ChunkCast: An Anycast Service for Large Content Distribution

Byung-Gon Chun, Peter Wu, Hakim Weatherspoon, and John Kubiatowicz
Computer Science Division
University of California, Berkeley
{bgchun,peterwu,hweather,kubitron}@cs.berkeley.edu

ABSTRACT

Fast and efficient large content distribution is a challenge in the Internet due to its high traffic volume. In this paper, we propose ChunkCast, an anycast service that optimizes large content distribution. We present a distributed locality-aware directory that supports an efficient query for large content. Our system improves the median downloading time by at least 32% compared to previous approaches and emulates multicast trees without any explicit coordination of peers.

1. INTRODUCTION

Over the last few years there has been increasing usage of content distribution networks (CDNs) that deliver large volume data objects such as video and software. For example, approximately 70% of Tier 1 ISP traffic measured in Europe was peer-to-peer traffic [2]; yet, the considerable bandwidth consumption is not necessary to satisfy the download demand [15]. The challenge for CDNs like BitTorrent [1] that transfer large objects is to minimize the download time while reducing network bandwidth consumption.

The download time and bandwidth costs due to downloading large objects are affected by the structure of CDNs. We assume that typical CDNs partition large objects into multiple *chunks*[1] and chunks are striped among peers. A peer in the CDN acts as both a client peer and a server peer. The CDN works as follows. First, a client peer that wants to download an object performs a lookup for each object chunk. A directory that stores mappings between object chunk and server peer locations returns all of the chunk server peers. Second, for each chunk, the client peer selects a chunk server peer from which to download the chunk. Finally, the client peer publishes the chunk information into the directory so that it can later service object chunk requests as a chunk server.

There are two problems associated with the CDN structure as presented that affect download time and bandwidth consumption. First, the number of lookups to the directory can be excessive. Suppose an object is partitioned into 10,000 chunks. The client peer requires 10,000 lookups. If each chunk was replicated 10 times, a client would need to select 10,000 server peers out of 100,000 peers (in the worst case) to download chunks. Second, the structure does not give any guidance on selecting peers to download chunks (*peer selection*). Moreover, it does not give guidance on the order in which chunks to download (*chunk selection*).

In summary, the state-of-the-art CDNs for large objects have weaknesses. First, systems often do not utilize locality information for peer selection. Second, many systems do not consider the order of downloaded chunks, which can result in downloading duplicate chunks into the same local area. Finally, some systems use a central directory, which limits scalability. Our goal is to address these weaknesses.

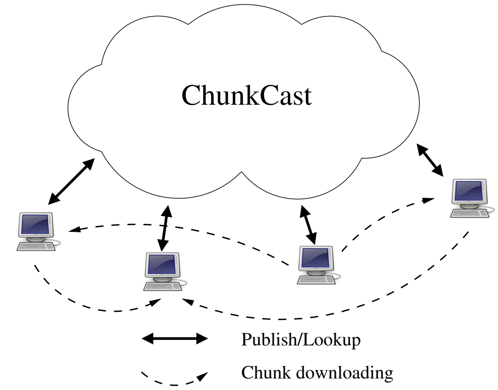


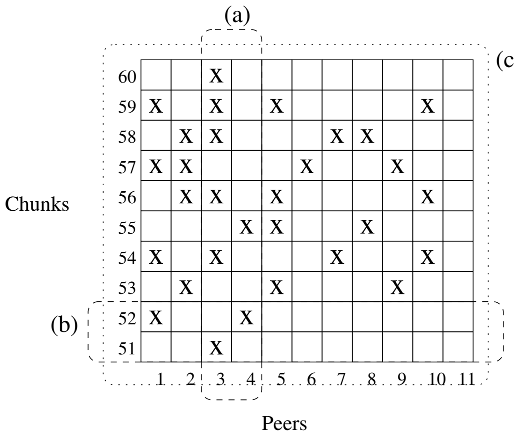
Figure 1: ChunkCast architecture

In this paper, we propose *ChunkCast*, a novel anycast service that optimizes large content distribution. Figure 1 shows the architecture of ChunkCast. Client peers publish and lookup chunks using ChunkCast and they download chunks directly from other peers. ChunkCast is a distributed locality-aware indexing structure for large content distribution networks that improves downloading time and network bandwidth consumption. The system employs an expressive anycast interface that efficiently guides client peers on peer and chunk selection choices. Since the large content distribution problem has significant impact, we claim that it is worthwhile to build a specialized indexing structure for large content. Our contributions consist of the following:

- An anycast interface exports peer locality and chunk information reducing download time and bandwidth consumption significantly.
- An efficient implementation of the anycast interface returns, in a single request, the set of server peers storing a set of chunks for a particular object.

ChunkCast builds an indexing structure on a structured overlay. ChunkCast constructs only one indexing tree per object; previous systems that create distributed indexes construct one indexing tree per chunk. For efficient publish and lookup, ChunkCast encodes a chunk as a bit in a bit vector

[1] Multi-chunk downloading improves downloading time in practice [10] and by a factor of the number of partitions in theory [21].



	1	2	3	4	5	6	7	8	9	10	11
60			X								
59	X		X		X					X	
58		X	X				X	X			
57	X	X				X			X		
56		X	X		X					X	
55				X	X			X			
54	X		X				X			X	
53		X			X				X		
52	X			X							
51			X								

Figure 2: Peer and chunk selection. The figure shows a snapshot of the chunks downloaded by peers. The peers are ordered by the distance from peer 11. The symbol "x" represents that the peer has the chunk. There are three approaches of a directory shown in the figure. Approach (a) and (b) are previous approaches and approach (c) is a new approach we propose.

where the vector represents the entire object. To exploit locality, we rely on the local convergence [7, 14, 22] property of the structured overlay and location information encoded with network coordinates [11]. Our preliminary evaluation shows that ChunkCast improves median downloading time by at least 32% when peers simultaneously download an object and emulates multicast trees without any coordination of peers.

2. SUPPORTING EFFICIENT QUERY

The goal for a large content distribution network query is to minimize both downloading time and total network bandwidth consumption. Ideally, a client peer would download each chunk from a server peer that is nearby (i.e. low network latency distance), has high available bandwidth, and is unloaded; thus, reducing the download time. Furthermore, only one client peer within a local area/region would download a chunk from a remote peer; thereby reducing the total network bandwidth consumption. We discuss what are the problems of previous approaches and propose a new appraoch below.

2.1 Previous Approaches

There are two previous approaches. In the first approach, a client peer makes a request to a directory to return a small number of server peers [1]. This approach downloads many chunks from a small number of server peers. With this approach bandwidth is wasted when nearby server peers exist that store chunks, but the server peers are not included in the set returned by the directory. As a result, chunks are downloaded from remote peers even though closer peers that store chunks exist.

In the second approach, a client peer makes a request to a directory to return a set of server peers that store a particular chunk [3, 17]. The client peer then downloads the chunk from an optimal peer. The client peer repeats the process for subsequent chunks until all chunks are downloaded. The problem with this approach is that it ignores the order with which chunks are downloaded. As a result, bandwidth is wasted when multiple nearby client peers download the same chunk from remote server peers.

Figure 2 is a snapshot of the downloading process and illustrates three different approaches: two described above and a third approach that combines the peer and chunk selection. In the network, client peer 11 looks up server peers to download chunks 51 to 60 of an object. Note that peers are ordered by the distance from client peer 11; server peer 1 is the farthest away and 10 is the closest. With approach (a), client peer 11 requests peers that download the object and the directory returns random peers (in this example, server peer 3 and 4) due to its lack of information. The query specifies only the object information. Peer 3 and 4 are far-away server peers. With approach (b), client peer 11 requests a set of server peers hosting specific chunks 51 and 52. The directory returns 1, 3, and 4, or a subset filtered by some ranking function. The server peers are also located far away from client peer 11. This result comes from the restriction of chunks client peer 11 lookups. Ideally, we want a directory that can return nearby server peers that store any chunk peer 11 needs. With approach (c), the directory can decide which peers to return from the entire (peer, chunk) points. In this case, the directory can return server peer 9 and 10, which are close to client peer 11. We call this approach an *object-level chunk anycast*.

2.2 Object-level Chunk Anycast Query

Ideally the client peer wants to choose the best (peer, chunk) point shown in Figure 2(c), where the chunk is one of the chunks the client peer needs to download. If the client peer performs parallel downloading, it wants to choose the best p (peer, chunk) points. The client peer wants to know a peer among the peers having chunks that has high bandwidth, spends small overall network bandwidth, and has many chunks it needs to download.

To achieve the goal, we first extend our query to cover a set of chunks and to return server peers that satisfy certain conditions. Our query is as follows:

> *Find a set of server peers, ranked by some ranking function R, that store any one of the chunks (C_1, ..., C_n) of an object O.*

There can be many server peers that store any one of the chunks (C_1, ..., C_n), so the directory uses a ranking function R to return the most relevant results back to the client peer. In particular, we support an object-level chunk "anycast" that returns, in a single request, the nearby peers having the largest set of chunks a client peer needs. Traditional approaches separate the chunk selection (which chunk to download) and the peer selection (which server peer to download from), but with the anycast we combine the peer selection and the chunk selection to exploit peer locality and difference in downloaded chunks.

To implement the anycast query efficiently is challenging. Naive designs are not scalable because they can incur high traffic to maintain state about which server peers store chunks and to locate nearby server peers that store chunks that the client peer requests. We present the design of an efficient and scalable system that supports the anycast query next.

3. DESIGN OF ChunkCast

In this section, we present ChunkCast, a locality-aware directory structure that supports an object-level chunk anycast query efficiently. ChunkCast is a distributed directory built on top of a structured overlay such as Bamboo [18].

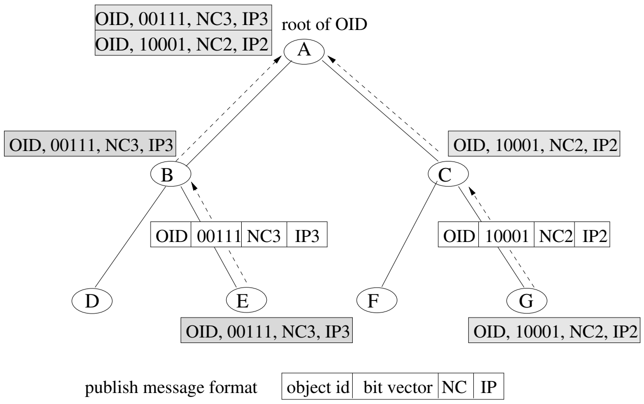


Figure 3: Chunk publication. Two nodes send publish messages whose fields are the object identifier, the bit vector representing chunk indexes to publish, the original node's network coordinate (NC), and the IP address. This information is cached by the nodes in the path from the publishing node to the root.

The structured overlay consistently hashes an identifier space over a set of nodes in which each node (called a *root*) is responsible for a portion of the identifier space. A specific message whose destination is *id* is routed to the root of *id*. A node participating in the structured overlay usually maintains $O(\log N)$ state (i.e. routing table) and the path from a node to the root takes $O(\log N)$ hops. In addition, each node in the overlay maintains a network coordinate.

An object is referenced with an object identifier, which is a SHA-1 hash of the object name. Different objects are likely to have different roots. Each chunk is referenced with a chunk index together with the object identifier. To represent multiple chunks of an object compactly, we use a bit vector in our messages.[2] An *i*th bit represents an *i*th chunk. For example, when an object (of 5 chunks) identifier is 0xFE32...27, and we want to represent the 1st, 3rd, and 5th chunks, we use the name (0xFE32...27, 10101).

To support the anycast efficiently, ChunkCast maintains an implicit indexing tree per object, which saves resources. For example, if a particular object had 10,000 chunks, previous systems like Coral [12] would create 10,000 indexing trees, but ChunkCast creates one. Furthermore, if there were 100 objects, these previous systems would create one million indexing trees whereas ChunkCast creates 100.

3.1 Chunk Publication

Publishing the location of object chunks ensure that future queries are able to find server peers. The publication process needs to contain enough information such that nearby server peers storing a set of chunks can be identified. Moreover, the process needs to be scalable. We describe this process below.

A server peer storing a chunk advertises its location through a publish message to a nearby index node. The peer can use Meridian [20], OASIS [13], or pings to find a nearby index node. The publish message contains the IP address, the network coordinate (e.g., Vivaldi [11]) of the index node that the server peer initiated the publish through[3], the object identifier, and the bit vector representing chunks stored at the server peer. The index node that receives the publish message stores the state contained in the message locally in its location pointer cache.

The state stored in the location pointer cache is periodically propagated towards the root instead of being immediately forwarded, reducing the cost of the entire publish process and improving scalability. A process that runs at each index node periodically sends one message to the next hop towards the root, which is the parent node in the index tree. The message can contain the state from multiple server peers, so we amortize the cost of publishing. Furthermore, not all state is forwarded, instead the state to forward is chosen either randomly or in a round robin fashion. Once state is chosen to be forwarded, each node in the path that receives the state caches it locally. The cached information is soft-state. If the state is not refreshed within a timeout, the node removes the state from the cache. In addition, for scalability we limit the number of entries per object. If the list is full, the index node may evict an entry in order to accept a new entry. The victim entry is selected randomly or by an eviction function weighted by network coordinate distance. Even with this lossy aggregation, we can locate nearby nodes hosting chunks well due to local convergence [7, 14, 22].

Figure 3 shows an example of two server peers publishing chunks for one object with identifier OID. All nodes represented are index nodes; the server peers are not shown. To begin, node E receives a publish message from a server peer. The publish message shows that the server peer stores chunk 3, 4, and 5. Node E stores the information in its local cache. Later, node E forwards the state to node B. When B receives the state, it similarly stores the information in its local cache and forwards the state to node A. Finally, when A receives the state, it locally stores the information and does not forward state since it is the root of the object identifier OID. As another example, a different server peer initiates a publish message via index node G. The message shows that the server peer stores chunk 1 and 5. The information is stored at G, C, and A. In the end, node A maintains two index entries published from E and G.

3.2 Chunk Lookup

To find the location of chunks, a client peer sends a lookup message to a nearby index node. The client peer can use Meridian [20], OASIS [13], or pings to find a nearby index node. The message specifies an object identifier, a start chunk index to query, a bit vector to represent the requested chunks, the number of server peers (p) to return in the result, the network coordinate of the index node that the client peer initiated the lookup through, and the client peer IP address. As an option, the message carries server peers to ignore. For example, when there are 5 chunks, the start index is 3, and the bit vector is 101, the client peer is interested in the 3rd and 5th chunks. The client peer sets p; an example value is the number parallel connections the client peer maintains to reduce downloading time. Additionally, the system enforces an upper bound on p to prevent client peers from overloading the system.

The lookup response contains at most p server peers that optimize a certain ranking function, which is explained in Section 3.3. Once an index node receives a query, it searches

[2] We use a simple representation in this paper, but more compact representations are possible.

[3] We assume that peers do not have network coordinates in general. If the peers participate in the same network coordinate system, ChunkCast can use the network coordinates of the peers. Creating network coordinates over the peers is a subject of future work.

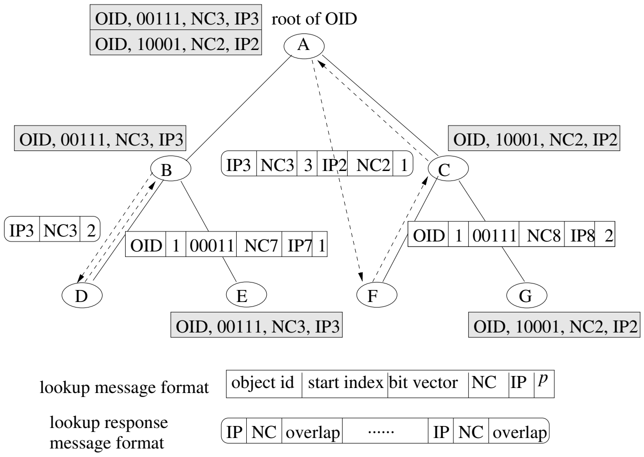


Figure 4: Chunk lookup. Two nodes send lookup messages, which consist of the object identifier, a bit vector representing chunk indexes to lookup, the original node's network coordinate (NC) and IP address, and the number of matches. The lookup on the left can be satisfied by the first local node. The lookup on the right is forwarded to the root, which returns the result, since the first local node does not contain two matches.

its own local index for server peers whose rank is among the top p. If one index node in the query path has p such server peers, the request can be satisfied by the index node. Otherwise, the index node forwards the request further toward the root. The index node excludes nodes on the list to ignore in querying its local index. When p server peer information is collected or the message arrives at the root, a response is returned to the client peer via the index node that it initiated the request through. The collection of p server peers contains peers that store at least one of the requested chunks. Subsequent lookup requests would need to be issued to find missing chunks. The result, a set of up to p records containing the IP addresses of server peers, network coordinates of index nodes the server peer published through, and the number of chunks matched, is returned back to the client peer through the index node that the client peer initially contacted.

Figure 4 shows an example of locating chunks from object OID. The information stored at index nodes A, B, C, E, and G is the same as that shown in Figure 3; no server or client peers are shown. A client peer issues a lookup via index node D. It requests to find one server peer that stores any chunk between chunks 4 and 5. Node D forwards the request to B since it does not store any index information. Node B knows a server peer that stores at least one of the requested chunks. Since B can satisfy the lookup, it returns the server peer location information to the client peer via node D and the lookup message is not forwarded further. As another example, a different client peer issues a lookup via index node F. It requests to find two server peers that store any chunk among chunks 3, 4, and 5. Node F forwards the lookup request to C. Node C knows only one server peer that satisfies the lookup request. So, it piggybacks its result to the lookup message and forwards it to A. A searches the top two nodes based on a ranking function among the nodes in its local index and the server peers piggybacked in the lookup message. Node A returns a lookup response containing two server peers back to the requesting client peer via node F. If the root node A could not find enough server peers to meet the lookup request requirements, it would have returned all the server peer information it did find.

3.3 Ranking Function

ChunkCast has two ranking functions. The first function optimizes the overlap between the bit vector of a server peer that publishes chunks and the requested bit vector of a client peers lookup query. In this case, we rely on the local convergence property to achieve locality. The second function optimizes closeness, which is defined by distance in the network coordinate space between the client peer and the server peer hosting the chunk. Exploring different ranking functions (e.g., a hybrid of the above two ranking functions) is object of future work.

4. APPLICATION

We describe how a client peer uses ChunkCast to efficiently and quickly download chunks. A client peer sends a lookup message to ChunkCast. It sets p of the message to at least the number of parallel connections it uses. ChunkCast responds with a set of p server peers that satisfy the lookup query. The lookup response, however, does not contain the chunks stored by each server peer. Instead, the client peer contacts each returned server peer to request the most up-to-date list of chunks stored. Excluding chunk information in the lookup response significantly reduces the load on ChunkCast by reducing the size of lookup results. After the client peer gathers information about the chunks that each server peer stores, it downloads chunks from multiple server peers in parallel. Finally, when downloading a chunk completes, the client peer sends a publish message to ChunkCast advertising the downloaded chunk.

A client peer reissues lookup requests. First, when a server peer has no more chunks to serve, the client peer issues a new lookup request to find a replacement. Second, since a query is very cheap in ChunkCast, the client peer periodically issues new lookup requests. When the client peer finds a new server peer closer than the server peer it downloads from, it switches from the existing server peer to the new server peer. This form of downloading emulates effective multicast distribution. If a chunk is downloaded in a local area, another peer in the same area is not likely to download the chunk.

The information obtained from ChunkCast is a hint. When the client peer contacts the server peer, it may find the server peer unresponsive; the server peer may have left the system or be slow to respond. When a server peer leaves the system, the related index entries expire after a timeout. Meanwhile, the client peer can specify to avoid such server peers in the lookup message. Therefore, the staleness of the entries does not affect the system's effectiveness. A server peer may be slow to respond if it is overloaded or the available bandwidth between the client peer and the server peer is low. Additionally, the server peer may reject the new chunk request entirely. In these cases, the client should specify these server peers as nodes to ignore in the lookup message.

5. PRELIMINARY EVALUATION

We discuss how ChunkCast performs compared to other systems. We first describe the experiment setup and present results.

5.1 Experiment Setup

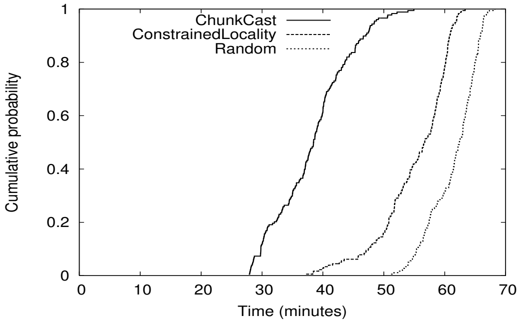


Figure 5: CDF of download time. ChunkCast improves download time of all peers. The median download time improves by at least 32%.

We have implemented ChunkCast on Bamboo [18]. ChunkCast uses Vivaldi [11] as its network coordinate system. We have also implemented peers that upload and download chunks by using ChunkCast.

We examine three systems. The first system, called *Random*, is similar to a conventional architecture, such as BitTorrent, that performs random peer selection. The root node of each chunk tracks all the locations of replicas and the location query returns a random peer hosting a replica. The second system, called *ConstrainedLocality*, is a system that uses a locality-aware structured overlay like Shark [3]. The lookup interface of the system allows querying of only one chunk at a time. The third system uses *ChunkCast* as an indexing service for large content.

We run our experiments using a cluster of 40 IBM xSeries PCs connected by Gigabit Ethernet. We use Modelnet [19] to emulate a wide-area physical topology that imposes network latency and bandwidth constraints. In addition, we use a 4000-node AS-level network with 50 distinct stubs created by the Inet topology generator [9]. The client-stub link capacity is 10Mbps, the stub-stub link capacity is 100Mbps, and the stub-transit link capacity is 45Mbps. Note that the client bandwidth is capped not to exceed 10Mbps. For all experiments, we run 180 index nodes located next to 180 peers. Running 200 peers or more is not possible for other systems compared due to overload in the cluster, although our ChunkCast can support such scale well.

In an experiment, each peer retrieves 1000 chunks, each the size of 256KB. Initially, all the chunks are placed in one node, and all the other 179 peers begin downloading chunks at the same time. The peer opens four connections to download chunks in parallel. For Random and ConstrainedLocality, the peer issues four lookup requests in parallel. For ChunkCast, the peer issues a lookup query with $p = 4$.

After downloading a chunk, the peer immediately registers the chunk information to publish to the colocated index peer and serves it upon request. Using this setup, we measure the time to download chunks, the bandwidth usage of the client, and the bandwidth usage in the network.

5.2 Results

Figure 5 shows the effectiveness of ChunkCast in supporting faster downloading of chunks. The median download time of ChunkCast decreases by 32% compared to ConstrainedLocality and decreases by 38% compared to Random. All peers download chunks faster with ChunkCast. ChunkCast helps peers effectively locate chunks that are downloaded by nearby peers.

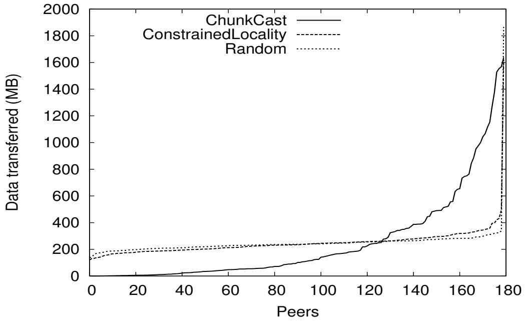


Figure 6: Bytes transferred for each peer sorted by the amount of data transferred.

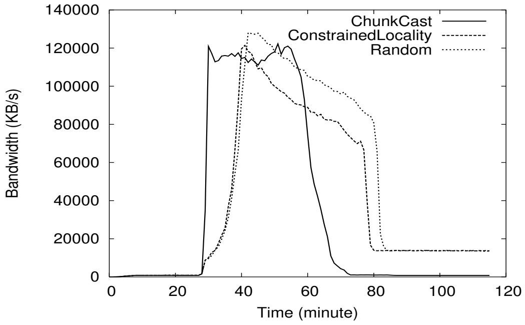


Figure 7: Bandwidth consumption over entire network links.

Figure 6 shows the data transferred by each peer, sorted by the amount of data served. With ChunkCast, some of the peers serve more data compared to ConstrainedLocality or Random. This is because ChunkCast constructs distribution paths similar to an effective multicast tree. Only one chunk is downloaded to a local area and the chunk is distributed to nodes in the local area. To lessen the skew, the peers can employ the fair-sharing mechanism like Tit-for-Tat as used in BitTorrent. However, this fairness mechanism is not a part of the anycast service, which is our focus.

We show the bandwidth consumption over entire network links in Figure 7. The bandwidth consumption of ChunkCast quickly increases to the sustainable throughput. Therefore, ChunkCast can quickly initiate downloading to a full speed. In addition, by using ChunkCast, the bandwidth consumption is sustained throughout the downloading period, allowing peers to finish downloading sooner than with the other schemes. With the other schemes, the bandwidth increases slowly and gradually decreases after it reaches peak usage. Another interesting observation is that ConstrainedLocality and Random continue to use considerable bandwidth for publishing chunk information, since a tree is constructed per chunk. On the contrary, ChunkCast uses very little bandwidth after the downloading finishes, since the chunk information is compactly represented and a tree is constructed for entire chunks.

6. RELATED WORK

The major difference between ChunkCast and previous approaches is that we support an object chunk anycast with which a client peer can efficiently find nearby peers hosting any chunk the client peer needs. In addition, we create an indexing tree per object.

One popular large content distribution program is BitTorrent [1]. The BitTorrent clients receive random peers from a central directory called a tracker, which maintains a list of peers storing objects. The directory does not guide peers in finding nearby peers that store needed chunks. Instead, the peers should find good peers by actually downloading chunks and opportunistically trying new peers.

CoBlitz is a HTTP-based CDN that splits a file into smaller chunks and caches those chunks at distributed nodes [17]. The system is based on unstructured overlay that maps each chunk to a node. In CoBlitz, the gateway node that handles clients' requests needs to query chunks in a certain order. The design of CoBlitz is revised with experience from the wide-area deployment [5]. Their new design consideration can also be applicable to peers that use ChunkCast.

Shark [3] makes use of Coral [12], which uses a distributed index over a DHT to find nearby copies of data and help reduce origin server bandwidth. Although it is distributed, Shark is limited in its ability to find nearby peers hosting chunks because clients need to query the node holding a specific chunk in a certain order.

The Julia content distribution network [4] is a system that adds locality-awareness to unstructured content distribution systems like BitTorrent. As the downloading progresses, a peer gathers information about which peers are closer than others. Then, it chooses to download more chunks from the nearby peers it learned of as the downloading progresses. This approach does not rely on an index service as ours, so a newly joined peer is forced to go through this navigation process even though there are already peers with chunks. With ChunkCast, the newly joined peer can immediately download chunks from nearby peers.

SplitStream [8] builds one multicast tree per stripe and an additional spare capacity group tree. The orphaned nodes use the spare capacity group tree to find peers that can forward stripes they want. The system employs an anycast primitive that performs depth first search in the tree. The primitive is not designed to support a query of a large number of chunks, and the search can potentially visit all nodes in the system if queried chunks are rare.

The informed content delivery [6] addresses tools to reconcile differences of downloaded coded symbols between a pair of nodes to exploit perpendicular connections. The focus of the study is to compactly approximate and to reconcile downloaded symbols from a large symbol space (e.g., one million symbols). Bullet [16] uses an overlay mesh to push data. The system distributes any data item to be equally likely to occur at any node and recovers missing items using approximate reconciliation techniques. Both systems operate to increase the discrepancy of downloaded chunks, which maximizes the effectiveness of the perpendicular connection.

7. CONCLUSION

Fast and efficient large content distribution is a challenge in the current Internet. In this paper, we argue that because of its big impact on performance and network usage, it is necessary to devise a specialized index service optimized for large content. We propose ChunkCast, an anycast service that exploits locality for large content distribution. Our preliminary evaluation shows that ChunkCast improves median downloading time by at least 32% compared to previous approaches and effectively creates a multicast type distribution when peers download chunks simultaneously. We plan to evaluate the effectiveness of such an anycast service in diverse environment settings.

REFERENCES

[1] Bittorrent. http://bittorrent.com.

[2] Cachelogic. http://www.cachelogic.com.

[3] S. Annapureddy, M. J. Freedman, and D. Mazires. Shark: Scaling file servers via cooperative caching. In *NSDI*, 2005.

[4] D. Bickson and D. Malkhi. The julia content distribution network. In *USENIX WORLDS*, 2005.

[5] B. Biskeborn, M. Golightly, K. Park, and V. S. Pai. (Re)Design considerations for scalable large-file content distribution. In *USENIX WORLDS*, 2005.

[6] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. In *SIGCOMM*, 2002.

[7] M. Castro, P. Druschel, Y. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer networks. In *Technical Report MSR-TR-2002-82*. Microsoft Research, 2002.

[8] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in cooperative environments. In *SOSP*, 2003.

[9] H. Chang, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. Towards capturing representative as-level internet topologies. In *Computer Networks Journal*, 2004.

[10] L. Cherkasova and J. Lee. Fastreplica: Efficient large file distribution within content delivery networks. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

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

[12] M. J. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with coral. In *NSDI*, 2004.

[13] M. J. Freedman, K. Lakshminarayanan, and D. Mazieres. OASIS: Anycast for any service. In *NSDI*, 2006.

[14] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *SIGCOMM*, Aug. 2003.

[15] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *SOSP*, 2003.

[16] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.

[17] K. Park and V. S. Pai. Deploying large file transfer on an http content distribution network. In *USENIX WORLDS*, 2004.

[18] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *USENIX Annual Tech. Conf.*, 2004.

[19] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *OSDI*, 2002.

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

[21] X. Yang and G. de Veciana. Service capacity of peer to peer networks. In *INFOCOM*, 2004.

[22] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. In *IEEE JSAC*, 2004.