoBlitz and its partner service, CoDeploy, have been running on PlanetLab, we have had the opportunity to observe its algorithms in practice, and to evolve its design. These changes stem not only from observations on its use, but also from a better understanding of their behavior in real-world conditions. This utilitarian approach has led us to better understand the effects of scale, peering policies, replication behavior, and congestion, giving us new insights into how to better improve their performance. With these changes, CoBlitz is able to deliver in excess of 1 Gbps on PlanetLab, and to outperform a range of systems, including research systems as well as the widely-used BitTorrent.

1 Introduction

Many new content distribution networks (CDNs) have recently been developed to focus on areas not generally associated with “traditional” Web (HTTP) CDNs. These systems often focus on distributing large files, especially in flash crowd situations where a news story or software release causes a spike in demand. These new approaches break away from the “whole-file” data transfer model, the common access pattern for Web content. Instead, clients download pieces of the file (called chunks, blocks, or objects) and exchange these chunks with each other to form the complete file. The most widely used system of this type is BitTorrent [12], while related research systems include Bullet [20], Shark [2], and FastReplica [9].

Using peer-to-peer systems makes sense when the window of interest in the content is short, or when the content provider cannot afford enough bandwidth or CDN hosting costs. However, in other scenarios, a managed CDN service may be an attractive option, especially for businesses that want to offload their bandwidth but want more predictable performance. The problem arises from the fact that HTTP CDNs have not traditionally handled this kind of traffic, and are not optimized for this workload. In an environment where objects average 10KB, and where whole-file access is dominant, suddenly introducing objects in the range of hundreds of megabytes may have undesirable consequences. For example, CDN nodes commonly cache popular objects in main memory to reduce disk access, so serving several large files at once could evict thousands of small objects, increasing their latency as they are reloaded from disk.

To address this problem, we have developed the CoBlitz large file transfer service, which runs on top of the CoDeeN content distribution network, an HTTP-based CDN. This combination provides several benefits: (a) using CoBlitz to serve large files is as simple as changing its URL – no rehosting, extra copies, or additional protocol support is required; (b) CoBlitz can operate with unmodified clients, servers, and tools like curl or wget, providing greater ease-of-use for users and for developers of other services; (c) obtaining maximum per-client performance does not require multiple clients to be downloading simultaneously; and (d) even after an initial burst of activity, the file stays cached in the CDN, providing latecomers with the cached copy.

From an operational standpoint, this approach of running a large-file transfer service on top of an HTTP content distribution network also has several benefits: (a) given an existing CDN, the changes to support scalable large-file transfer are small; (b) no dedicated resources need to be devoted for the large-file service, allowing it to be practical even if utilization is low or bursty; (c) the algorithmic changes to efficiently support large files also benefit smaller objects.

Over the 18 months that CoBlitz and its partner service, CoDeploy, have been running on PlanetLab, we have had the opportunity to observe its algorithms in practice, and to evolve its design, both to reflect its actual use, and to better handle real-world conditions. This utilitarian approach has given us a better understanding of the effects of scale, peering policies, replication behavior, and congestion, giving us new insights into how to improve performance and reliability. With these changes, CoBlitz is able to deliver in excess of 1 Gbps on PlanetLab, and to outperform a range of systems, including research systems as well as BitTorrent.

In this paper, we discuss what we have learned in the process, and how the observations and feedback from long-term deployment have shaped our system. We discuss how our algorithms have evolved, both to improve performance and to cope with the scalability aspects of our system. Some of these changes stem from observing the real behavior of the system versus the abstract underpinnings of our original algorithms, and others from observing how our system operates when pushed to its limits. We believe that our observations will be useful for three classes of researchers: (a) those who are considering deploying scalable large-file transfer services; (b) those trying to understand how to evaluate the performance of such systems, and; (c) those who are trying to capture salient features of real-world behavior in order to improve the fidelity of simulators and emulators.

2 Background

In this section, we provide general information about HTTP CDNs, the problems caused by large files, and the history of CoBlitz and CoDeploy.

2.1 HTTP Content Distribution Networks

Content distribution networks relieve Web congestion by replicating content on geographically-distributed servers. To provide load balancing and to reduce the number of objects served by each node, they use partitioning schemes, such as consistent hashing [17], to assign objects to nodes. CDN nodes tend to be modified proxy servers that fetch files on demand and cache them as needed. Partitioning reduces the number of nodes that need to fetch each object from the origin servers (or other CDN nodes), allowing the nodes to cache more objects in main memory, eliminating disk access latency and improving throughput.

In this environment, serving large files can cause several problems. Loading a large file from disk can temporarily evict several thousand small files from the in-memory cache, reducing the proxy's effectiveness. Popular large files can stay in the main memory for a longer period, making the effects more pronounced. To get a sense of the performance loss that can occur, one can examine results from the Proxy Cacheoffs [25], which show that the same proxies, when operating as "Web accelerators," can handle 3-6 times the request rate than operating in "forward mode," with much larger working sets. So, if a CDN node suddenly starts serving a data set that exceeds its physical memory, its performance will drop dramatically, and latency rises sharply. Bruce Maggs, Akamai's VP of Research, states:

> "Memory pressure is a concern for CDN developers, because for optimal latency, we want to ensure that the tens of thousands of popular objects served by each node stay in the main memory. Especially in environments where caches are deployed inside the ISP, any increase in latency caused by objects being fetched from disk would be a noticeable degradation. In these environments, whole-file caching of large files would be a concern [21]."

Akamai has a service called EdgeSuite Net Storage, where large files reside in specialized replicated storage, and are served to clients via overlay routing [1]. We believe that this service demonstrates that large files are a qualitatively different problem for CDNs.

2.2 Large-file Systems

As a result of these problems and other concerns, most systems to scalably serve large files departed from the use of HTTP-based CDNs. Two common design principles are evident in these systems: treat large files as a series of smaller chunks, and exchange chunks between clients, instead of always using the origin server. Operating on chunks allows finer-grained load balancing, and avoids the trade-offs associated with large-file handling in traditional CDNs. Fetching chunks from other peers not only reduces load on the origin, but also increases aggregate capacity as the number of clients increases.

We subdivide these systems based on their inter-client communication topology. We term those that rely on greedy selection or all-to-all communication as examples of the *swarm* approach, while those that use tree-like topologies are termed *stream* systems.

Swarm systems, such as BitTorrent [12] and FastReplica [9], preceded stream systems, and scaled despite relatively simple topologies. BitTorrent originally used a per-file centralized directory, called a tracker, that lists clients that are downloading or have recently downloaded the file. Clients use this directory to greedily find peers that can provide them with chunks. The newest BitTorrent can operate with tracker information shared by clients. In FastReplica, all clients are known at the start, and each client downloads a unique chunk from the origin. The clients then communicate in an all-to-all fashion to exchange chunks. These systems reduce link stress compared to direct downloads from the origin, but some chunks may traverse shared links repeatedly if multiple clients download them.

The stream systems, such as ESM [10], SplitStream [8], Bullet [20], Astrolabe [30], and FatNemo [4] address the issues of load balancing and link stress by optimizing the peer-selection process. The result generates a tree-like topology (or a mesh or gossip-based network inside the tree), which tends to stay relatively stable during the download process. The effort in tree-building can

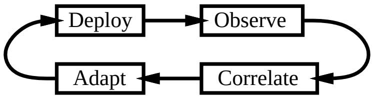


Figure 1: Operational model for CoBlitz improvement

produce higher aggregate bandwidths, suitable for transmitting the content simultaneously to a large number of receivers. The trade-off, however, is that the higher link utilization is possible only with greater synchrony. If receivers are only loosely synchronized and chunks are transmitted repeatedly on some links, the transmission rate of any subtrees using those nodes also decreases. As a result, these systems are best suited for synchronous activity of a specified duration.

2.3 CoBlitz, CoDeploy, and CoDeeN

This paper discusses our experience running two large-file distribution systems, CoBlitz and CoDeploy, which operate on top of the CoDeeN content distribution network. CoDeeN is a HTTP CDN that runs on every available PlanetLab node, with access restrictions in place to prevent abuse and to comply with hosting site policies. It has been in operation for nearly three years, and currently handles over 25 million requests per day. To use CoDeeN, clients configure their browsers to use a CoDeeN node as a proxy, and all of their Web traffic is then handled by CoDeeN. Note that this behavior is only part of CoDeeN as a policy decision – CoBlitz does not require changing any browser setting.

Both CoBlitz and CoDeploy use the same infrastructure, which we call CoBlitz in the rest of this paper for simplicity. The main difference between the two is the access mechanism – CoDeploy requires the client to be a PlanetLab machine, while CoBlitz is publicly accessible. CoDeploy was launched first, and allows PlanetLab researchers to use a local instance of CoDeeN to fetch experiment files. CoBlitz allows the public to access CoDeploy by providing a simpler URL-based interface. To use CoBlitz, clients prepend the original URL with `http://coblitz.codeen.org:3125/` and fetch it like any other URL. A customized DNS server maps the name coblitz.codeen.org to a nearby PlanetLab.

In 18 months of operation, the system has undergone three sets of changes: scaling from just North American PlanetLab nodes to all of PlanetLab, changing the algorithms to reduce load at the origin server, and changing the algorithms to reduce overall congestion and increase performance. Our general mode of operation is shown in Figure 1, and consists of four steps: (1) deploy the system, (2) observe its behavior in actual operation, (3) determine how the underlying algorithms, when exposed to the real environment, cause the behaviors, and (4) adapt the algorithms to make better decisions using the real-world data. We believe this approach has been absolutely critical to our success in improving CoBlitz, as we describe later in this paper.

3 Design of Large File Splitting

Before discussing CoBlitz's optimizations, we first describe how we have made HTTP CDNs amenable to handling large files. Our approach has two components: modifying large file handling to efficiently support them on HTTP CDNs, and modifying the request routing for these CDNs to enable more swarm-like behavior under heavy load. Though we build on the CoDeeN CDN, we do not believe any of these changes are CoDeeN-specific – they could equally be applied to other CDNs. Starting from an HTTP CDN maintains compatibility with standard Web clients and servers, whereas starting with a stream-oriented CDN might require more effort to efficiently support standard Web traffic.

3.1 Requirements

We treat large files as a set of small files that can be spread across the CDN. To make this approach as transparent as possible to clients and servers, the dynamic fragmentation and reassembly of these small files is performed inside the CDN, on demand. Each CDN node has an agent that accepts clients' requests for large files and converts them into a series of requests for pieces of the file. Pieces are specified using HTTP/1.1 byte ranges [14], which the Apache Web server has supported since August 1996 (version 1.2), and which appeared in other servers in the same timeframe. After these requests are injected into the CDN, the results are reassembled by the agent and passed to the client. For simplicity, this agent occupies a different port number than regular CoDeeN requests. The process has some complications, mostly related to the design of traditional CDNs, limitations of HTTP, and the limitations of standard HTTP proxies (which are used as the CDN nodes). Some of these problems include:

Chunk naming – If chunks are named using the original URL, all of a file's chunks will share the same name, and will be routed similarly since CDNs hash URLs for routing [16, 31]. Since we want to spread chunks across the CDN, we must use a different chunk naming scheme.

Range caching – We know of no HTTP proxies that cache arbitrary ranges of Web objects, though some can serve ranges from cached objects, and even recreate a full object from all of its chunks. Since browsers are not likely to ask for arbitrary and disjoint pieces of an object, no proxies have developed the necessary support. Since we want to cache at the chunk level instead of the file level, we must address this limitation.

original request

```
GET /file.iso
Host: www.example.com
```

↓

resulting series of requests

```
GET /file.iso,start=0,end=9999
Host: www.example.com
X-Bigfile: 1

GET /file.iso,start=10000,end=19999
Host: www.example.com
X-Bigfile: 1
...
```

Figure 2: The client-facing agent converts a single request for a large file into a series of requests for smaller files. The new URLS are only a CDN-internal representation – neither the client nor the origin server see them.

Congestion – During periods of bursty demand and heavy synchrony, consistent hashing may produce roving instantaneous congestion. If many clients at different locations suddenly ask for the same file, a lightly-loaded CDN node may see a burst of request. If the clients all ask for another file as soon as the first download completes, another CDN node may become instantly congested. This bursty congestion prevents using the aggregate CDN bandwidth effectively over short time scales.

We address these problems as a whole, to avoid new problems from piecemeal fixes. For example, adding range caching to the Squid proxy has been discussed since 1998 [24], but would expand the in-memory metadata structures, increasing memory pressure, and would require changing the internet cache protocol (ICP) used by caches to query each other. Even if we added this support to CoDeeN's proxies, it would still require extra support in the CDN, since the range information would have to be hashed along with the URL.

3.2 Chunk Handling Mechanics

We modify intra-CDN chunk handling and request redirection by treating each chunk as a real file with its own name, so the bulk of the CDN does not need to be modified. This name contains the start and end ranges of the file, so different chunks will have different hash values. Only the CDN ingress/egress points are affected, at the boundaries with the client and the origin server.

The agent takes the client's request, converts it into a series of requests for chunks, reassembles the responses, and sends it to the client. The client is not aware that the request is handled in pieces, and no browser modifications are needed. This process is implemented in a small program on each CDN node, so communication between it and the CDN infrastructure is cheap. The requests sent into the CDN, shown in Figure 2, contain extended filenames that specify the actual file and the desired byte range, as well as a special header so that the CDN modifies these requests on egress. Otherwise, these requests look like ordinary requests with slightly longer filenames. The full set of steps are shown in Figure 3, where each solid rectangle is a separate machine connected via the Internet.

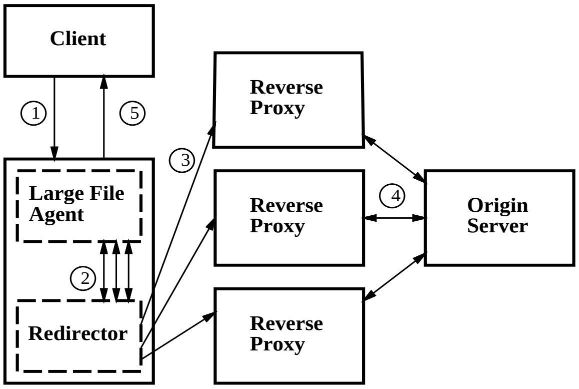
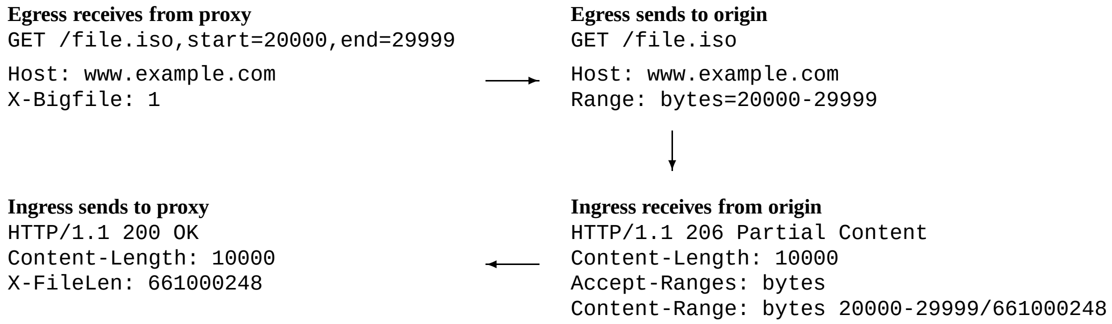


Figure 3: Large-file processing – 1. the client sends the agent a request, 2. the agent generates a series of URL-mangled chunk requests, 3. those requests are spread across the CDN, 4. assuming cache misses, the URLs are de-mangled on egress, and the responses are modified, 5. the agent collects the responses, reassembles if needed, and streams it to the client

All byte-range interactions take place between the proxy and the origin server – on egress, the request's name is reverted, and range headers are added. The server's response is changed from a HTTP 206 code (partial content received) to 200 (full file received). The underlying proxy never sees the byte-range transformations, so no range-caching support is required. Figure 4 shows this process with additional temporary headers. These headers contain the file length, allowing the agent to provide the content length for the complete download.

Having the agent use the local proxy avoids having to reimplement CDN code (such as node liveness, or connection management) in the agent, but can cause cache pollution if the proxy caches all of the agent's requests. The ingress add a cache-control header that disallows local caching, which is removed on egress when the proxy routes the request to the next CDN node. As a result, chunks are cached at the next-hop CDN nodes instead of the local node.

Since the CDN sees a large number of small file requests, 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



Egress receives from proxy

```
GET /file.iso,start=20000,end=29999
Host: www.example.com
X-Bigfile: 1
```

Egress sends to origin

```
GET /file.iso
Host: www.example.com
Range: bytes=20000-29999
```

Ingress receives from origin

```
HTTP/1.1 206 Partial Content
Content-Length: 10000
Accept-Ranges: bytes
Content-Range: bytes 20000-29999/661000248
```

Ingress sends to proxy

```
HTTP/1.1 200 OK
Content-Length: 10000
X-FileLen: 661000248
```

Figure 4: Egress and ingress transformations when the CDN communicates with the origin server. The CDN internally believes it is requesting a small file, and the egress transformation requests a byte-range of a large file. The ingress converts the server's response to a response for a complete small file, rather than a piece of a large file.

pressure, it can evict as many pieces as needed, instead of evicting one large file. Similarly, the addition/departure of nodes will only cause missing pieces to be re-fetched, instead of the whole file. The only external difference is that the server sees byte-range requests from many proxies instead of one large file request from one proxy.

3.3 Agent Design

The agent is the most complicated part of CoBlitz, since it must operate smoothly, even in the face of unpredictable CDN nodes and origin servers outside our control. The agent monitors the chunk downloads for correctness checking and for performance. The correctness checking consists of issues such as ensuring that the server is capable of serving HTTP byte-range requests, verifying that the response is cacheable, and comparing modification headers (file length, last-modified time, etc) to detect if a file has changed at the origin during its download. In the event of problems, the agent can abort the download and return an error message to the client. The agent is the largest part of CoBlitz – it consists of 770 semicolon-lines of code (1975 lines total), versus 60-70 lines of changes for ingress/egress modifications.

To determine when to re-issue chunk fetches, the agent maintains overall and per-chunk statistics during the download. Several factors may slow chunk fetching, including congestion between the proxy and its peers, operational problems at the peers, and congestion between the peers and the origin. After downloading the first chunk, the agent has the header containing the overall file size, and knows the total number of chunks to download. It issues parallel requests up to its limit, and uses non-blocking operations to read data from the sockets as it becomes available.

Using an approach inspired by LoCI [3], slow transfers are addressed by issuing multiple requests – whenever a chunk exceeds its download deadline, the agent opens a new connection and re-issues the chunk request. The most recent request for the same chunk is allowed to continue downloading, and any earlier requests for the chunk are terminated. In this way, each chunk can have at most two requests for it in flight from the agent, a departure from LoCI where even more connections are made as the deadline approaches. The agent modifies a non-critical field of the URL in retry requests beyond the first retried request for each chunk. This field is stripped from the URL on egress, and exists solely to allow the agent to randomize the peer serving the chunk. In this way, the agent can exert some control over which peer serves the request, to reduce the chance of multiple failures within the CDN. Keeping the same URL on the first retry attempts to reduce cache pollution – in a load-balanced, replicated CDN, the retry is unlikely to be assigned to the same peer that is handling the original request.

The first retry timeout for each chunk is set using a combination of the standard deviation and exponentially-weighted moving average for recent chunks. Subsequent retries use exponential backoff to adjust the deadline, up to a limit of 10 backoffs per chunk. To bound the backoff time, we also have a hard limit of 10 seconds for the chunk timeout. The initial timeout is set to 3 seconds for the first chunk – while most nodes finish faster, using a generous starting point avoids overloading slow origin servers. In practice, 10-20% of chunks are retried, but the original fetch usually completes before the retry. We could reduce retry aggressiveness, but this approach is unlikely to cause much extra traffic to the origin since the first retry uses a different replica with the same URL.

By default, the agent sends completed chunks to the client as soon as they finish downloading, as long as all preceding chunks have also been sent. If the chunk at the head of the line has not completed downloading, no new data is sent to the client until the chunk completes. By using enough parallel chunk fetches, delays in downloading chunks can generally be overlapped with others in the pipeline. If clients that can use chunked transfer encoding provide a header in the request indicating they are capable of handling chunks in any order, the agent sends chunks as they complete, with no head-of-line blocking. Chunk position information is returned in a trailer following each chunk, which the client software can use to assemble the file in the correct order.

The choice of chunk size is a trade-off between efficiency and latency – small chunks will result in faster chunk downloads, so slower clients will have less impact. However, the small chunks require more processing at all stages – the agent, the CDN infrastructure, and possibly the origin server. Larger chunks, while more efficient, can also cause more delay if head-of-line blocking arises. After some testing, we chose a chunk size of 60KB, which is large enough to be efficient, but small enough to be manageable. In particular, this chunk size can easily fit into Linux's default outbound kernel socket buffers, allowing the entire chunk to be written to the socket with a single system call that returns without blocking.

3.4 Design Benefits

We believe that this design has several important features that not only make it practical for deployment now, but will continue to make it useful in the future:

No client synchronization – Since chunks are cached in the CDN when first downloaded, no client synchronization is needed to reduce origin traffic. If clients are highly synchronized, agents can use the same chunk to serve many client requests, reducing the number of intra-CDN transfers, but synchronization is not required for efficient operation.

Trading bandwidth for disk seeks – Fetching most chunks from other CDN nodes trades bandwidth for disk seeks. Given the rate of improvement of each, this trade-off will hold for the foreseeable future. Bandwidth is continually dropping in price, and disk seek times are not scaling. If this bandwidth cost is an issue, it can be billed just as regular bandwidth is billed.

Increasing chunk utility – Having all nodes store chunks makes them available to a larger population than storing the entire file at a small number of nodes. Many more nodes can now serve large files, so the total capacity is the sum of the bandwidths they have to serve clients, and the aggregate intra-CDN capacity is available to exchange chunks.

Using cheaper bandwidth – When CDN nodes communicate with each other, this bandwidth consumption is either within a LAN cluster hosting the CDN nodes, or toward the network core, away from the clients that sit at the edge of the network. Core bandwidth has been improving in price/performance more rapidly than edge bandwidth, and LAN bandwidth is virtually free, so this consumption is in the more desirable direction.

Scaling with CDN size – As CDN size increases, and aggregate physical memory increases, chunks can be replicated more widely. The net result is that desired chunks are more likely to be in nearby nodes, so link stress drops as the CDN grows.

Tunable memory consumption – Varying the number of parallel chunks downloads that are used for each client controls the memory consumption of this approach. Slower clients can be allocated fewer parallel chunks, and the aggregate number of chunks can be reduced if a node is experiencing heavy load.

In-order or out-of-order delivery – For regular browsers or other standard client software, chunks are delivered in order so that the download appears exactly like a non-CoBlitz download from the origin, and performance-hungry clients can use software that supports chunked encoding.

4 Coping With Scale

One first challenge for CoBlitz was handling scale – at the time of CoBlitz's original deployment, CoDeeN was running on all 100 academic PlanetLab node in North America. The first major scale issue was roughly quadrupling the node count, to include every PlanetLab node. In the process, we adopted three design decisions that have served us well: (a) make peering a unilateral, asynchronous decision, (b) use minimum application-level ping times when determining suitable peers, and (c) apply hysteresis to the peer set. These are described in the remainder of this section.

4.1 Unilateral, Asynchronous Peering

In CoDeeN, we have intentionally avoided any synchronized communication for group maintenance, which results in avoiding any quorum protocols, 2-phase behavior, or any group membership protocols. The motivations behind this decision were simplicity and robustness – by making every decision unilaterally and independently at each node, we avoid any situation where forward progress fails because some handshaking protocol fails. As a result, CoDeeN has been operational even in some very extreme circumstances, such as in February 2005, when a kernel bug caused the sudden, near-simultaneous failures of nodes, with more than half of all PlanetLab nodes freezing.

One side-effect of asynchronous communication is that all peering is unilateral – nodes independently pick

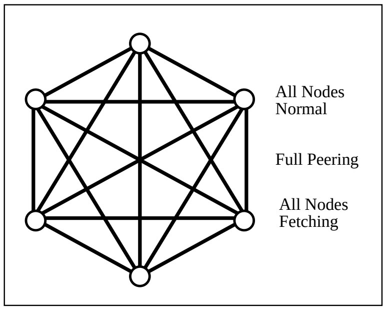


Figure 5: Standard peering for 6 unrestricted nodes

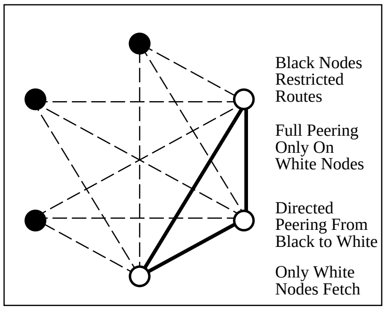


Figure 6: Peering with semi-routable Internet2

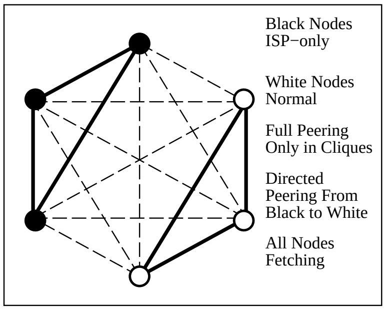


Figure 7: Peering with policy-restricted nodes

their peers, using periodic heartbeats and acknowledgments to judge peer health. Pairwise heartbeats are simple, robust, and particularly useful for testing reachability. More sophisticated techniques, such as aggregating node health information using trees, can reduce the number of heartbeats, but can lead to *worse* information, since the tree may miss or use different links than those used for pairwise communication.

Unilateral and unidirectional peering improves CoBlitz's scalability, since it allows nodes with heterogeneous connectivity or policy issues to participate to the extent possible. These scenarios are shown in Figures 5, 6, and 7. For example, research networks like Internet2 or CANARIE (the Canadian high-speed network) do not peer with the commercial Internet, but are reachable from a number of research sites including universities and corporate labs. These nodes advertise that they do not want any nodes (including each other) using them as peers, since they cannot fetch content from the commercial Internet. These sites can unidirectionally peer with any CoDeeN nodes they can reach – regular CoDeeN nodes do not reciprocate, since the restricted nodes cannot fetch arbitrary Web content. Also, in certain PlanetLab locations, both corporate and regional, political/policy considerations make the transit of arbitrary content an unwise idea, but the area may have a sizable number of nodes. These nodes advertise that only other nodes from the same organization can use them as peers. These nodes will peer both with each other and with unrestricted nodes, giving them more peers for CoBlitz transfers than they would have available otherwise. Policy restrictions are not PlanetLab-specific – ISPs host commercial CDN nodes in their network with the restriction that the CDN nodes only serve their own customers.

4.2 Peer Set Selection

With the worldwide deployment of CoDeeN, some step had to be taken to restrict the set of CoDeeN nodes that each node would use as peers. Each CoDeeN node sends one heartbeat per second to another node, so at 600+ PlanetLab nodes (of which 400+ are alive at any time), a full sweep would take 10 minutes. Using our earlier measurements that node liveness is relatively stable at short timescales [32], we limit the peer set to 60 nodes, which means that using an additional once-per-second ping, the peers can be swept once per minute.

To get some indication of application health, CoDeeN uses application-level pings, rather than network pings, to determine round trip times (RTTs). Originally, CoDeeN kept the average of the four most recent RTT values, and selected the 60 closest peers within a 100ms RTT cutoff. 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.

This approach exhibited two problems – a high rate of change in the peer sets, and low overlap among peer sets for nearby peers. The high change rate potentially impacts chunk caching in CoBlitz – if the peer that previously fetched a chunk is no longer in the peer set, the new peer that replaces it may not yet have fetched the chunk. To address this issue, hysteresis was added to the peer set selection process. Any peer not on the set could only replace a peer on the set if it was closer in two-thirds of the last 32 heartbeats. Even under the worst-case conditions, using the two-thirds threshold would keep a peer on the set for 10 minutes at a time. While hysteresis reduced peer set churn, it also reinforced the low overlap between neighboring peer sets. Further investigation indicated that CoDeeN's application-level heartbeats had more than an order of magnitude variance than network pings. This variance led to instability in the average RTT calculations, so once nodes were added to the peer set, they rarely got displaced.

Switching from an *average* application-level RTT to the *minimum* observed RTT (an approach also used in other systems [6, 13, 22]) and increasing the number of samples yielded significant improvement, with

```
∀ nodes, hash(i) = hashcalc(URL, node name(i))
hash = sort(hash)
hash = truncate(hash, NumCandidates)
∀ nodes, index(i) = node index number of hash(i)
minval = min(load(index(i)))
hash = select hash where load(index(i)) == minval
return index(random() modulo size(hash))
```

Figure 8: Replicated Highest Random Weight with Load Balancing, as used in CoDeeN

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 increased 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+ peers. More details of the redesign process and its corresponding performance improvement can be found in our previous study [5].

4.3 Scaling Larger

It is interesting to consider whether this approach could scale to a much larger system, such as a commercial CDN like Akamai. By the numbers, Akamai is about 40 times as large as our deployment, at 15,000 servers across 1,100 networks. However, part of what makes scaling to this size simpler is deploying clusters at each network point-of-presence (POP), which number only 2,500. Further, their servers have the ability to issue reverse ARPs and assume the IP addresses of failing nodes in the cluster, something not permitted on PlanetLab. With this ability, the algorithms need only scale to the number of POPs, since the health of a POP can be used instead of querying the status of each server. Finally, by imposing geographic hierarchy and ISP-level restrictions, the problem size is further reduced. With these assumptions, we believe that we can scale to larger sizes without significant problems.

5 Reducing Load & Congestion

Reducing origin server load and reducing CDN-wide congestion are related, so we present them together in this section. Origin load is an important metric for CoBlitz, because it determines CoBlitz's caching benefit and impacts the system's overall performance. From a content provider's standpoint, CoBlitz would fetch only a single copy of the content, no matter what the demand. However, for reasons described below, this goal may not be practical.

5.1 The HRW Algorithm

CoDeeN uses the Highest Random Weight (HRW) [29] algorithm to route requests from clients. This algorithm is functionally similar to Consistent Hashing [17], but has some properties that make it attractive when object replication is desired [31]. The algorithm used in CoDeeN, Replicated HRW with Load Balancing, is shown in Figure 8.

For each URL, CoDeeN generates an array of values by hashing the URL with the name of each node in the peer set. It then prunes this list based on the replication factor specified, and then prunes it again so only the nodes with the lowest load values remain. The final candidate is chosen randomly from this set. Using replication and load balancing reduces hot spots in the CDN – raising the replication factor reduces the chance any node gets a large number of requests, but also increases the node's working set, possibly degrading performance.

5.2 Increasing Peer Set Size

Increasing the peer set size, as described in Section 4.2 has two effects – each node appears as a peer of many more nodes than before, and the number of nodes chosen to serve a particular URL is reduced. In the extreme, if all CDN nodes were in each others' peer sets, then the total number of nodes handling any URL would equal to *NumCandidates*. In practice, the peer sets give rise to overlapping regions, so the number of nodes serving a particular URL is tied to the product of the number of regions and *NumCandidates*.

When examining origin server load in CoBlitz, we found that nodes with fewer than five peers generate almost one-third of the traffic. Some poorly-connected sites have such high latency that even with an expanded RTT criterion, they find few peers. At the same time, few sites use them as peers, leading to them being an isolated cluster. For regular Web CDN traffic, these small clusters are not much of an issue, but for large-file traffic, the extra load these clusters cause on the origin server slows the rest of the CDN significantly. Increasing the minimum number of peers per node to 60 reduces traffic to the origin. Because of unilateral peering, this change does not harm nearby nodes – other nodes still avoid these poorly-connected nodes.

Reducing the number of replicas per URL reduces origin server load, since fewer nodes fetch copies from the origin, but it also causes more bursty traffic at those replicas if downloading is synchronized. For CoBlitz, synchronized downloads occur when developers push software updates to all nodes, or when cron-initiated tasks simultaneously fetch the same file. In these cases, demand at any node experiences high burstiness over short time scales, which leads to congestion in the CDN.

5.3 Fixing Peer Set Differences

Once other problems are addressed, differences in peer sets can also cause a substantial load on the origin server. To understand how this arises, imagine a CDN of 60 nodes, where each node does not see one peer at random. If we ask all nodes for the top candidate in the HRW list for a given URL, at least one node is likely to return the candidate that would have been the second-best choice elsewhere. If we ask for the top k candidates, the set will exceed k candidates with very high probability. If each node is missing two peers at random, the union of the sets is likely to be at least k+2. Making the matter worse is that these "extra" nodes fetching from the origin also provide very low utility to the rest of the nodes – since few nodes are using them to fetch the chunk, they do not reduce the traffic at the other replicas.

To fix this problem, we observe that when a node receives a forwarded request, it can independently check to see whether it should be the node responsible for serving that request. On every forwarded request that is not satisfied from the cache, the receiving node performs its own HRW calculation. If it finds itself as one of the top candidates, it considers the forwarded request reasonable and fetches it from the origin server. If the receiver finds that it is not one of the top candidates, it forwards the request again. We find that 3-7% of chunks get re-forwarded this way in CoBlitz, but it can get as high as 10-15% in some cases. When all PlanetLab nodes act as clients, this technique cuts origin server load almost in half.

Due to the deterministic order of HRW, 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. Given the relatively low rate of chunks forwarded in this manner, restricting it to only one additional hop appears sufficient.

5.4 Reducing Burstiness

To illustrate the burstiness resulting from improved peering, consider a fully-connected clique of 120 CDN nodes that begin fetching a large file simultaneously. If all have the same peer set, then each node in the replica set *k* will receive 120/*k* requests, each for a 60KB chunk. Assuming 2 replicas, the traffic demand on each is 28.8 Mbits. Assuming a 10 Mbps link, it will be fully utilized for 3 seconds just for this chunk, and then the utilization will drop until the next burst of chunks.

The simplest way of reducing the short time-scale node congestion is to increase the number of replicas for each chunk, but this would increase the number of fetches to the origin. Instead, we can improve on the purely mesh-based topology by taking some elements of the stream-oriented systems, which are excellent for reducing link stress. These systems all build communication trees, which eliminates the need to have the same data traverse a link multiple times. While trees are an unattractive option for standard Web CDNs because they add extra latency to every request fetched from the origin, a hybrid scheme can help the large-file case, if the extra hops can reduce congestion.

We take the re-forwarding support to forward misdirected chunks, and use it to create broader routing trees in the peer sets. We change the re-forwarding logic to use a different number of replicas when calculating the HRW set, leading to a broad replica set and a smaller set of nodes that fetch from the origin. We set the *NumCandidates* value to 1 when evaluating the re-forwarding logic, while dynamically selecting the value at the first proxy. The larger replica set at the first hop reduces the burstiness at any node without increasing origin load.

To dynamically select the number of replicas, we observe that we can eliminate burstiness by spreading the requests equally across the peers at all times. With a target per-client memory consumption, we can determine how many chunks are issued in parallel. So, the replication factor is governed by the following equation:

$$replicationfactor = \frac{peersize * chunksize}{memoryconsumption} \qquad (1)$$

At 1 MB of buffer space per client, a 60KB chunk size, and 120 peers, our replication factor will be 7. We can, of course, cap the number of peers at some reasonable fraction of the maximum number of peers so that memory pressure does not cause runaway replication for the sake of load balancing. In practice, we limit the replication factor to 20% of the minimum target peer set, which yields a maximum factor of 12.

5.5 Dynamic Window Scaling

Although parallel chunk downloads can exploit multipath bandwidth and reduce the effect of slow transfers, using a fixed number of parallel chunks also has some congestion-related drawbacks which we address. When the content is not cached, the origin server may receive more simultaneous requests than it can handle if each client is using a large number of parallel chunks. For example, the Apache Web Server is configured by default to allow 150 simultaneous connection, and some sites may not have changed this value. If a CDN node has limited bandwidth to the rest of the CDN, too many parallel fetches can cause self-congestion, possibly underutilizing bandwidth, and slowing down the time of all fetches. The problem in this scenario is that too many slow chunks will cause more retries than needed.

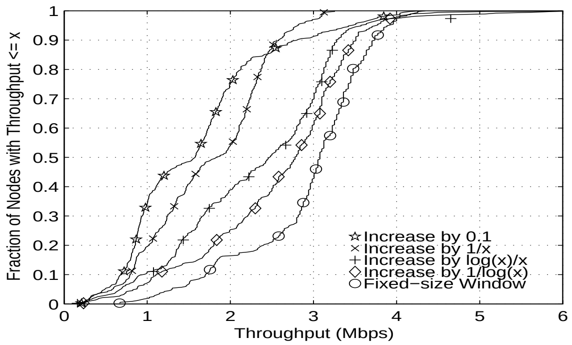


Figure 9: Throughput distribution for various window adjusting functions - Test scheme is described in section 6

In either of these scenarios, using a smaller number of simultaneous fetches would be beneficial, since the per-chunk download time would improve. We view finding the "right" number of parallel chunks as a congestion issue, and address it in a manner similar to how TCP performs congestion control. Note that changing the number of parallel chunks is not an attempt to perform low-level TCP congestion control – since the fetches are themselves using TCP, we have this benefit already. Moreover, since the underlying TCP transport is already using additive-increase multiplicative-decrease, we can choose whatever scheme we desire on top of it.

Drawing on TCP Vegas [6], we use the extra information we have in the CoBlitz agent to make the chunk "congestion window" a little more stable than a simple sawtooth. We use three criteria: (1) if the chunk finishes in less than the average time, increase the window, (2) if the first fetch attempt is killed by retries, shrink the window, and (3) otherwise, leave the window size unmodified. We also decide that if more chunk fetches are in progress than the window size dictates, existing fetches are allowed to continue, but no new fetches (including retries) are allowed. Given that our condition for increasing the window is already conservative, we give ourselves some flexibility on exactly how much to add. Similarly, given that the reason for requiring a retry might be that any peer is slow, we decide against using multiplicative decrease when a chunk misses the deadline.

While determining the decrease rate is fairly easy, choosing a reasonable increase rate required some experimentation. The decrease rate was chosen to be one full chunk for each failed chunk, which would have the effect of closing the congestion window very quickly if all of the chunks outstanding were to retry. This logic is less severe than multiplicative decrease if only a small number of chunks miss their deadlines, but can shrink the window to a single chunk within one "RTT" (in this case, average chunk download time) in the case of many failures.

Some experimentation with different increase rates is shown in Figure 9. The purely additive condition, $\frac{1}{x}$ on each fast chunk (where x is the current number of chunks allowed), fares poorly. Even worse is adding one-tenth of a chunk per fast chunk, which would be a slow multiplicative increase. The more promising approaches, adding $\frac{log(x)}{x}$ and $\frac{1}{log(x)}$ (where we use log(1) = 1) produce much better results. The $\frac{1}{x}$ case is not surprising, since it will always be no more than additive, since the window grows only when performing well. In TCP, the "slow start" phase would open the window exponentially faster, so we choose to use $\frac{1}{log(x)}$ to achieve a similar effect – it grows relatively quickly at first, and more slowly with larger windows. The chunk congestion window is maintained as a floating-point value, which has a lower bound of 1 chunk, and an upper bound as dictated by the buffer size available, which is normally 60 chunks. The final line in the graph, showing a fixed-size window of 60 chunks, appears to produce better performance, but comes at the cost of a higher node failure rate – 2.5 times as many nodes fail to complete with the fixed window size versus the dynamic sizing.

6 Evaluation

In this section, we evaluate the performance of CoBlitz, both in various scenarios, and in comparison with BitTorrent [12]. We use BitTorrent because of its wide use in large-file transfer [7], and because other research systems, such as Slurpie, Bullet' and Shark [2, 19, 26], are not running (or in some cases, available) at the time of this writing. As many of these have been evaluated on PlanetLab, we draw some performance and behavior comparisons in Section 7.

One unique aspect of our testing is the scale – we use every running PlanetLab node except those at Princeton, those designated as alpha testing nodes, and those behind firewalls that prevent CoDeeN traffic. The reason for excluding the Princeton nodes is because we place our origin server at Princeton, so the local PlanetLab nodes would exhibit unrealistically large throughputs and skew the means. During our testing in September and early October 2005, the number of available nodes that met the criteria above ranged from 360-380 at any given time, with a union size of 400 nodes.

Our test environment consists of a server with an AMD Sempron processor running at 1.5 GHz, with Linux 2.6.9 as its operating system and lighttpd 1.4.4 [18] as our web server. Our testing consists of downloading a 50MB file in various scenarios. The choice of this file size was to facilitate comparisons with other work [2, 19], which uses file sizes of 40-50MB in their testing. Our testing using a 630MB ISO image for

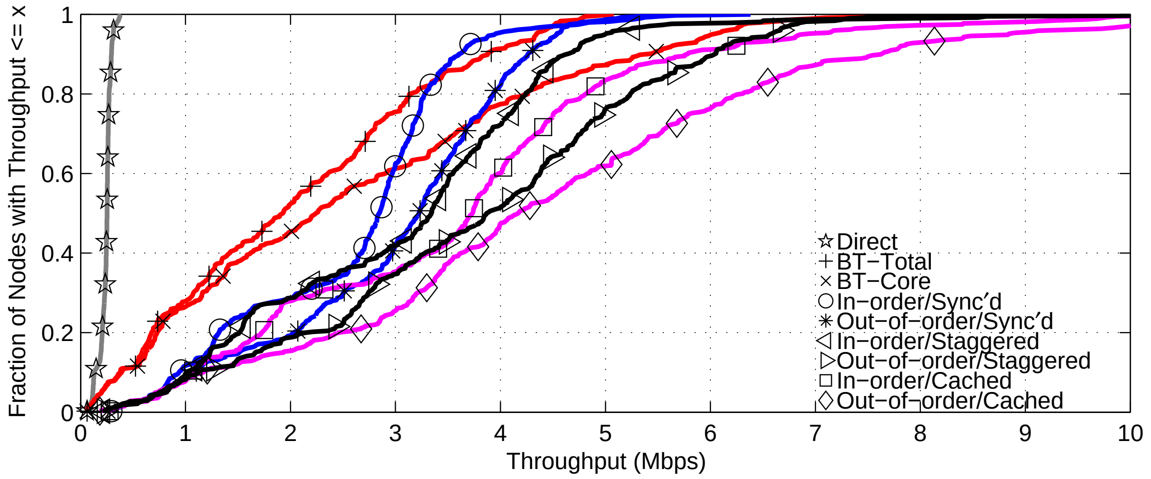


Figure 10: Achieved throughput distribution for all live PlanetLab nodes

the Fedora Core 4 download yielded slightly higher performance, but would complicate comparisons with other systems. Given that some PlanetLab nodes are in parts of the world with limited bandwidth, our use of 50MB files also reduces contention problems for them. Each test is run three times, and the reported numbers are the average value across the tests for which the node was available. Due to the dynamics of PlanetLab, over any long period of time, the set of available nodes will change, and given the span of our testing, this churn is unavoidable.

We tune BitTorrent for performance – the clients and the server are configured to seed the peers indefinitely, and the maximum number of peers is increased to 60. While live BitTorrent usage will have lower performance due to fewer peers and peers departing after downloading, we want the maximum BitTorrent performance.

We test a number of scenarios, as follows:

Direct – all clients fetch from the origin in a single download, which would be typical of standard browsers. For performance, we increase the socket buffer sizes from the system defaults to cover the bandwidth-delay product.

BitTorrent Total – this is a wall-clock timing of BitTorrent, which reflects the user's viewpoint. Even when all BitTorrent clients are started simultaneously, downloads begin at different times since clients spend different amounts of time contacting the tracker and finding peers.

BitTorrent Core – this is the BitTorrent performance from the start of the actual downloading at each client. In general, this value is 25-33% higher than the BitTorrent Total time, but is sometimes as much as 4 times larger.

In-order CoBlitz with Synchronization – Clients use CoBlitz to fetch a file for the first time and the chunks are delivered in order. All clients start at the same time.

In-order CoBlitz with Staggering – We stagger the start of each client by the same amount of time that BitTorrent uses before it starts downloading. These stagger times are typically 20 to 40 seconds, with a few outliers as high as 150-230 seconds.

In-order CoBlitz with Contents Cached – Clients ask for a file that has already been fetched previously, and whose chunks are cached at the reverse proxies. All clients begin at the same time.

Out-of-order tests – Out-of-order CoBlitz with Synchronization, Out-of-order CoBlitz with Staggering, and Out-of-order CoBlitz with Contents Cached are the same as their in-order counterparts described above, but the chunks are delivered to the clients out of order.

6.1 Overall Performance

The throughputs and download times for all tests are shown in Figure 10 and Figure 11, with summaries presented in Table 1. For clarity, we trim the x axes of both graphs, and the CDFs shown are of all nodes completing the tests. The actual number of nodes finishing each test are shown in the table. In the throughput graph, lines to the right are more desirable, while in the download time graph, lines to the left are more desirable.

From the graphs, we can see several general trends: all schemes beat direct downloading, uncached CoBlitz generally beats BitTorrent, out-of-order CoBlitz beats

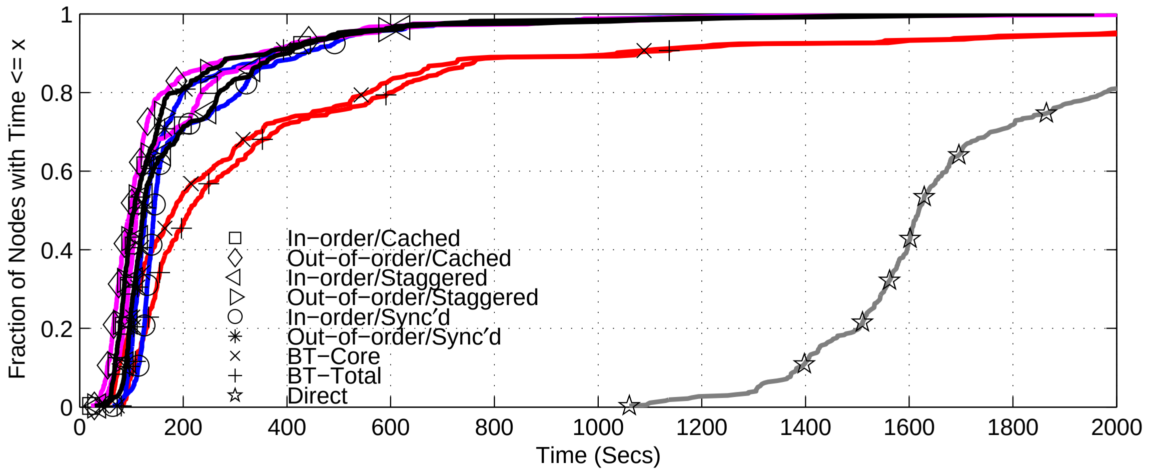


Figure 11: Download times across all live PlanetLab nodes

Strategy	Nodes		Throughput			Download Time		
	Good	Failed	Mean	50%	90%	Mean	50%	90%
Direct	372	17-18	0.23	0.20	0.38	1866.8	1618.2	3108.7
BitTorrent-Total	367	21-25	1.97	1.88	3.79	519.0	211.7	1078.3
BitTorrent-Core	367	21-25	2.52	2.19	5.32	485.1	181.1	1036.9
CoBlitz In-order Sync'd	380	8-12	2.50	2.78	3.52	222.4	143.6	434.3
CoBlitz In-order Staggered	383	5-9	2.99	3.26	4.54	122.4	141.7	406.4
CoBlitz In-order Cached	377	12-16	3.51	3.65	5.65	185.2	109.5	389.1
CoBlitz Out-of-order Sync'd	381	8-10	2.91	3.15	4.17	193.9	127.0	381.6
CoBlitz Out-of-order Staggered	384	5-8	3.68	3.78	5.91	105.4	124.6	365.2
CoBlitz Out-of-order Cached	379	8-13	4.36	4.08	7.46	164.3	98.1	349.5

Table 1: Throughputs (in Mbps) and times (in seconds) for various downloading approaches with all live PlanetLab nodes. The count of good nodes is the typical value for nodes completing the download, while the count of failed nodes shows the range of node failures.

in-order delivery, staggered downloading beats synchronized delivery, and cached delivery, even when synchronized, beats the others. Direct downloading at this scale is particularly problematic – we had to abruptly shut down this test because it was consuming most of Princeton's bandwidth and causing noticeable performance degradation.

The worst-case performance for CoBlitz occurs for the uncached case where all clients request the content at exactly the same time and more load is placed on the origin server at once. This case is also very unlikely for regular users, since even a few seconds of difference in start times defeats this problem.

The fairest comparison between BitTorrent and CoBlitz is BT-Total versus CoBlitz out-of-order with Staggering, in which case CoBlitz beats BitTorrent by 55-86% in throughput and factor of 1.7 to 4.94 in download time. Even the worst-case performance for CoBlitz, when all clients are synchronized on uncached content, generally beats BitTorrent by 27-48% in throughput and a factor of 1.47 to 2.48 in download time.

In assessing how well CoBlitz compares against BitTorrent, it is interesting to examine the 90^{th} percentile download times in Table 1 and compare them to the mean and median throughputs. This comparison has appeared in other papers comparing with BitTorrent [19, 26]. We see that the tail of BitTorrent's download times is much worse than comparing the mean or median values. As a result, systems that compare themselves primarily with the worst-case times may be presenting a much more optimistic benefit than seen by the majority of users.

It may be argued that worst-case times are important for systems that need to know an update has been propagated to its members, but if this is an issue, more important than delay is failure to complete. In Table 1, we show the number of nodes that finish each test, and these vary considerably despite the fact that the same set of machines is being used. Of the approximately 400 ma-

CoBlitz				BitTorrent
Sync		Stagger		
In-Order	Out	In-Order	Out	
7.0	7.9	9.0	9.0	10.0

Table 2: Bandwidth consumption at the origin, measured in multiples of the file size

chines available across the union of all tests, only about 5-12 nodes fail to complete using CoBlitz, while roughly 17-18 fail in direct testing, and about 21-25 fail with BitTorrent. The 5-12 nodes where CoBlitz eventually stops trying to download are at PlanetLab sites with highly-congested links, poor bandwidth, and other problems – India, Australia, and some Switzerland nodes.

6.2 Load at the Origin

Another metric of interest is how much traffic reaches the origin server in these different tests, and this information is provided in Table 2, shown as a multiple of the file size. We see that the CoBlitz scenarios fetch a total of 7 to 9 copies in the various tests, which yields a utility of 43-55 nodes served per fetch (or a cache hit rate of 97.6 - 98.2%). BitTorrent has comparable overall load on the origin, at 10 copies, but has a lower utility value, 35, since it has fewer nodes complete. For Shark, the authors observed it downloading 24 copies from the origin to serve 185 nodes, yielding a utility of 7.7. We believe that part of the difference may stem from peering policy – CoDeeN's unilateral peering approach allows poorly-connected nodes to benefit from existing clusters, while Coral's latency-oriented clustering may adversely impact the number of fetches needed.

A closer examination of fetches per chunk, shown in Figure 12, shows that CoBlitz's average of 8 copies varies from 4-11 copies by chunk, and these copies appear to be spread fairly evenly geographically. The chunks that receive only 4 fetches are particularly interesting, because they suggest it may be possible to cut CoBlitz's origin load by another factor of 2. We are investigating whether these chunks happen to be served by nodes that overlap with many peer sets, which would further validate CoBlitz's unilateral peering.

6.3 Performance after Flash Crowds

Finally, we evaluate the performance of CoBlitz after a flash crowd, where the CDN nodes can still have the file cached. This was one of motivations for building CoBlitz on top of CoDeeN – that by using an infrastructure geared toward long-duration caching, we could serve the object quickly even after demand for it drops. This test is shown in Figure 13, where clients at all PlanetLab nodes try downloading the file individually, with no two operating simultaneously. We see that performance is still good after the flash crowd has dissipated – the median for this in-order test is above 7 Mbps, almost tripling the median for in-order uncached and doubling the median of in-order cached. At this bitrate, clients can watch DVD-quality video in real time. We include BitTorrent only for comparison purposes, and we see that its median has only marginally improved in this scenario.

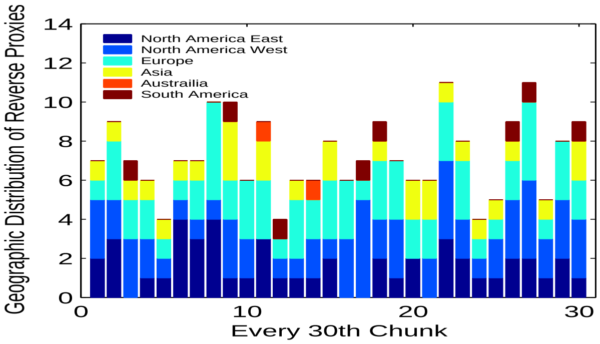


Figure 12: Reverse proxy location distribution

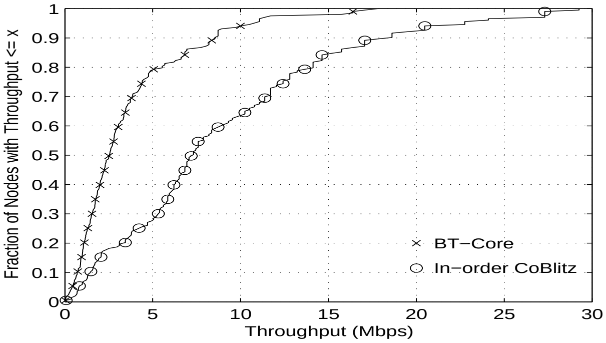


Figure 13: Single node download after flash crowds

6.4 Real-world Usage

One of our main motivations when developing CoBlitz was to build a system that could be used in production, and that could operate with relatively little monitoring. These decisions have led us not only to use simpler, more robust algorithms where possible, but also to restrict the content that we serve. To keep the system usage focused on large-file transfer with a technical focus, and to prevent general-purpose bandwidth cost-shifting, we have placed restrictions on what the general public can serve using CoBlitz. Unless the original file is hosted at a university, CoBlitz will not serve HTML files, most graphics types, and most audio/video formats. As a result of these policies, we have not received any complaints related to the content served by CoBlitz, which has simplified our operational overhead.

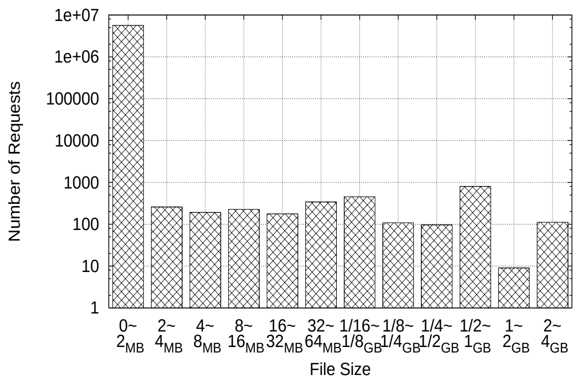


Figure 14: CoBlitz Feb 2006 usage by requests

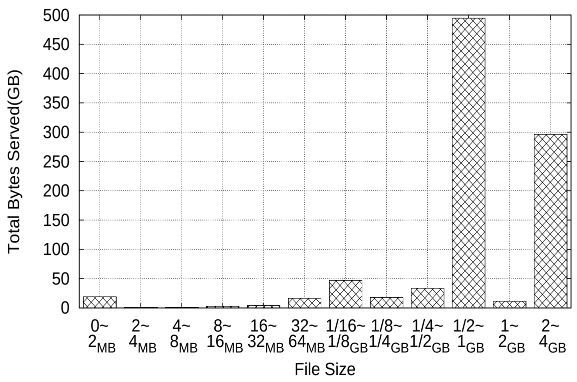


Figure 15: CoBlitz Feb 2006 usage by bytes served

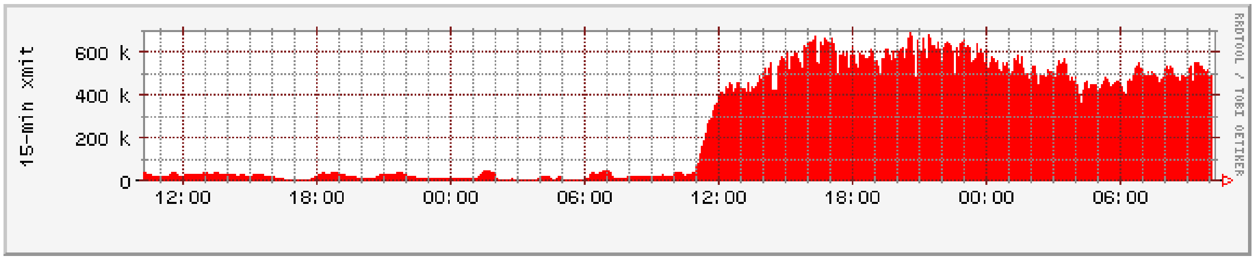


Figure 16: CoBlitz traffic in Kbps on release of Fedora Core 5, averaged over 15-minute intervals. The 5-minute peaks exceeded 700 Mbps.

To get a sense of a typical month's CoBlitz usage, we present the breakdown for February 2006 traffic in Figures 14 (by number of requests) and 15 (by bytes served). Most of the requests for files less than 2MB come from the Stork service [28], which provides package management on PlanetLab, and the CiteSeer Digital Library [11], which provides document downloads via CoBlitz. The two spikes in bytes served are from the Fedora Core Linux distribution, available as either downloadable CD images or DVD images. Most of the remaining traffic comes from smaller sites, other PlanetLab users, and Fedora Core RPM downloads.

A more unusual usage pattern occurred on March 20, 2006, when the Fedora Core 5 Linux distribution was released. Within minutes of the official announcement on the Fedora mailing lists, the availability was mentioned on the front page of Slashdot [27], on a Monday morning for the US. The measurements from this day and the previous day are shown in Figure 16. In less than an hour, CoBlitz went from an average of 20Mbps of traffic to over 400 Mbps, and sustained 5-minute peaks exceeded 700Mbps. CoBlitz functioned as expected, with one exception – many of the clients were using "download agents" that fetch files using a "no-cache" HTTP header. CoBlitz had been honoring these requests for PlanetLab researchers who wanted to force refreshes, and we had not seen a problem in other environments. However, for this download, these headers were causing unnecessary fetches to the origin that were impacting performance. We made a policy decision to disregard these headers for the Fedora Mirror sites, at which point origin traffic dropped dramatically. This flash crowd had a relatively long tail – it average 200-250Mbps on the third day, and only dropped to less than 100Mbps on the fifth day, a weekend. The memory footprint of CoBlitz was also low – even serving the CD and DVD images on several platforms (PPC, i386, x86_64), the average memory consumption was only 75MB per node.

7 Related Work

Several projects that perform large file transfer have been measured on PlanetLab, with the most closely related ones being Bullet' [19], and Shark [2], which is built on Coral [15]. Though neither system is currently accessible to the public, both have been evaluated recently. Bullet', which operates out-of-order and uses UDP, is reported to achieve 7 Mbps when run on 41 PlanetLab hosts at different sites. In testing under similar conditions, CoBlitz achieves 7.4 Mbps (uncached) and 10.6 Mbps (cached) on average. We could potentially achieve even higher re-

	Shark	CoBlitz				Bullet’
		Uncached		Cached		
		In	Out	In	Out	
# Nodes	185	41	41	41	41	41
Median	1.0	6.8	7.4	7.4	9.2	
Mean		7.0	7.4	8.4	10.6	7.0

Table 3: Throughput results (in Mbps) for various systems at specified deployment sizes on PlanetLab. All measurements are for 50MB files, except for Shark, which uses 40MB.

sults by using a UDP-based transport protocol, but our experience suggests that UDP traffic causes more problems, both from intrusion detection systems as well as stateful firewalls. Shark’s performance for transferring a 40MB file across 185 PlanetLab nodes shows a median throughput of 0.96 Mbps. As discussed earlier, Shark serves an average of only 7.7 nodes per fetch, 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 3. The missing data for Bullet’ and Shark reflect the lack of information in the publications, or difficulty extracting the data from the provided graphs.

The use of parallel downloads to fetch a file has been explored before, but in a more narrow context – Rodriguez *et al.* use HTTP byte-range queries to simultaneously download chunks in parallel from different mirror sites [23]. Their primary goal was to improve single client downloading performance, and the full file is pre-populated on all of their mirrors. What distinguishes CoBlitz from this earlier work is that we make no assumptions about the existence of the file on peers, and we focus on maintaining stability of the system even when a large number of nodes are trying to download simultaneously. CoBlitz works if the chunks are fully cached, partially cached, or not at all cached, fetching any missing chunks from the origin as needed. In the event that many chunks need to be fetched from the origin, CoBlitz attempts to reduce origin server overload. Finally, from a performance standpoint, CoBlitz attempts to optimize the memory cache hit rate for chunks, something not considered in Rodriguez’s system.

While comparing with other work is difficult due to the difference in test environment, we can make some informed conjecture based on our experiences. Fast-Replica’s evaluation includes tests of 4-8 clients, and their per-client throughput drops from 5.5 Mbps with 4 clients to 3.6 Mbps with 8 clients [9]. Given that their file is broken into a small number of equal-sized pieces, the slowest node in the system is the overall bottleneck. By using a large number of small, fixed-size pieces, CoBlitz can mitigate the effects of slow nodes, either by increasing the number of parallel fetches, or by retrying chunks that are too slow. Another system, Slurpie, limits the number of clients that can access the system at once by having each one randomly back off such that only a small number are contacting the server regardless of the number of nodes that want service. Their local-area testing has clients contact the server at the rate of one every three seconds, which staggers it far more than BitTorrent. Slurpie’s evaluation on PlanetLab provides no absolute performance numbers [26], making it difficult to draw comparisons. However, their performance appears to degrade beyond 16 nodes.

The scarcity of deployed systems for head-to-head comparisons supports part of our motivation – by reusing CDN infrastructure, we have been able to easily deploy CoBlitz and keep it running.

8 Conclusion

We show that, with a relatively small amount of modification, a traditional, HTTP-based content distribution network can be made to efficiently support scalable large-file transfer. Even with no modifications to clients, servers, or client-side software, our approach provides good performance under demanding conditions, but can provide even higher performance if clients implement a relatively simple HTTP feature, chunked encoding.

Additionally, we show how we have taken the experience gained from 18 months of CoBlitz deployment, and used it to adapt our algorithms to be more aware of real-world conditions. We demonstrate the advantages provided by this approach by evaluating CoBlitz’s performance across all of PlanetLab, where it exceeds the performance of BitTorrent as well as all other research efforts known to us.

In the process of making CoBlitz handle scale and reduce congestion both within the CDN and at the origin server, we identify a number of techniques and observations that we believe can be applied to other systems of this type. Among them are: (a) using unilateral peering, which simplifies communication as well as enabling the inclusion of policy-limited or poorly-connected nodes, (b) using request re-forwarding to reduce the origin server load when nodes send requests to an overly-broad replica set, (c) dynamically adjusting replica sets to reduce burstiness in short time scales, (d) congestion-controlled parallel chunk fetching, to reduce both origin server load as well as self-interference at slower CDN nodes.

We believe that the lessons we have learned from CoBlitz should help not only the designers of future systems, but also provide a better understanding of how to design these kinds of algorithms to reflect the unpredictable behavior we have seen in real deployment.

Acknowledgments

We would like to thank our shepherd, Neil Spring, and the anonymous reviewers for their useful feedback on the paper. This work was supported in part by NSF Grants ANI-0335214, CNS-0439842, and CNS-0520053.

References

[1] Akamai Technologies Inc., 1999. `http://www.akamai.com/`.

[2] 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.

[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] S. Birrer, D. Lu, F. E. Bustamante, Y. Qiao, and P. Dinda. FatNemo: Building a resilient multi-source multicast fat-tree. In *Proceedings of 9th International Workshop on Web Content Caching and Distribution (IWCW'04)*, Beijing, China, October 2004.

[5] B. Biskeborn, M. Golightly, K. Park, and V. S. Pai. (Re)Design considerations for scalable large-file content distribution. In *Proceedings of Second Workshop on Real, Large Distributed Systems(WORLDS)*, San Francisco, CA, December 2005.

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

[7] CacheLogic, 2004. `http://www.cachelogic.com/news/pr040715.php`.

[8] 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 SOSP'03*, Oct 2003.

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

[10] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang. A case for end system multicast. In *IEEE Journal on Selected Areas in Communication (JSAC), Special Issue on Networking Support for Multicast*, 2002.

[11] CiteSeer Scientific Literature Digital Library. `http://citeseer.ist.psu.edu/`.

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

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

[14] 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.

[15] M. J. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with coral. In *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation(NSDI'04)*, 2004.

[16] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *Proceedings of the 8th International World-Wide Web Conference*, 1999.

[17] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. 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.

[18] J. Kneschke. lighttpd. `http://www.lightttpd.net`.

[19] 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 USENIX Annual Technical Conference*, 2005.

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

[21] B. Maggs. Personal communication (email) with Vivek S. Pai, October 20th, 2005.

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

[23] P. Rodriguez, A. Kirpal, and E. W. Biersack. Parallel-access for mirror sites in the Internet. In *Proceedings of IEEE Infocom*, Tel-Aviv, Israel, March 2000.

[24] A. Rousskov. Range requests - squid mailing list. `http://www.squid-cache.org/mail-archive/squid-dev/199801/0005.html`.

[25] A. Rousskov, M. Weaver, and D. Wessels. The fourth cache-off. `http://www.measurement-factory.com/results/`.

[26] R. Sherwood, R. Braud, and B. Bhattacharjee. Slurpie: A cooperative bulk data transfer protocol. In *Proceedings of IEEE Infocom*, Hong Kong, 2004.

[27] Slashdot. `http://slashdot.org/`.

[28] Stork on PlanetLab. `http://www.cs.arizona.edu/stork/`.

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

[30] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. In *ACM Transactions on Computer Systems*, May 2003.

[31] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirecion on CDN Robustness. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation(OSDI)*, Boston, MA, December 2002.

[32] 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.