Parallel Processing Letters, Vol. 13, No. 2 (2003) 207–223
© World Scientific Publishing Company

ALGORITHMS FOR HIGH PERFORMANCE, WIDE-AREA DISTRIBUTED FILE DOWNLOADS

JAMES S. PLANK, SCOTT ATCHLEY, YING DING and MICAH BECK
Logistical Computing and Internetworking Lab, Department of Computer Science
University of Tennessee, Knoxville, TN 37996, USA
http://loci.cs.utk.edu

Received December 2002
Revised April 2003
Accepted by J. Dongarra & B. Tourancheau

ABSTRACT

As peer-to-peer and wide-area storage systems become in vogue, the issue of delivering content that is cached, partitioned and replicated in the wide area, with high performance, becomes of great importance. This paper explores three algorithms for such downloads. The storage model is based on the Network Storage Stack, which allows for flexible sharing and utilization of writable storage as a network resource. The algorithms assume that data is replicated in various storage depots in the wide area, and the data must be delivered to the client either as a downloaded file or as a stream to be consumed by an application, such as a media player. The algorithms are threaded and adaptive, attempting to get good performance from nearby replicas, while still utilizing the far-away replicas. After defining the algorithms, we explore their performance downloading a 50 MB file replicated on six storage depots in the U.S., Europe and Asia, to two clients in different parts of the U.S. One algorithm, called *progress-driven redundancy*, exhibits excellent performance characteristics for both file and streaming downloads.

Keywords: Wide-area storage; peer-to-peer storage; adaptive downloads; scalable storage; replicated storage.

1. Introduction

Advanced, wide-area storage infrastructures are becoming increasingly in vogue [CSWH00, DKK$^+$01, PBB$^+$01, RWE$^+$01]. As with all wide-area network infrastructures, they must be able to deal gracefully and efficiently with transient, permanent and unpredictable failures, whose causes can range from administrative reasons to hardware/software failures to changing network conditions. Thus, they typically provide primitives for caching and replication, and perhaps more advanced features such as striping and erasure encoding [HO93, WK02, Pla97]. Given a variety of ways to store information on such infrastructures, the question of how to retrieve data most efficiently becomes a challenging one.

This paper provides an experimental exploration of the following question:

Given a piece of content (e.g. a file) that is striped and replicated in the wide-area, how can that content best be delivered to a client?

Standard replica-management architectures (e.g. [RWE$^+$01, CHM$^+$02]) simply have the client select a replica manager and download the entire file, or an entire stripe from that manager. However, the range of download strategies that may be employed is extremely vast, especially when the client aggressively employs multiple communication channels. In this paper, we explore three downloading algorithms that attempt to provide simple, yet effective and high-performance downloading methodologies in wide-area settings. The goal is to provide insight to designers of wide-area storage infrastructures, so that they may implement download operations that perform well in the wide area. However, the methodologies are applicable whenever data is distributed in the wide-area and needs to be delivered to a client, and thus are relevant to parallel, distributed, peer-to-peer and Grid computing. The main result of this paper is a simple algorithm called *progress-driven redundancy*, which exhibits excellent downloading performance and characteristics, including increased efficiency as replicas to data are added, and delay-minimal streaming performance.

The paper is organized as follows: In section 2 we detail the Network Storage Stack and Logistical Runtime System, which is our infrastructure testbed. This testbed has several features which make it an interesting experimental platform. In section 3 we describe the downloading strategies whose performance we explore experimentally in section 4. We conclude in section 5.

2. The Network Storage Stack and Logistical Runtime System

For our experimental infrastructure, we use the Network Storage Stack and Logistical Runtime System, developed at the University of Tennessee. The goal of the Network Storage Stack (Figure 1) is to layer abstractions of network storage that allow *writable* storage resources to be part of the wide-area network in an efficient, flexible, sharable and scalable way. Its model, which achieves all these goals for data transmission, is the IP stack, and its guiding principle has been to follow the tenets laid out by End-to-End arguments [SRC84, RSC98, BMP02]. Two fundamental principles of this layering are that each layer should (a) *abstract* the layers beneath it in a meaningful way, but (b) *expose* an appropriate amount of its own resources so that higher layers may abstract them meaningfully (see [BMP01, BMP02] for more detail on this approach).

2.1. *IBP*

The lowest layer of the storage stack that is globally accessible from the network is the *Internet Backplane Protocol (IBP)* [PBB$^+$01]. IBP is server daemon software and a client library that allows storage owners to insert their storage into the network, and to allow generic clients to allocate and use this storage. The unit of storage is a time-limited, append-only byte-array. With IBP, byte-array allocation is like a network **malloc()** call — clients may request an allocation from a specific IBP storage server (or *depot*), and if successful, are returned trios of cryptographically

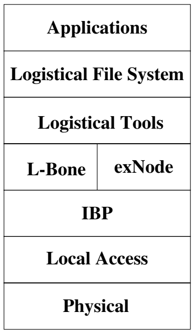

Fig. 1. The network storage stack.

secure text strings (called "capabilities") for reading, writing and management. Capabilities may be used by *any* client in the network, and may be passed freely from client to client, much like a URL.

IBP does its job as a low-level layer in the storage stack. It abstracts away many details of the underlying physical storage layers: block sizes, storage media, control software, etc. However, it also exposes many details of the underlying storage, such as network location, network transience and the ability to fail, so that these may be abstracted more effectively by higher layers in the stack.

2.2. *The L-Bone and exNode*

While individual IBP allocations may be employed directly by applications for some benefit [PBB+01], they, like IP datagrams, benefit from some higher-layer abstractions. The next layer contains the *L-Bone*, for resource discovery and proximity resolution, and the *exNode*, a data structure for aggregation. Each is defined here.

The L-Bone (Logistical Backbone) is a distributed runtime layer that allows clients to perform IBP depot discovery. IBP depots register themselves with the L-Bone, and clients may then query the L-Bone for depots that have various characteristics, including minimum storage capacity and duration requirements, and basic proximity requirements. For example, clients may request an ordered list of depots that are close to a specified city, airport, US zipcode, or network host. Once the client has a list of IBP depots, it may then request that the L-Bone use the Network Weather Service (NWS) [WSH99] to order those depots according to bandwidth predictions using live networking data. Thus, while IBP gives clients access to remote storage resources, it has no features to aid the client in figuring out which storage resources to employ. The L-Bone's job is to provide clients with those features.

The exNode is a data structure for aggregation, analogous to the Unix inode (Figure 2). Whereas the inode aggregates disk blocks on a single disk volume to compose a file, the exNode aggregates IBP byte-arrays to compose a logical entity that may be used like a file. Two major differences between exNodes and inodes

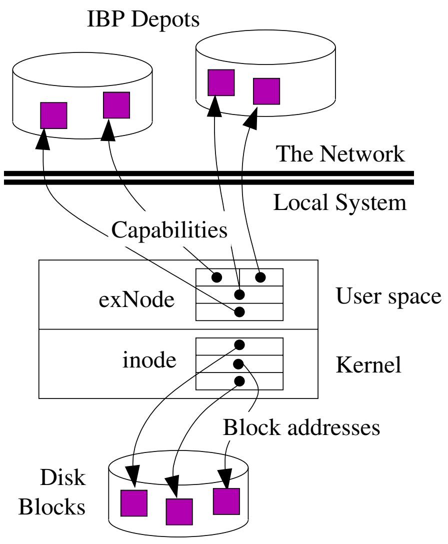

Fig. 2. The exNode in comparison to the Unix inode.

are that the IBP buffers may be of any size, and their extents may overlap and be replicated. Thus, the exNode allows users and applications to create network files out of time-limited and failure-prone IBP allocations in such a way that much stronger properties (e.g. fault-tolerance, longer durations) may be achieved. ExNodes are represented by XML encodings, manipulated by an exNode library. Like IBP capabilities, they may be passed from client to client, anywhere in the network, with no registration from a central authority.

2.3. *The logistical runtime system*

The next level in the stack are tools and a client library that compose the Logistical Runtime System (LoRS). These tools allow users to create, manipulate and use the network "files" supported by the exNode. These files reside on IBP depots located by the L-Bone. The functionalities supported by LoRS are:

- **Upload**: Create a network file from a local file, input stream or memory buffer.
- **Download**: Get the bytes from a network file and store them locally or stream them to an application.
- **Augment**: Add more replicas to a network file.
- **Trim**: Subtract replicas from a network file.
- **Refresh**: Extend the time limits of the IBP allocations.

Note that both **upload** and **augment** allow the user to stripe and replicate the file in a very flexible manner. Moreover, **augment** and **trim** allow the user to *route* the file from one network location to another.

2.4. *Status*

IBP, the L-Bone, the exNode and the Logistical Runtime System are all software supported by the Logistical Computing and Internetworking (LoCI) Laboratory (http://loci.cs.utk.edu). The power of this suite of software has been demonstrated with several applications:

- **IBP-Mail** is an application that allows users to mail large files to other users by uploading them into the network, and then mailing the exNode to the recipient.
- **IBP-ster** is an media player that plays audio and video files stored in IBP allocations on the network. The files may be arbitrarily striped and replicated, and the player performs a streaming download to play them.
- **IBPvo** is an application which users may schedule to record television programs into IBP allocations. The user is sent an exNode, which he or she may use to re-play the program from the network storage buffers.

LoCI supports a main L-Bone (http://loci.cs.utk.edu/lbone/cgi-bin/lbone_list_view.cgi) that currently is composed of 143 depots at locations in the United States, Europe, Asia, and Australia, serving over ten terabytes of network storage. The software has been designed to run without an L-Bone, or for users to configure their own, private L-Bone.

A prototype, read-only logistical file system has built, and we are exploring log-structured techniques and available software (such as Ivy [MMGC02] and Swarm [HMS99]) for implementing the Logistical File System layer of the Network Storage Stack.

3. The Challenge of Downloading

Given that context, let us now focus upon **download**. Suppose a user has an exNode file, whose contents are striped and replicated in IBP buffers spread throughout the world, and the user wants to download the entire file to local disk storage or to a streaming application as quickly as possible. What strategy should the LoRS download tool use? This is a problem that is easy to state, but hard to solve. There are a wide variety of factors that make this problem difficult, including changing network conditions, transient failures, heterogeneity in operating systems and working environments, differing buffer sizes, differing administrative decisions, etc. Add to this the Pandora's Box (see Section 3.1) of multiple TCP connections between a pair of hosts, and the complexity of implementing a download functionality that delivers optimal performance is overwhelming. However, after discussing the issue of multiple TCP streams, we present some simple downloading strategies that should deliver effective performance in a variety of settings.

3.1. *The Pandora's box of multiple TCP connections between a pair of hosts*

It is an unfortunate fact that when sending a large amount of data between a client and a server, doing so with some number of simultaneous TCP connections can vastly outperform using one connection. As a consequence, some file transfer tools, such as GridFTP [ABB+02] and bbftp [Fer02] allow the clients and servers to perform downloads with multiple simultaneous TCP streams. As an example of the possible gains, in October, 2001, Cottrell reported achieving over 100 Mbps throughput for a Trans-Atlantic file transfer using 40 simultaneous TCP streams and a window size of 64kB. With only 10 streams, the bandwidth was under 30 Mbps [Cot01].

Setting up clients and servers to use multiple streams is a simple task. IBP allows server owners to specify the maximum number of allowable simultaneous connections, and the LoRS tools allow users to specify a total number of threads to perform the download. As in Cottrell's experiments, at iGrid 2002 (September, 2002 in Amsterdam), the LoRS tools were able to demonstrate over 100 Mbps on a Trans-Atlantic download using *untuned* TCP implementations (typically 8kB windows) and over 200 threads.

The problem with multiple streams is well-documented – they circumnavigate TCP's congestion-control mechanisms, and therefore do not act well in under-provisioned or transiently congested environments [Tou95, ADG+00, EHT00, Flo00]. As a result, some institutions treat TCP-unfriendly activities, such as multiple streams, as akin to a denial-of-service attack, and may disable service to the offending client. In two separate IETF RFC's, the recommended number of multiple streams that a client should initiate to a single server is two [Flo00] and one [ADG+00] respectively. The suggestion is that single-stream TCP performance should improve in the near future, perhaps with self-modifying window sizes [DMT02].

Therefore, we have the Pandora's box: On one hand, we may achieve excellent performance with a minimum of effort by employing multiple streams, and on over-provisioned networks there are no adverse effects. However, the activity is TCP-unfriendly, and in the commodity Internet may lead to dire consequences.

As a result, in this paper we will present download algorithms that employ either one stream per client-server pair, or two. The intent is to present numbers that do not circumvent the congestion-control methodologies of TCP, but that hint toward the better performance that may currently be achieved from multiple streams. Until the networking community devises a better solution than the current one, we will have to accept the Pandora's Box as a given.

3.2. *Download algorithms*

We restrict our attention to a file of size S. This file is stored in its entirety at each of N IBP servers, denoted $I_1, \ldots, I_N$. From these IBP allocations, we create N exNodes, labeled $E_1, \ldots, E_N$, such that E_x contains all replicas from I_1 through I_x. We assume that $S \gg N$. While larger values of x would appear to be beneficial, care must be taken in how the replicas are utilized. Indeed, more replicas means more potentially parallel paths to data, however they also may increase the probability that a slow server is selected from which to download. How the algorithms handle this issue is fundamental to their performance.

Streaming Considerations

LoRS downloads can be to local files, local memory buffers, or directly to streaming applications, such as an audio or video player. Such players have quantifiable needs in terms of sustained bandwidth. For example, uncompressed audio is typically consumed at 1.38 Mbps, while a typical MP3 file requires only 0.125 Mbps. Video files are encoded up to 300 Kbps for online streaming and up to 15 Mbps for DVD quality files. Raw consumer digital video cameras output video at approximately 50 Mbps.

Like most streaming media players, the LoRS streaming download tool employs a lookahead buffer to tolerate variable network latencies. First, some user-specified portion of the buffer is filled, and at that point the player starts to consume the buffer. As long as the download proceeds with enough aggregate bandwidth and as long as the variability in individual downloads is low enough, the player may play the file with no problems. However, if the bandwidth wanes, or a portion of the download exhibits very bad performance, then the player typically pauses until the slow data arrives. If several pauses occur in rapid succession, the player will appear to stutter. No frames are dropped since the download uses a reliable TCP connection.

Thus, our experiments will reflect streaming considerations, focusing on the delay that is induced by variability in the performance of individual block downloads.

3.2.1. *Basic downloading algorithm*

Our basic download algorithm is a straightforward, adaptive algorithm. Suppose that exNode E_x is being downloaded. The file is broken into blocks of size b, and nx threads are created such that each IBP server will be serviced by n TCP streams (again, in this paper, n will be either one or two). Each thread selects a different block to download, and all threads start downloading. When a thread is finished with its block, it selects a new block that is not being downloaded by any other thread, and works on that block. If a download fails, then the failed block becomes free so that another thread servicing a different IBP server may attempt to download it, thereby giving the download a degree of fault-tolerance.

This algorithm is adaptive, because IBP servers with high bandwidth to the client should download many more blocks than those with low bandwidth. Moreover, as long as there are many blocks to be downloaded, the algorithm may adapt to fluctuating network conditions. The selection of the blocksize is of concern. Blocks that are too small may suffer too much from the effects of latency and overhead in their downloads, while blocks that are too large may hinder the degree of adaptive load-balancing that the algorithm may achieve. It will be a matter of experimental exploration to determine an optimal block size, and to see if that block size applies over a range of servers and clients.

3.2.2. *Aggressive redundancy*

An obvious problem with the basic algorithm is that one or more slow downloads can significantly hurt performance, especially when the download is streaming to an application. A straightforward, yet rather heavy-handed algorithm to solve this problem is to download each block simultaneously from more than one depot. We introduce a *redundancy factor* R, which is the number of threads that will simultaneously download each block. We anticipate that this will lower the variability of download times for each block downloaded, at the expense of the overall download bandwidth (since approximately $\frac{R-1}{R}$ of the downloads will not contribute to the useful work of the download).

3.2.3. *Less aggressive redundancy*

Perhaps a better idea than replicating downloads is instead to monitor the progress of each thread's download, and to retry a download when it is deemed to be progressing too slowly. The challenge is the define exactly what "too slowly" means. The following is a simple algorithm called **Progress-Driven Redundancy**:

With progress-driven redundancy, a *progress number* P is selected along with a redundancy factor R, prior to the download. The blocks are numbered consecutively, starting at zero, and each block is assigned an initial *download number* of zero. Whenever a thread attempts to download a block, it increments the block's download number. When a thread finishes a block download, its next task is to select a new block to download. If there is a block B with a download number less than R that has not completed its download, and there are more than P blocks with numbers greater than B whose downloads have completed, then the thread selects block B to download. If there are no such blocks, then the thread works on the next block whose download number is zero. When all blocks have download numbers greater than zero, then when a thread has completed a download, it searches for the smallest block B with a download number less than R, and works on it.

Note that all three algorithms are really variants of progress-driven redundancy. The basic algorithm may be viewed as progress-driven redundancy with $P = 0$ and $R = 1$. Similarly, aggressive redundancy may be viewed as progress-driven redundancy with $P = 0$.

As an example to further illuminate the algorithms, consider the downloading scenarios depicted in Figure 3. In each there are four copies of the file residing in four IBP depots, and four threads are performing the download. In Scenario 1, thread 4 has to decide which block to download next. In the basic algorithm, this is simply the next block upon which no download has been started – block 9. With progress-driven redundancy, the values of R and P dictate which block is selected to download by thread 4. If P is less than or equal to three, and R is greater than one, then thread 4 downloads block 2 along with thread 1. This is because there are four blocks between blocks 2 and 9 whose downloads have completed. If P is

Example

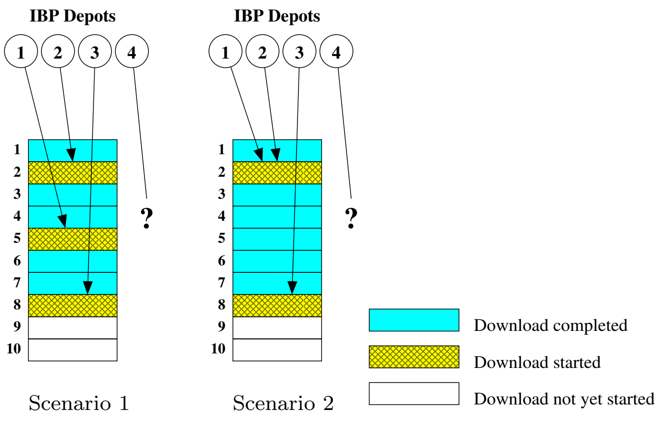

Fig. 3. Example downloading scenarios.

greater than three, then thread 4 commences with block 9. Note, scenario 1 cannot occur with aggressive redundancy.

In Scenario 2, there is already some redundancy to the download, and thus the scenario cannot occur with the basic algorithm. With aggressive redundancy ($R = 2$), thread 4 must redundantly download block 8. With progress-driven redundancy, again the selection of thread 4's block depends on R and P. Assuming $P > 0$, if $R = 2$, then thread 4 starts downloading block 9. If $R > 2$, then the only way the scenario can occur is for $P \leq 4$. As such, thread 4 must also redundantly download block 2.

4. Experimental Results

To test the various download algorithms, we replicated a 50 MB data file on the following IBP servers:

#	Server	#	Server
I_1	Texas A&M (College Station, TX)	I_4	Singapore
I_2	University of California at Santa Barbara	I_5	University of Tennessee (Knoxville, TN)
I_3	Harvard University (Cambridge, MA)	I_6	Surfnet (Amsterdam, NL)

We combined the replicas into six exNodes, $E_1 \ldots E_6$, where each E_x has exactly x replicas, from servers $1 \ldots x$. We then attempted to download the exNodes from two separate clients: one at the University of Tennessee and one at the University of California at Santa Barbara. Each result below is the average of many runs executed at various times throughout the day.

4.1. *Download bandwidth*

The download bandwidth of the three algorithms for various block sizes is plotted in Figures 4 and 5. Note, the legend in Figure 4 applies to that and all following figures. The performance is clearly dependent on the location of the clients, each of which has one nearby server among the collection: Server I_5 is on the same local area network as the Tennessee client, and Server I_2 is on the same local area network as the Santa Barbara client. Accordingly, the graphs in Figure 4 show drastic performance improvements at exNode E_5 (Tennessee), and the graphs in Figure 5 show drastic improvements at exNode E_2 (Santa Barbara).

As shown by the leftmost graphs in both figures, the basic algorithm's performance peaks when the replica with the nearby depot is reached. Unfortunately, however, as more replicas are added, its performance drops drastically. The reason is is that the slow download of a single block can penalize the overall performance drastically. For example, in one randomly selected test, when the client at UT downloads exNode E_6 with one thread per depot and a block size of 2MB, Server I_5 (at Tennessee) is responsible for 17 of the 25 downloads. After it completes its last download, the client must wait 1.7, 2.5, 4.0, 7.3 and 103.7 seconds for the downloads from servers I_1, I_2, I_3, I_4 and I_6 respectively.

The aggressive redundancy results (middle graphs) show roughly the same peak performance as the basic algorithm, but the performance drop-off occurs with less frequency. For example, with a block size of 128 KB, the performance from the Santa Barbara client stays constant at roughly 50 Mbps for exNodes E_2 through

—■—	1 Thread/depot. Blocksize: 128 KB	—■—	2 Threads/depot. Blocksize: 128 KB
– × –	1 Thread/depot. Blocksize: 256 KB	– × –	2 Threads/depot. Blocksize: 256 KB
- - + - -	1 Thread/depot. Blocksize: 512 KB	- - + - -	2 Threads/depot. Blocksize: 512 KB
—●—	1 Thread/depot. Blocksize: 1 MB	—●—	2 Threads/depot. Blocksize: 1 MB
– ▲ –	1 Thread/depot. Blocksize: 2 MB	– ▲ –	2 Threads/depot. Blocksize: 2 MB

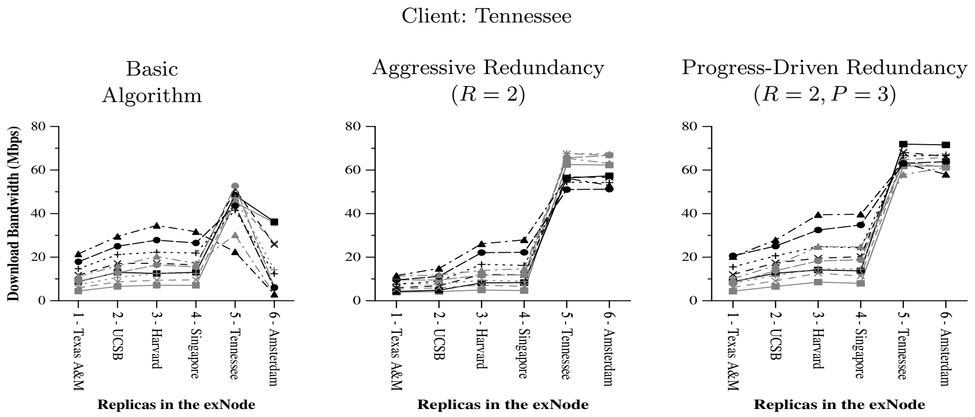

Fig. 4. Download performance to Tennessee of the three algorithms on a replicated 50 MB file.

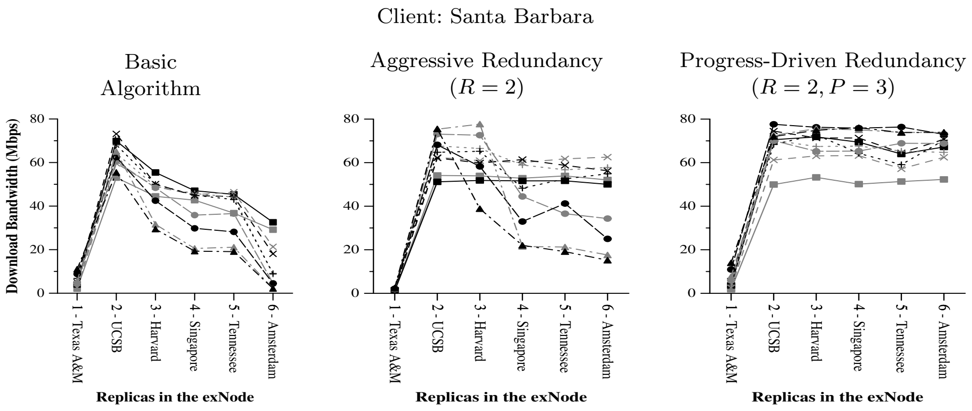

Fig. 5. Download performance to Santa Barbara of the three algorithms on a replicated 50 MB file.

E_6. As the block sizes increase, however, the probability that two slow depots will be responsible for a large block increases. For example, when downloading E_6 from Santa Barbara with 1 thread per depot and a block size of 2 MB, the overall performance of the download is penalized when the Harvard and Singapore depots are responsible for block #1, which arrives from Singapore over 7 seconds later than all the other blocks (at that point, the Harvard download had not even completed).

The progress-driven results (rightmost graphs) have the most desirable properties. The absolute performance numbers in all cases are slightly better than the other two algorithms, and more importantly, the performance does not drop as more replicas are added. The reason that this is important is that to optimize the other two algorithms, some notion of proximity will be required, be it online monitoring and forecasting, or use of an external monitoring entity such as the Network Weather Service [WSH99]. With progress-driven redundancy, the self-adapting nature of the algorithm allows the client to simply try downloading from *all* replicas, in order to gain the benefits of finding the one that closest without being penalized by downloading from too many servers.

The effect of the block size on the download is variable. When the performance of all servers is roughly equal (as in E_1 through E_4 from Tennessee), the improvements from downloading large blocks is the significant factor in improving performance. However, when the variability of download times is heightened, as occurs when a high-performance or low-performance server is added (such as E_5 and E_6 respectively from Tennessee), then in the first two algorithms, large block sizes become a problem, and the smaller block sizes perform better. With progress-driven redundancy, when there is a high-performance server, it is unlikely that multiple low-performance servers will be responsible for a single block's download. For that reason, the large block sizes perform well even when there is a great disparity in server download speeds.

Although we did not test block sizes greater than 2 MB, we anticipate that they will perform better with progress-driven redundancy.

4.1.1. *Changing R and P*

To assess the effects of changing R on the aggressive download algorithm, we measured $R = 2$ and 3 in the Santa Barbara downloads. The results are in Figure 6. As would be expected, this reduces the drop-off as more replicas are downloaded, due to the fact that the variability in individual block downloads is decreased. An unforeseen effect is that the overall peak bandwidth appears to stay roughly the same. The reason is that due to the small number of TCP streams, the available bandwidth to the client has not been saturated. Were we to employ more parallel TCP streams, we anticipate that the overall bandwidth would lessen.

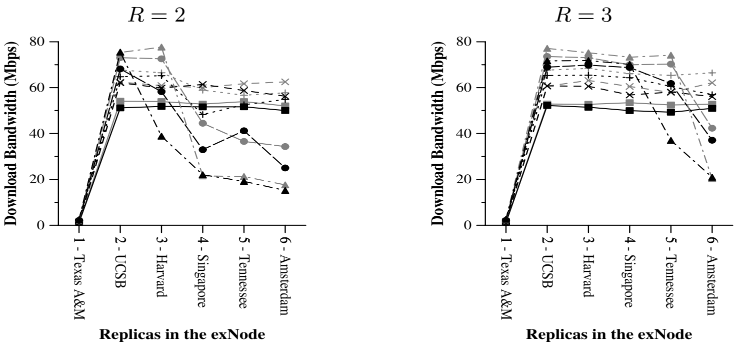

Fig. 6. The effect of altering R on the Santa Barbara downloads.

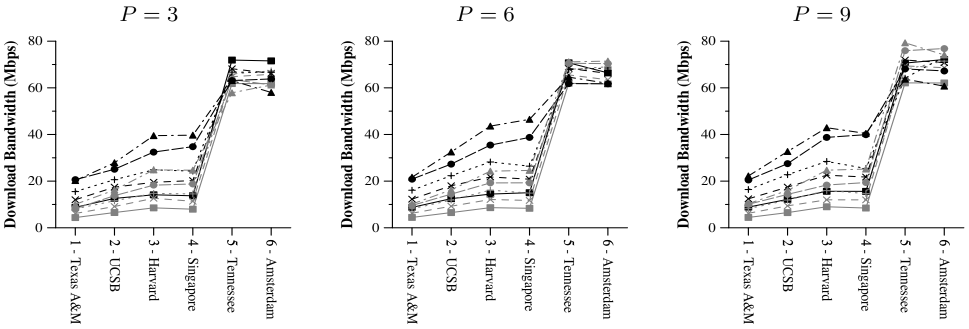

Fig. 7. The effect of altering P on the Tennessee downloads.

Figure 7 displays the effect of setting P to 3, 6 and 9 in the Tennessee downloads. The change is minimal as P is increased, except the download bandwidth goes up slightly as P increases. For example, the maximum bandwidth with $P = 3$ is 71.9 Mbps, while for $P = 9$, it increases 10% to 79.3 Mbps.

4.2. *Streaming performance*

We also tested the streaming performance of these downloads. While average bandwidth is indeed a valid measure of performance, there are times, for example showing multimedia, when the download needs to maintain a certain base performance. For each test displayed above, we also calculated the delay that the download would induce were it to be consumed at a specified rate. These calculations are in Figures 8, 9 and 10. In these graphs, $R = 2$ and $P = 3$.

For each download, we assumed that the application allowed five seconds for buffering before consuming the bytes. Then the bytes have to be present at the client at the specified rate. For example, at 1.3 Mbps and a block size of 128 KB, the second block must be completely downloaded at $\frac{128*8}{1024*1.3} = 0.77$ seconds after the pre-buffering phase. Otherwise a delay will be induced.

Figures 8, 9 and 10 plot the average delay per run calculated in this manner, plus 0.1 second (so that a log scale may be used to plot the results). Thus a value of 0.1 in the figures corresponds to the case where there is no delay.

Figure 8 displays the performance when the file is to be consumed at 1.3 Mbps, the speed of an audio player playing uncompressed audio. As would be expected

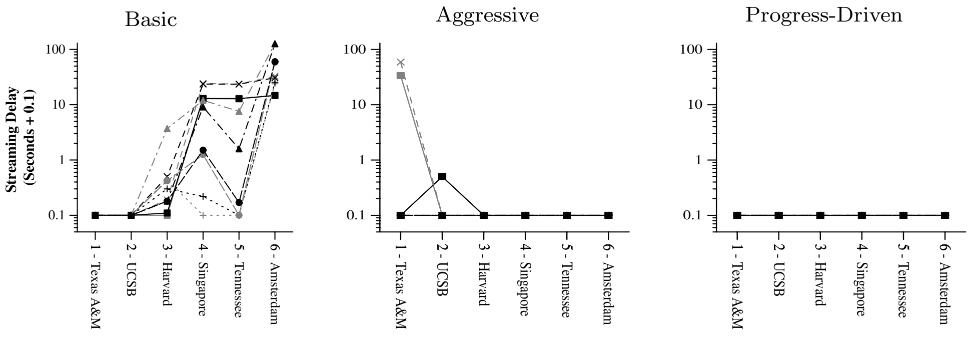

Fig. 8. Streaming download performance to Tennessee at 1.3 Mpbs (5 second pre-buffering).

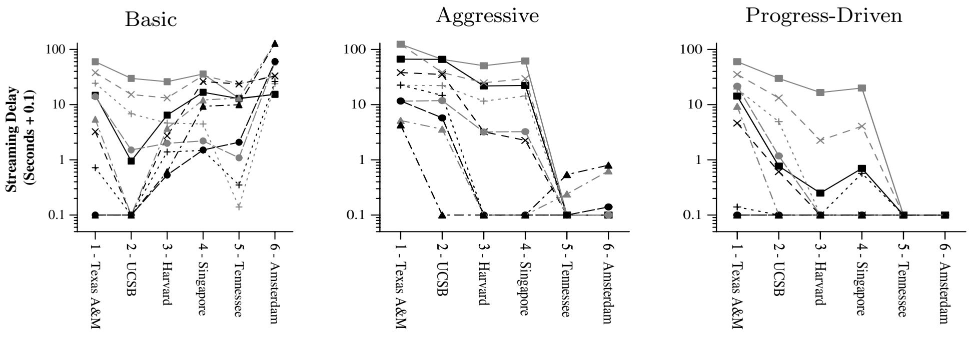

Fig. 9. Streaming download performance to Tennessee at 15 Mpbs (5 second pre-buffering).

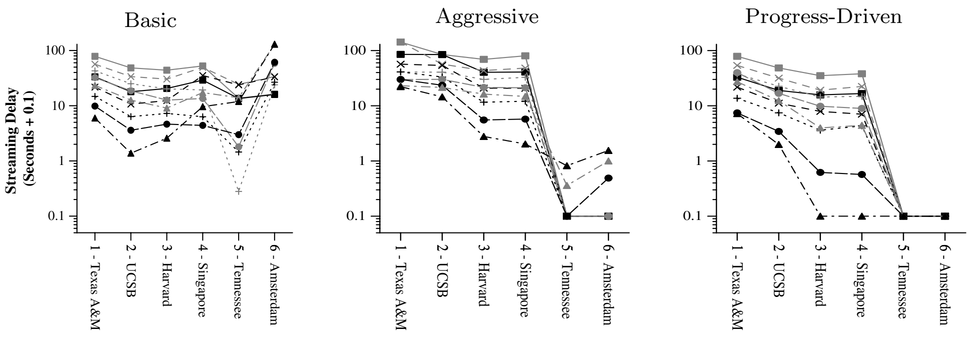

Fig. 10. Streaming download performance to Tennessee at 50 Mpbs (5 second pre-buffering).

from the previous graphs, the basic algorithm induces large delays when playing replicated files. This is because very slow downloads of blocks may occur with no redundancy. This effect is more pronounced with larger block sizes. aggressive redundancy is able to counteract this effect in almost all cases, and progress-driven redundancy keeps up with the required rate of the download in all cases.

Figure 9 shows a rate of 15 Mbps, the speed of high-quality compressed video. Now the basic algorithm can only maintain the rate with no delay in one or two cases, while aggressive redundancy keeps up with either large blocks and little replication, or small blocks and a lot of replication. Progress-driven redundancy (with a 2 MB block size and two threads per depot) can deliver the required performance with no delay in all levels of replication.

Finally, Figure 10 shows a rate of 50 Mbps, which is the uncompressed speed of some video cameras. At this rate, the basic algorithm is unusable, inducing very high delays. aggressive redundancy is only successful when the Tennessee IBP server is part of the download. Progress-driven redundancy, on the other hand, can sustain 50 Mbps with 2 MB block sizes, two threads per depot, and three *non-local* replicas. This is significant, because the overall bandwidth of the download (Figure 4) is only 39.5 Mbps. Given the pre-buffering of roughly half the file, the slower download proceeds at an even enough pace that 50 Mbps is sustained.

5. Conclusions

We have detailed an architecture that allows data to be stored in time-limited storage depots in the wide area. One novel feature of this architecture is the ability to replicate data and download it to multiple clients in a variety of ways. This paper explores three algorithms for downloading replicated files, and tests their performance in a wide-area (global, in fact) setting.

As displayed by the results, an adaptive algorithm that uses a simple metric to retry slow downloads exhibited excellent performance and desirable performance characteristics downloading to clients in Tennessee and in Santa Barbara. These characteristics include:

- Increasing download bandwidth as more replicas are added to the data, regardless of the speed of the replicas' servers.
- Good sustained download rates for streaming, which allow media players to stream content in real time from wide-area servers.

The progress-driven algorithm is currently implemented in the LoRS tools, which are exported as open-source tools to the community, not only for their use "out of the box," but as vehicles for research on issues such as those addressed by this paper (see http://loci.cs.utk.edu for details on the tools).

We will continue to explore algorithms and methodologies for managing data with high performance on the wide area. We anticipate that the performance of downloads can benefit from two further principles. First, we have ignored the ability to perform monitoring and forecasting of communication data. One lesson learned from this paper is that when retrying a download, it is best to retry it from a fast server. This is because the fact that a download needs retrying means that its progress is slow. A fast server is required to "catch it up." By employing a forecasting methodology such as the Network Weather Service, we can likely improve performance more by characterizing servers as "fast" and "slow" and thereby only perform retries from fast servers.

Relatedly if the forecasts are reliable enough, we may also be able to schedule downloads from slow servers that will be likely not to need retrying. For example, if forecast measurements from the NWS predict that the servers will give an aggregate download bandwidth of X Mbps, and a slow depot is forecasted to download at $\frac{X}{10}$ Mbps, then if the blocksize is X Mb, we can have the slow depot start downloading 10 blocks ahead of where a fast depot would start downloading. That way, if forecasts are accurate, the block will be unlikely to need a retry.

A second way to improve performance may be to employ error-correcting coding, such as Reed-Solmon [Pla97, Riz97] or Tornado [LMS$^+$97] codes. Instead of downloading all data blocks and retrying slow blocks, the download tool may download n data blocks and m coding (e.g. parity) blocks. Then instead of retrying slow data blocks, the tool may instead calculate them from the already-downloaded data and coding blocks. The selection of n and m will be parameters for experimental study, building on the original Digital Fountain studies based on Tornado codes [BLM99].

Downloading is the first tool whose performance we have explored. Uploading and augmenting are the next tools deserving of attention. Adding to the complexity of these tools are the fact that data placement strategies, as well as actually moving the bytes, will be important.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grants ANI-0222945, ANI-9980203, EIA-9972889, EIA-9975015 and EIA-0204007, the Department of Energy under grant DE-FC02-01ER25465, and the University of Tennessee Center for Information Technology Research. The authors greatly acknowledge Stephen Soltesz and Yong Zheng who provided their time and intimate knowledge of the LoRS tools, Jeremy Millar who provided the same with the exNode tools, and Rich Wolski for numerous invaluable discussions. The authors also acknowledge Wolski, Graziano Obertelli, Norman Ramsey, Hunter Hagewood and the Planet Lab project [a] headed by Larry Peterson for depot access, and Terry Moore for his valuable insights.

References

[ABB$^+$02] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, and S. Tuecke. Data management and transfer in high performance computational grid environments. *Parallel Computing Journal,*, 28(5):749–771, May 2002.

[ADG$^+$00] M. Allman, S. Dawkins, D. Glover, J. Griner, D. Tran, T. Henderson, J. Heidemann, J. Touch, H. Druse, S. Ostermann, K. Scott, and J. Semke. Ongoing TCP research related to satellites. IETF RFC 2760 (`http://www.ietf.org/rfc/rfc2760.txt`), February 2000.

[BLM99] J. W. Byers, M. Luby, and M. Mitzenmacher. Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads. In *IEEE INFOCOM*, pages 275–283, New York, NY, March 1999.

[BMP01] M. Beck, T. Moore, and J. S. Plank. Exposed vs. encapsulated approaches to grid service architecture. In *2nd International Workshop on Grid Computing*, Denver, November 2001. `http://www.gridcomputing.org/grid2001`.

[BMP02] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable network storage. In *ACM SIGCOMM '02*, Pittsburgh, August 2002.

[CHM$^+$02] I. Clarke, T. W. Hong, S. G. Miller, O. Sandberg, and B. Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[Cot01] L. Cottrell. Achieving high performance throughput in production networks. Presentation at the ESnet Site Coordinating Committee Meeting, Argonne National Labs, `http://www.slac.stanford.edu/grp/scs/net/talk/thru-escc-oct01.html`, October 2001.

[CSWH00] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *International Workshop on Design Issues in Anonymity and Unobservability, LNCS 2002*, Berkeley, CA, July 2000. Springer.

[DKK$^+$01] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[DMT02] T. Dunigan, M. Mathis, and B. Tierney. A TCP tuning daemon. In *SC02: High Performance Networking and Computing Conference*, Baltimore, 2002.

[EHT00] L. Eggert, J. Heidemann, and J. Touch. Effects of Ensemble-TCP. *ACM Computer Communiation Review*, 30(1):15–19, January 2000.

[Fer02] G. Ferrache. bbftp: On-line documentation for release 2.2.1. `http://doc.in2p3.fr/bbftp/doc.2.2.1.html`, Documentation from the IN2P3 Computing Center, Lyon, France, August 2002.

[Flo00] S. Floyd. Congestion control principles. IETF RFC 2914 (`http://www.ietf.org/rfc/rfc2914.txt`), September 2000.

[a]http://www.planet-lab.org

[HMS99] J. H. Hartman, I. Murdock, and T. Spalink. The Swarm scalable storage system. In *19th International Conference on Distributed Computing Systems (ICDCS)*, June 1999.

[HO93] J. H. Hartman and J. K. Ousterhout. The zebra striped network file system. *Operating Systems Review – 14th ACM Symposium on Operating System Principles*, 27(5):29–43, December 1993.

[LMS+97] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann. Practical loss-resilient codes. In *29th Annual ACM Symposium on Theory of Computing,*, pages 150–159, 1997.

[MMGC02] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[PBB+01] J. S. Plank, A. Bassi, M. Beck, T. Moore, D. M. Swany, and R. Wolski. Managing data storage in the network. *IEEE Internet Computing*, 5(5):50–58, September/October 2001.

[Pla97] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[Riz97] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Computer Communication Review*, 27(2):24–36, 1997.

[RSC98] D. P. Reed, J. H. Saltzer, and D. D. Clark. Comment on active networking and end-to-end arguments. *IEEE Network*, 12(3):69–71, 1998.

[RWE+01] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5):40–49, 2001.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems,*, 2(4):277–288, November 1984.

[Tou95] J. Touch. Protocol parallelization. In G. Neufeld and M. Ito, editors, *Protocols for High Speed Networks IV*, pages 349–360. Chapman and Hall, London, 1995.

[WK02] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *First International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[WSH99] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5-6):757–768, 1999.