Downloading Replicated, Wide-Area Files – A Framework and Empirical Evaluation

Rebecca L. Collins James S. Plank

Department of Computer Science
University of Tennessee
Knoxville, TN 37996
[rcollins,plank]@cs.utk.edu

University of Tennessee Technical Report UT-CS-04-527
June 2004

Abstract

The challenge of efficiently retrieving files that are broken into segments and replicated across the wide-area is of prime importance to wide-area, peer-to-peer, and Grid file systems. Two differing algorithms addressing this challenge have been proposed and evaluated. While both have been successful in differing performance scenarios, there has been no unifying work that can view both algorithms under a single framework. In this paper, we define such a framework, where download algorithms are defined in terms of four dimensions: the number of simultaneous downloads, the degree of work replication, the failover strategy, and the server selection algorithm. We then explore the impact of varying parameters along each of these dimensions.

1 Introduction

In wide-area, peer-to-peer and Grid file systems [6–9, 11, 13, 16–18], the storage servers that hold data for users are widely distributed. To tolerate failures and to take advantage of proximity to a variety of clients, files on these systems are typically broken into blocks, which are then replicated across the wide area. As such, clients are faced with an extremely complex problem when they desire to access a file. Specifically:

> **Given a file that is partitioned into blocks that are replicated throughout a wide-area file system, how can a client retrieve the file with the best performance?**

This problem was named the "Plank-Beck" problem by Allen and Wolski [1], who denoted it as one of the two representative data movement problems for computational grids. In 2003, two major studies of this problem were published [1, 12], and each presented a different algorithm:

- A greedy algorithm where a client simultaneously downloads blocks of the file from random servers, and uses the progress of the download to specify when a block's download should be retried. This is termed *Progress-Driven Redundancy* [12].

- An algorithm where the client serially downloads blocks from the closest location and uses adaptive timeouts to determine when to retry a download. [1] In this paper, we call this the *Bandwidth-Prediction Strategy*.

In a wide-area experiment, Allen and Wolski showed that the two algorithms performed similarly in their best cases [1], but their performance could differ significantly. Beyond that conclusion, neither their work, nor the work in [12] lends much insight into *why* the algorithms perform the way they do, how they relate to one another in a more fundamental manner, and how one can draw general conclusions about them.

In this paper, we attempt to unify this work, providing a framework under which both algorithms may be presented and compared. We then explore the following four facets of the framework and how their modification and interoperation impact performance.

1. The number of simultaneous downloads.
2. The degree of work replication.
3. The failover strategy.
4. The selection of server scheduling algorithm.

We conclude that the two most important dimensions of downloading algorithms are the number of simultaneous downloads, and the server selection algorithm. The others do impact performance, but the extent of their impact depends on the number of downloads and the server selection algorithm.

2 Framework

In this section, a framework is built under which Progress-Driven Redundancy and the Bandwidth-Prediction Strategy can both reside. The object of this exercise is not to prove ultimately that one approach is better than the other, but instead to observe the ways in which the two algorithms are similar and different, and to explore the successful aspects of each algorithm.

Given a file that is partitioned into blocks that are replicated throughout a file system, the challenge of retrieving it is composed of four basic dimensions:

- **The number of simultaneous downloads**: How many blocks should be retrieved in parallel? The trade-off in this decision is as follows: too few simultaneous downloads may result in the incoming bandwidth not matching that of the client, and in the latency of downloads having too great an impact; while too many simultaneous downloads may result in congestion, either in the network or at the client. We quantify the number of simultaneous downloads by the variable T, as simultaneous downloads are usually implemented with multiple threads.

- **The degree of work replication**: Overall, what percentage of the work should be redundant? We assume that blocks are retrieved in their entirety, or not at all. Thus, when multiple retrievals of the same block are begun, any data collected in addition to one complete copy of the block from one source is discarded. In our study, work replication is parameterized by the variable R, which is the maximum number of simultaneous downloads allowed for any one block.

- **The failover strategy**: When do we decide that a block must be retried? Aside from a socket error, timeout expiration is the simplest way to determine that a new attempt to retrieve a block must be initiated. However, if timeouts are the only means of detecting failure, then they must be accurate if failures are to be handled efficiently. While adaptive timeouts have been shown to perform as well as optimally chosen static timeouts [2, 15], their implementation is more complicated than the implementation of static timeouts.

An alternative to failure identification via timeouts is the approach used in Progress-Driven Redundancy, where the success of a given retrieval attempt is evaluated in comparison to the progress of the rest of the file. When the download of a block is deemed to be progressing too slowly, additional attempts are simultaneously made to retrieve the block. The first attempt need not be terminated when new attempts begin, and in this way, all of the work of the first attempt is not lost if it finishes shortly after the new attempts begin. We quantify the notion of download progress with the parameter P, which specifies how much progress needs to be made with the file after a block's first download begins before that block requires replication.

- **The selection of server scheduling algorithm**: Which replica of a block should be retrieved? When blocks of a file are distributed, especially over the wide area, the servers where different copies of the same block reside have different properties. Each server possesses two traits by which it may be characterized, speed and load. A server's speed is approximately bandwidth, or more specifically, the time the server takes to deliver one MB. A server's load is the number of threads currently connected to the server from our client application. We investigate seven server scheduling algorithms, each of which is described in section 3.3

3 Algorithms

Now that a framework is established for the comparison of wide-area download algorithms, the Progress-Driven Redundancy and Bandwidth-Prediction Strategy algorithms are presented in sections 3.1 and 3.2, respectively. Following that, several server scheduling algorithms are outlined.

In order to understand the details of the following algorithms, suppose the desired file is subdivided into blocks, and the blocks are indexed by their offset in the file. Suppose also that each of the file's blocks is replicated C times such that no two copies of the same block reside in the same place. The algorithms attempt to acquire blocks by the order of their indices.

3.1 Progress-Driven Redundancy

As originally defined [12], with Progress-Driven Redundancy, a progress number P and a redundancy number R are selected at startup. Strictly speaking, R cannot be greater than C. The number of threads, which determines the maximum number of simultaneous downloads, is also chosen.

Each block is given a download number initialized to zero. The download number of a block is incremented whenever a thread attempts to retrieve one of the block's copies. When a thread is ready to select a new block to download, it first checks to see if a block exists that has a download number less than R, such that more than P blocks with higher offsets in the file have already been retrieved. If such blocks exist, then the thread chooses the block with the lowest offset that meets these requirements. If not, then the thread selects the block with the lowest offset whose download number is zero.

Since blocks near the end of the file can never meet the progress requirement, once a thread finds that no more blocks can be selected according to download number, P, and R; it selects the block with the lowest offset whose download number is less than R. We call this a "swarm finish".

Relating back to the previously outlined framework, the number of threads determines the number of simultaneous downloads; the redundancy number determines the degree of work replication; and the progress number determines the failover strategy. When initially presented, the server scheduling assumed that the file was fully replicated at every site, and threads were assigned to individual servers [12]. This was augmented in [1] so that server selection was performed randomly. In this work, we explore a variety of server selection algorithms.

3.2 Bandwidth-Prediction Strategy

To proceed with the Bandwidth-Prediction Strategy, we simply need a means to determine which server is the closest, or the fastest. The original authors assume that the Network Weather Service [20] is implemented at each site, and employ that to determine server speed. Then, the blocks are retrieved in order, one at a time, from the fastest server. Timeouts, whose values are determined by the NWS, are used as the failover strategy. Thus, relating back to the previously outlined framework, T is one, R is one, failover is determined by timeouts, and server selection is done with an external bandwidth predictor.

3.3 Server Scheduling

The original work on Progress-Driven Redundancy did not address server scheduling. The work of Allen and Wolski employed the Network Weather Service for the Bandwidth Prediction Algorithm, and random server selection for Progress-Driven Redundancy. In this paper, we explore a wider variety of server selection algorithms. We assume either that there is a bandwidth monitoring entity such as the Network Weather Service, or that the client has access to previous performance from the various servers, and can augment that with performance metrics gleaned from the download itself. With this assumption, we outline seven server selection algorithms:

1. The **random** strategy chooses a random server.

2. The **forecast** algorithm uses monitoring and forecasting to select the server that should have the best performance.

3. The **lightest-load** algorithm assigns a *current load* l to each server. This is equal to the number of threads currently downloading from the server, and is monitored by the client. With **lightest-load**, the server with smallest value of l is selected. In the case of ties, server speed is employed, and the fastest server is selected.

4. The **strict-load** algorithm enforces tcp-friendliness by disallowing multiple simultaneous connections to the same server. It works just like **lightest-load**, except it always chooses servers where $l = 0$. If there are no unloaded servers, then no servers are selected.

5. The remaining three algorithms use a combination of load and speed to rank the servers. Specifically, they select the server with smallest values of $time * (\alpha * l + 1)$, where $time$ is the predicted time to download one block of the file when there is no contention. For $\alpha = 0$, we call this algorithm **fastest**$_0$.

6. **fastest**$_1$ minimizes $time * (l + 1)$.

7. **fastest**$_{1/2}$ minimizes $time * (l/2 + 1)$.

Dimension	Range of Parameters
Simultaneous Downloads	$T \in [1, 2, 3, 5, 10, 15, 20, 25, 30]$
Work Replication	$R \in [1, 2, 3, 4]$
Failover Strategy	$P \in [1, 2, 3, 5, 10, 15, 20, 25, 30]$, static timeouts
Server Selection	The seven selection strategies

Table 1: **Ranges of parameters explored.**

Region	Num. Servers	Num. Serv. Typically Up
University of Alabama (UAB)	7	6-7
University of California - Santa Barbara (UCSB)	6	4-5
Wisconsin (WISC)	4	2-3
United Kingdom (UK)	7	3-4

Table 2: **Regions used in regional distribution**

4 Experiment

During May and June 2004, we conducted a series of experiments in order to study the dynamics of the progress-driven redundancy algorithm. The goal of the experiments was to determine the impact of modifying parameters of the four dimensions when downloading a 100 MB (megabyte) file distributed on the wide area. Specifically, we tested all combinations of the ranges of parameters detailed in Table 1. Note that R cannot exceed T, and that if $R = 1$, then blocks are only retried upon socket failure (host unreachable or socket timeout). For speed determination and prediction, we employed a static list of observed speeds from each server. For the **forecast** algorithm, this list was used as the starting point, and subsequent block download speeds were fed into the Network Weather Service's forecasting software, to yield a prediction of the speed of the next download.

IBP [13] servers were used to store the blocks of the file. IBP is a software package that makes remote storage available as a sharable network resource. IBP servers allow clients to allocate space on specific servers (called "depots") and then manage the transfer of data to and from allocations. IBP servers use TCP sockets and can operate on a wide variety of architectures. A list of publicly available IBP servers and their current status can be found on the LoCI website: `http://loci.cs.utk.edu`. The client machine used for the experiments ran Linux RedHat version 9, had an Intel (R) Celeron (R) 2.2 GHz processor, and was located at the University of Tennessee in Knoxville. The downloads took place over the commodity Internet. The tests were executed in a random order so that trends due to local or unusual network activity were minimized, and each data point presented is the average of ten runs.

We tested two separate network files. Both are 100 MB files, broken into blocks of size 1 MB. Each block is replicated at four different servers. The two files differ in the nature of the replication. The first, which we call **regional**, has each block replicated in four network regions. This is typical of a piece of content that is being managed so that it is cached in strategically chosen regions. The regions for this file are detailed in Table 2. Note, there are multiple servers in each region, and since these are live servers in

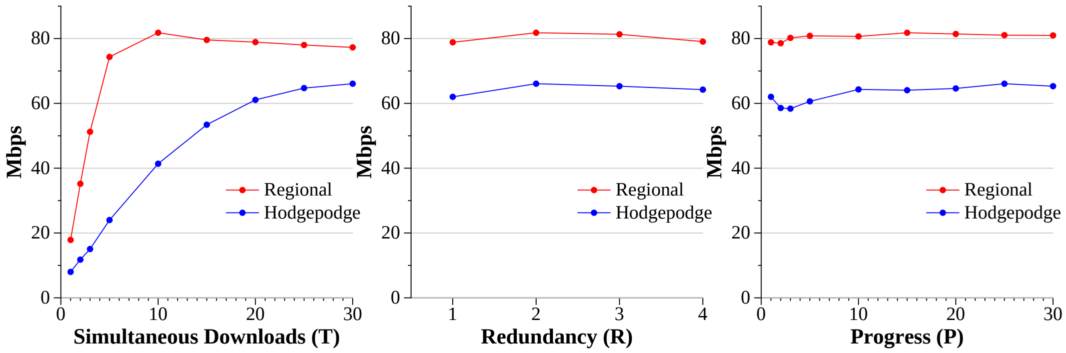


Figure 1: **The best performance of threads, redundancy and progress**

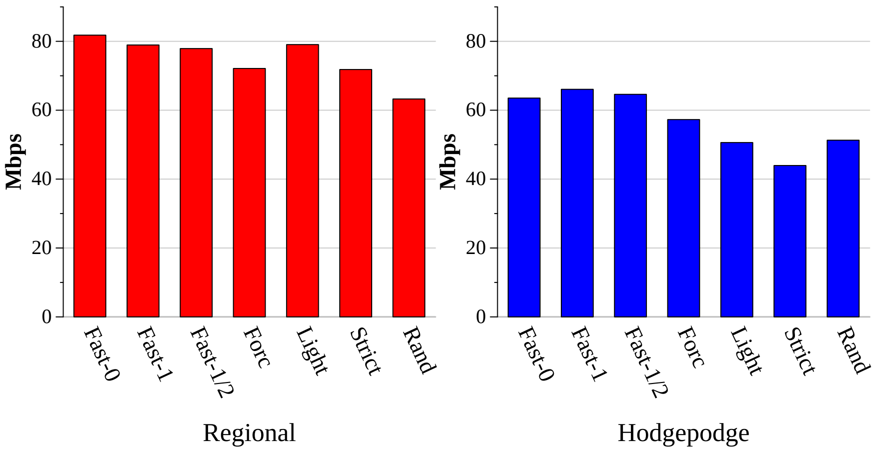


Figure 2: **The best performance of each server scheduling algorithm**

the wide-area, they have varying availability, also denoted in the table.

The second file is called **hodgepodge**, as its blocks are stored at servers randomly distributed throughout the globe. Specifically, fifty regionally distinct servers were chosen, and the blocks of the file were striped across all fifty servers. A list of the set of servers that the fifty were chosen from for the hodgepodge distribution is included in the Appendix along with a more precise description of the distribution. In both files, no two copies of the same block resided in the same region, and no blocks were stored at the University of Tennessee, where the client was located.

5 Results

We present the results first as broad trends for each of the four dimensions presented. We then explore more specific questions concerning the interaction between the parameters and some of the details of the downloads.

5.1 Broad Trends for Each Dimension

Figures 1 and 2 show the best performing downloads when parameters for each dimension are fixed. For example, in the leftmost graph of Figure 1, T ranges from one to thirty, and for each

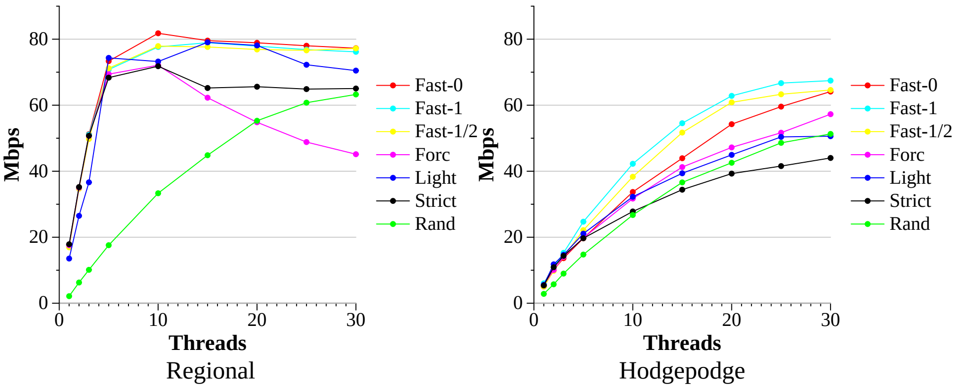


Figure 3: **Best performance of each server scheduling algorithm plotted over threads**

value of T, the combination of P, R and scheduling algorithm that yields the best average download performance is plotted.

Two obvious results are clear from the figures. First, the composition of the file affects both the performance of downloading and the optimal set of parameters. The regional file has an optimal download speed of 82 Mbps (Megabits per second), while the hodgepodge file achieves a lower optimal speed of 66 Mbps. Second, the number of simultaneous downloads has far more basic impact on the performance of the algorithm than the choice of R and P. However, it is not true that bigger values of T necessarily translate into better performance. Specifically, in the regional file, the optimal performance comes when $T = 10$, while in the hodgepodge, it occurs when $T = 30$. We surmise that the performance is best when the number of threads can utilize the capacity of the network. Beyond that, contention and thread context-switch overhead penalize the employment of more threads.

From figure 2, we conclude that the scheduling algorithms that incorporate some kind of speed prediction are the most successful. Observe the poor performance of the random algorithm in both types of file distributions. The **strict-load** algorithm also has low overall performance for both distributions. While in some applications it may be necessary to adhere to limitations on the number of connections made to the same server, such limitations clearly hinder performance for the following reasons: first, the client cannot take advantage of multiple network paths from the server, and second, in cases where a great disparity exists between the performance of servers, too few downloads are permitted from the faster servers.

The **forecast** algorithm performs relatively poorly as well. A likely explanation of this behavior is that its forecasts are too coarse-grained for this application and as a result, the algorithm cannot adapt quickly enough to the changing environment. A finer grained forecaster would probably have better performance, and it is possible that the coarse forecaster may have better performance given a bigger file and thus more history for each server as the download progresses.

While there does not appear to be an optimal scheduling algorithm *per se*, the three **fastest**$_\alpha$ algorithms as a whole outperform the others.

5.2 The Interaction of Server Selection and Threads

Figure 3 gives a more in-depth picture of the interaction of the scheduling algorithms and the number of threads. The best performance of each

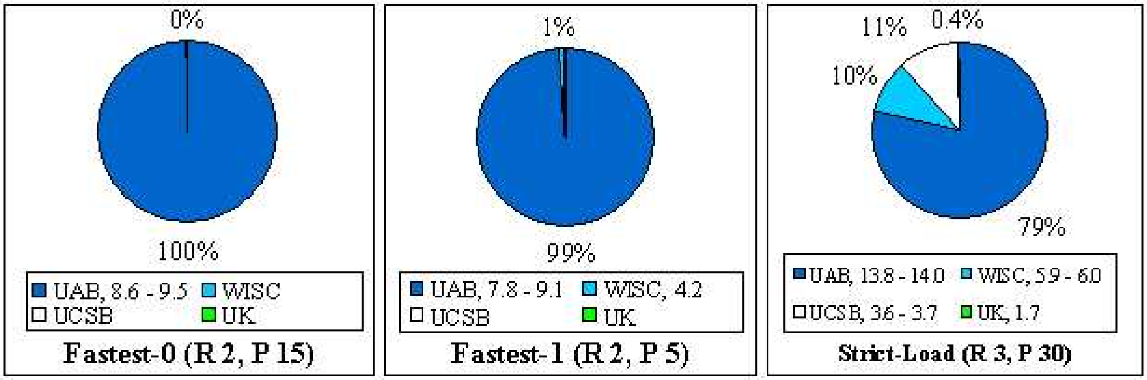


Figure 4: **Breakdown of where blocks came from in the best performances of the fastest$_0$, fastest$_1$, and strict-load algorithms with 10 threads (range of average download speeds in Mbps is listed in legend)**

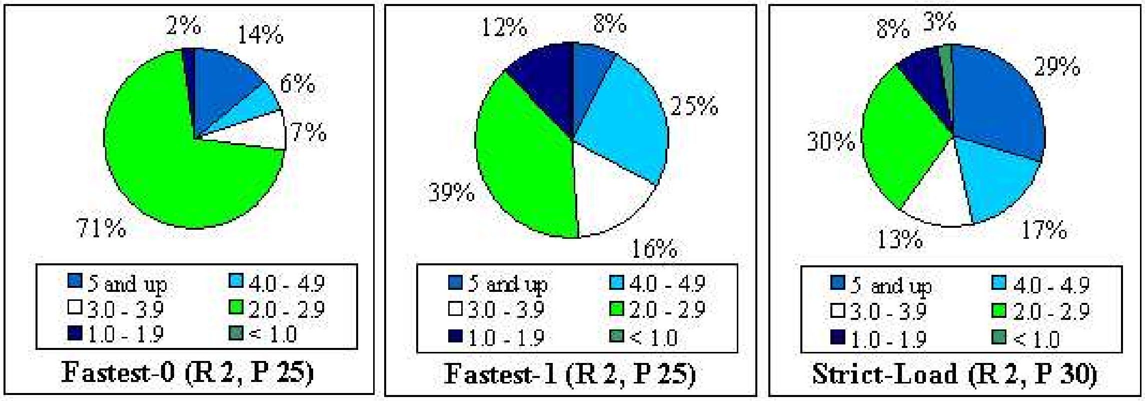
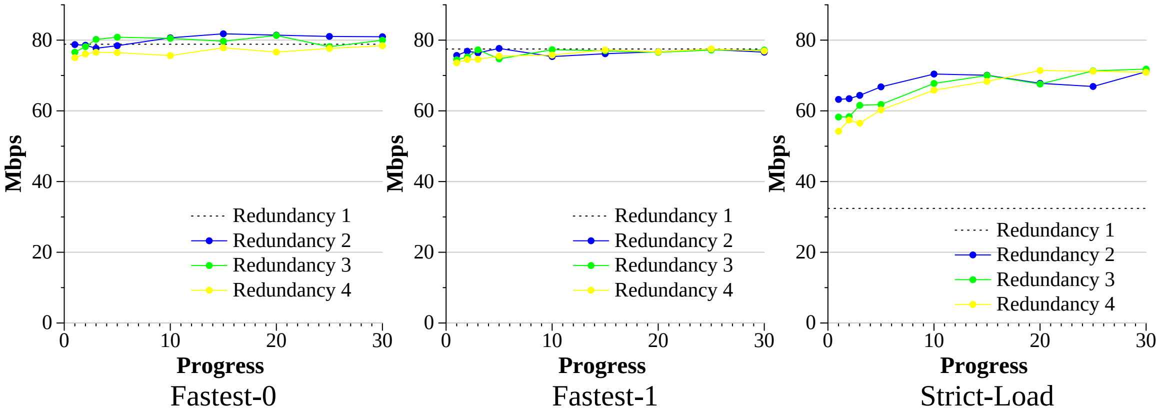


Figure 5: **Breakdown of where blocks came from in the best performances of the fastest$_0$, fastest$_1$, and strict-load algorithms with 30 threads (range of average download speeds in Mbps is listed in legend)**

algorithm given the number of threads is plotted. The overall trends in figure 2 still hold in most cases. However, in the regional distribution, the **forecast** algorithm experiences a marked degradation as the number of threads increase. As noted before, the **forecast** algorithm appears to adapt too sluggishly to the changing environment. As the number of threads increases, the degree to which each server's performance varies also increases, due to the fact that a wider range of concurrent connections can be made to each server.

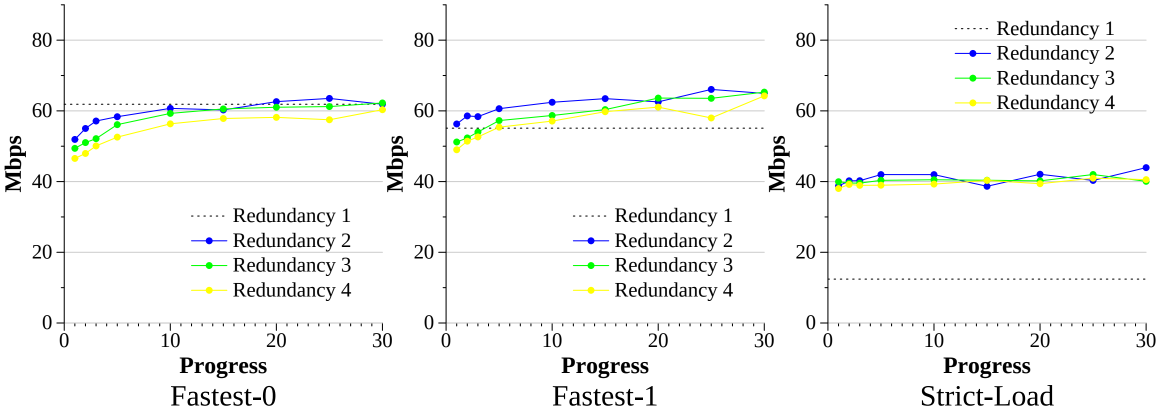
5.3 Where do the blocks come from?

Figures 4 and 5 display a breakdown of where blocks came from in some of the the best performing instances of the **fastest**$_0$, **fastest**$_1$ and **strict-load** algorithms. The instances of the regional distribution are broken down over the regions, while the instances of the hodgepodge distribution are broken down over ranges of average download speeds. From earlier figures, the **strict-load** algorithm performs poorly in comparison to the other algorithms. In both the regional and the hodgepodge distributions, we can see that the **strict-load** algorithm is forced



Fastest-0 Fastest-1 Strict-Load

Figure 6: **The relationship of progress and redundancy with 10 threads over the regional distribution**



Fastest-0 Fastest-1 Strict-Load

Figure 7: **The relationship of progress and redundancy with 30 threads over the hodgepodge distribution**

to retrieve larger percentages of its blocks from slower servers. The reader may notice that the average download speed from the UAB region is faster for the **strict-load** algorithm than it is for the **fastest**$_0$ and **fastest**$_1$ algorithms. This is because the **strict-load** algorithm avoids congestion of TCP streams. However, the fact that the performance of the other algorithms is faster shows the availability of more network capacity from these sites than can be exploited by a single TCP stream.

This behavior is also apparent in figure 5 where the blocks are split up according to download speed. Notice that the **fastest**$_0$ algorithm has a larger percentage of blocks in the $2.0 - 2.9$ Mbps range and a smaller percentage of blocks in the $4.0 - 4.9$ Mbps range than the **fastest**$_1$ algorithm even though the **fastest**$_0$ algorithm always chooses the faster server regardless of that server's load.

5.4 The Interaction of P and R

The interaction of progress with redundancy is shown in Figures 6 and 7. While better performance does tend to lean slightly to higher progress numbers in some cases, for the most part, as long as $R \geq 2$, the performance does not change significantly with progress. In both

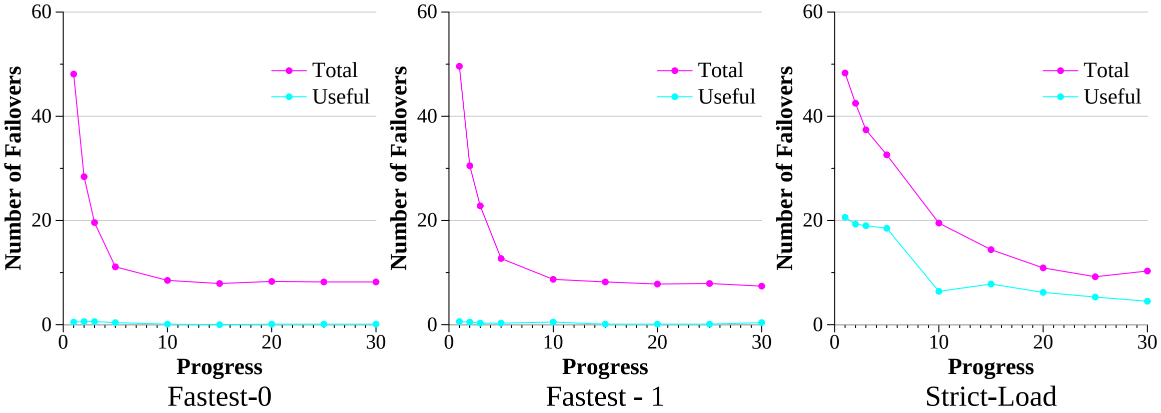


Figure 8: **Number of failovers with 10 threads, R=2, over the regional distribution**

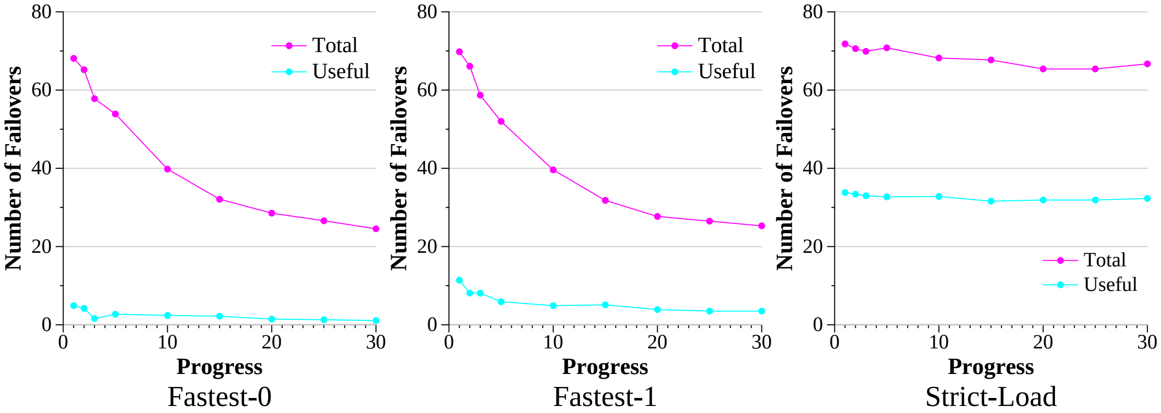


Figure 9: **Number of failovers with 30 threads, R=2, over the hodgepodge distribution**

distributions, the performance when $R = 1$ is very close to the performance when $R = 2, 3$ or 4, when the **fastest**$_0$ and **fastest**$_1$ algorithms are used. However, in the **strict-load** algorithm, where optimal choices are not always permitted, the ability to add redundant work to a block proves to be advantageous.

5.5 When is Aggressive Failover useful?

Given that it is sometimes advantageous to make retries, how often is a failover necessary? Figures 8 and 9 show the number of failovers versus the progress number when $R = 2$ and there are 10 threads over the regional distribution and 30 threads over the hodgepodge distribution. The total number of failovers is shown along with the total number of useful failovers, that is, the number of times a retry was attempted and number of times the retry completed before the original attempt. Clearly, small progress numbers lead to excessive numbers of failovers, while larger progress numbers result in a higher percentage of useful failovers. It is also clear that a higher percentage of retries are useful to the **strict-load** algorithm, which is constrained to choose slow servers at times because of the restriction of permitting only single TCP streams.

6 Conclusion

Given a file that is distributed across a system, how can we best leverage the properties of the system to retrieve the file as quickly as possible? With regard for the two previously proposed approaches to this problem, Progress-Driven Redundancy and Bandwidth-Prediction, we have explored the impact and interrelationships of the following download parameters: the number of simultaneous downloads, the degree of redundancy, the failover strategy, and the server selection algorithm.

As an obvious result, we found that performance tends to improve as the number of simultaneous downloads increases to a point, and that the distribution of the file across the system impacts the way the download parameters perform and interact.

With respect to the Bandwidth-Prediction approach, some form of bandwidth prediction greatly improves performance, and with respect to Progress-Driven Redundancy, some form of redundancy is very useful when poorly-performing servers are selected for downloads. Concerning performance prediction, in our tests, exploiting knowledge from the client (concerning the load from each server) is more beneficial to performance than having an external prediction engine try to react to the observed conditions. However, as stated above, this may be an artifact of the monitoring granularity, and more fine-grained monitoring may lead to better performance of predictive algorithms.

We anticipate that the results of this work will be implemented in the **Lo**gistical **R**untime **S**ystem [4], which already implements a variant of Progress-Driven Redundancy as the major downloading algorithm for its file system built upon faulty and time-limited storage servers, and has seen extensive use as a Video delivery service [3] and medical visualization back-end [10].

This work does have limitations. First, we did not employ an external monitoring agent such as the Network Weather Service. This is because we did not have access to such as service on the bulk of the machines in our testbed. With the availability of such a service, we anticipate an improvement in the **fastest**$_\alpha$ algorithms; however, we also anticipate that these algorithms should still incorporate knowledge of server load.

Second, we did not test the performance from multiple clients. However, we anticipate that the results from the one client are indicative of performance from generic clients, when the clients are not co-located with the data.

Third, we did not assess the impact of timeout-based strategies, which have been shown to be important in some situations [1, 15]. Instead, we have focused on algorithm progress and socket timeout as the failover mechanism. We intend to explore the impact of timeouts as a complementary failover mechanism in the future.

Finally, *erasure codes* have arisen as a viable alternative to replication for both caching and fault-tolerance in wide-area file systems [5, 14, 19, 21]. In future work, we intend to see how these downloading algorithms apply to file systems based on erasure codes, what additional considerations apply, and what the performance impact is.

7 Acknowledgements

This material is based upon work supported by the National Science Foundation under grants ACI-0204007, ANI-0222945, and EIA-9972889, and the Department of Energy under grant DE-FC02-01ER25465. The authors thank Micah Beck and Scott Atchley for helpful discussions, and Larry Peterson for PlanetLab access.

References

[1] M. S. Allen and R. Wolski. The Livny and Plank-Beck Problems: Studies in data movement on the computational grid. In *SC2003*, Phoenix, November 2003.

[2] M. S. Allen, R. Wolski, and J. S. Plank. Adaptive timeout discovery using the network weather service. In *11th International Symposium on High Performance Distributed Computing (HPDC-11),*, Edinburgh, Scotland, July 2002.

[3] S. Atchley, S. Soltesz, J. S. Plank, and M. Beck. Video IBPster. *Future Generation Computer Systems*, 19:861–870, 2003.

[4] 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.

[5] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *ACM SIGCOMM '98*, pages 56–67, Vancouver, August 1998.

[6] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, 2001.

[7] 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.

[8] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, Berkely, CA, June 2003.

[9] 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.

[10] J. Ding, J. Huang, M. Beck, S. Liu, T. Moore, and S. Soltesz. Remote visualization by browsing image based databases with logistical networking. In *SC2003 Conference*, Phoenix, AZ,, November 2003.

[11] G. Kan. Gnutella. In A. Oram, editor, *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, pages 94–122. O'Reilly, Sebastopol, CA, 2001.

[12] J. S. Plank, S. Atchley, Y. Ding, and M. Beck. Algorithms for high performance, wide-area distributed file downloads. *Parallel Processing Letters*, 13(2):207–224, June 2003.

[13] 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.

[14] J. S. Plank and M. G. Thomason. A practical analysis of low-density parity-check erasure codes for wide-area storage applications. In *DSN-2004: The International Conference on Dependable Systems and Networks*. IEEE, June 2004.

[15] J. S. Plank, R. Wolski, and M. Allen. The effect of timeout prediction and selection on wide area collective operations. In *IEEE International Symposium on Network Computing and Applications (NCA-2001)*, Cambridge, MA, October 2001.

[16] 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.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329+, 2001.

[18] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *5th Symposium on Operating Systems Design and Implementation*. Usenix, 2002.

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

[20] 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):757–768, October 1999.

[21] Z. Zhang and Q. Lian. Reperasure: Replication protocol using erasure-code in peer-to-peer storage network. In *21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, pages 330–339, October 2002.

Appendix

When a file is distributed with the hodgepodge distribution, fifty regionally distinct servers are chosen, and four copies of the file are striped across all fifty servers. Two servers are considered to be regionally distinct if the last two elements of their urls do not match. Of approximately 300 IBP servers, the 133 listed below were determined to be up fairly frequently during May and June 2004, and of the 133 servers in the list, between 59 and 75 regionally distinct servers were typically up at a given moment. Due to the instability of such a large set of globally distributed servers, an additional two copies of the file were striped across the fifty chosen for experimentation. The download tool would tolerate failures as long as each block had at least four available copies, and the tool would only use four copies to complete a download.
The following set of IBP servers was used to generate a hodgepodge distribution of the file:

200.19.119.112

206.220.241.47

aladdin.planetlab.extranet.uni-passau.de

charcoal.cs.ucsb.edu

cisa.cs.ucsb.edu

csplanetlab3.kaist.ac.kr

disk2.lab.ac.uab.edu

disk3.lab.ac.uab.edu

disk5.lab.ac.uab.edu

disk6.lab.ac.uab.edu

disk7.lab.ac.uab.edu

disk8.lab.ac.uab.edu

dschinni.planetlab.extranet.uni-passau.de

dsi.i2.hawaii.edu

dsj2.uits.iupui.edu

i2tools.cookman.edu

ibp-rm.6net.garr.it

ibp.caspur.6net.garr.it

ibp.doshisha.ac.jp

ibp.ibcp.fr

ibp.unifi.6net.garr.it

ibp1.lab.ac.uab.edu

itchy.cs.uga.edu

kupl1.ittc.ku.edu

kupl2.ittc.ku.edu

lefthand.eecs.harvard.edu

pl1.unm.edu
plab1.nec-labs.com
planet1.berkeley.intel-research.net
planet1.cs.huji.ac.il
planet1.cs.rochester.edu
planet1.cs.ucsb.edu
planet1.scs.cs.nyu.edu
planet2.berkeley.intel-research.net
planet2.cs.huji.ac.il
planet2.cs.rochester.edu
planet2.ecse.rpi.edu
planetlab-01.bu.edu
planetlab-1.it.uu.se
planetlab-1.scla.nodes.planet-lab.org
planetlab-2.cmcl.cs.cmu.edu
planetlab-2.cs.princeton.edu
planetlab-2.it.uu.se
planetlab-2.scla.nodes.planet-lab.org
planetlab-3.scla.nodes.planet-lab.org
planetlab02.cs.washington.edu
planetlab03.cs.washington.edu
planetlab1.arizona-gigapop.net
planetlab1.bgu.ac.il
planetlab1.cis.upenn.edu
planetlab1.cnds.jhu.edu
planetlab1.cs.arizona.edu
planetlab1.cs.cornell.edu
planetlab1.cs.northwestern.edu
planetlab1.cs.purdue.edu
planetlab1.cs.ucla.edu
planetlab1.cs.uiuc.edu
planetlab1.cs.unb.ca
planetlab1.cs.virginia.edu
planetlab1.cs.vu.nl
planetlab1.cs.wayne.edu
planetlab1.cse.nd.edu
planetlab1.csres.utexas.edu
planetlab1.diku.dk
planetlab1.eecs.umich.edu
planetlab1.enel.ucalgary.ca
planetlab1.flux.utah.edu
planetlab1.frankfurt.interxion.planet-lab.org
planetlab1.iis.sinica.edu.tw
planetlab1.it.uts.edu.au
planetlab1.koganei.wide.ad.jp
planetlab1.millennium.berkeley.edu
planetlab1.postel.org
planetlab1.ucsd.edu
planetlab1.xeno.cl.cam.ac.uk
planetlab10.millennium.berkeley.edu
planetlab11.millennium.berkeley.edu
planetlab2.bgu.ac.il
planetlab2.cnds.jhu.edu
planetlab2.cs.dartmouth.edu
planetlab2.cs.purdue.edu
planetlab2.cs.ubc.ca
planetlab2.cs.ucla.edu
planetlab2.cs.unb.ca
planetlab2.cs.uoregon.edu
planetlab2.cs.wayne.edu
planetlab2.csres.utexas.edu
planetlab2.dcs.bbk.ac.uk
planetlab2.di.unito.it
planetlab2.flux.utah.edu
planetlab2.inria.fr
planetlab2.it.uts.edu.au
planetlab2.millennium.berkeley.edu
planetlab2.postel.org
planetlab2.tamu.edu
planetlab2.ucsd.edu
planetlab3.cambridge.intel-research.net
planetlab3.cs.uoregon.edu
planetlab3.csres.utexas.edu
planetlab3.flux.utah.edu
planetlab3.millennium.berkeley.edu
planetlab3.ucsd.edu
planetlab3.xeno.cl.cam.ac.uk
planetlab4.millennium.berkeley.edu
planetlab6.millennium.berkeley.edu
planetlab6.nbgisp.com
planetlab7.millennium.berkeley.edu
planetlab7.nbgisp.com
planetlab8.idsl.nodes.planet-lab.org
planetlab8.millennium.berkeley.edu
planetlab9.millennium.berkeley.edu
planetslug1.cse.ucsc.edu

planlab1.cs.caltech.edu

planlab2.cs.caltech.edu

pli1-crl-1.crl.hpl.hp.com

pli1-crl-2.crl.hpl.hp.com

pli1-pa-3.hpl.hp.com

pli2-pa-1.hpl.hp.com

pli2-pa-2.hpl.hp.com

portal.grid.csp.it

raven.cs.ucsb.edu

recall.snu.ac.kr

ricepl-2.cs.rice.edu

righthand.eecs.harvard.edu

scratchy.cs.uga.edu

silo.showcase.surfnet.nl

valnure.cs.ucsb.edu

video.ils.unc.edu

vn1.cs.wustl.edu

vrvs-ag.internet2.edu

vrvs3.internet2.edu

w20gva.inria.datatag.org

watson.ecs.baylor.edu