To the Graduate Council:

I am submitting herewith a thesis written by Rebecca Lynn Collins entitled “A Framework for Downloading Wide-Area Files”. I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

James Plank
Major Professor

We have read this thesis
and recommend its acceptance:

Brad Vander Zanden

Micah Beck

Accepted for the Council:

Anne Mayhew
Vice Chancellor and
Dean of Graduate Studies

(Original signatures are on file in the Graduate Student Services Office.)

A Framework for Downloading Wide-Area Files

A Thesis
Presented for the
Master of Science Degree
The University of Tennessee, Knoxville

Rebecca Lynn Collins
December 2004

Acknowledgments

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. I would like to thank Scott Atchley for helpful discussions, and Larry Peterson for PlanetLab access. I would also like to thank Dr. Beck and Dr. Vander Zanden for serving on my graduate committee, and Dr. Langston for the enrichment I gained when I worked in his research group. Finally, I would like to thank my advisor, Dr. Plank, whose guidance and instruction have been invaluable to my education.

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 different algorithms addressing this challenge have been proposed and evaluated. While both have been successful in different performance scenarios, there has been no unifying work that can view both algorithms under a single framework. In this thesis, we define such a framework, where download algorithms are defined in terms of the four dimensions that the client always controls: 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, testing the framework over several types of file distributions. In addition, the additional dependencies and trends that arise when files are augmented with erasure codes rather than replication are examined.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Experimental Testbed . . . 2
  - 1.2.1 Logistical Networking . . . 2
  - 1.2.2 Internet Backplane Protocol (IBP) . . . 2
  - 1.2.3 EXnode, LBone, and LoRS . . . 2
  - 1.2.4 This Work in the Context of Logistical Networking . . . 4

2 Downloading Replicated Wide-Area Files **5**
- 2.1 Framework . . . 5
- 2.2 Algorithms . . . 6
  - 2.2.1 Progress-Driven Redundancy . . . 6
  - 2.2.2 Bandwidth-Prediction Strategy . . . 6
  - 2.2.3 Server Scheduling . . . 7
- 2.3 Experiment . . . 7
- 2.4 Results . . . 9
  - 2.4.1 Broad Trends for Each Dimension . . . 9
  - 2.4.2 Interaction of Server Selection and Threads . . . 11
  - 2.4.3 Where do the blocks come from? . . . 11
  - 2.4.4 The Interaction of P and R . . . 13
  - 2.4.5 When is Aggressive Failover useful? . . . 13
- 2.5 Conclusions . . . 13

3 Erasure Codes in the Wide-Area **16**
- 3.1 Introduction . . . 16
- 3.2 Erasure Coding . . . 17
  - 3.2.1 Reed-Solomon Coding . . . 17
  - 3.2.2 Low Density Parity Check Coding . . . 17
- 3.3 Reed-Solomon Experiments . . . 18
  - 3.3.1 Expected Trends . . . 18
  - 3.3.2 Results . . . 19
- 3.4 Reed-Solomon vs. LDPC . . . 21
  - 3.4.1 Subtleties of LDPC Implementation . . . 22
  - 3.4.2 Broad Trends . . . 23
  - 3.4.3 Block Preferences . . . 24
  - 3.4.4 How Bad is the LDPC Block Overhead? . . . 24

3.5 Conclusions 24

4 Conclusions **28**

Bibliography **30**

Appendix **34**

Vita **37**

List of Tables

2.1 Ranges of parameters explored 8
2.2 Regions used in regional distribution 10

3.1 Reed-Solomon Experiment Space 18
3.2 Summary of Reed-Solomon and LDPC Coding 21
3.3 LDPC vs. Reed-Solomon Experiment Space 21
3.4 Block Preferences 25

List of Figures

1.1 Network Storage Stack 3

2.1 The best performance of threads, redundancy and progress 10
2.2 The best performance of each server scheduling algorithm 10
2.3 Best performance of each server scheduling algorithm plotted over threads 11
2.4 Breakdown of where blocks came from best performances of the fastest$_0$, fastest$_1$, and strict-load algs. with 10 threads (range of average download speeds in Mbps is listed in legend) . . 12
2.5 Breakdown of where blocks came from in the best performances of the fastest$_0$, fastest$_1$, and strict-load algs. with 30 threads (range of average download speeds in Mbps is listed in legend) 12
2.6 Relationship of progress and redundancy with 10 threads over the regional distribution . . . 13
2.7 Relationship of progress and redundancy with 30 threads over the hodgepodge distribution . 14
2.8 Number of failovers with 10 threads, R=2, over the regional distribution 14
2.9 Number of failovers with 30 threads, R=2, over the hodgepodge distribution 14

3.1 Example LDPC code – data blocks on the left and check blocks on the right 17
3.2 Best overall performance given n and m over the Slow Regional Distribution 19
3.3 Best overall performance given n and m over the Regional Distribution 20
3.4 Best overall performance given n and m over the Hodgepodge Distribution 20
3.5 Distribution Woes. The performance of different server scheduling algorithms can vary greatly depending on the distribution of a file with LDPC coding. In distribution 1, the **Lightest-load** algorithm performs best, while in distribution 2, the **Fastest**$_0$ algorithm performs best. 22
3.6 Reed-Solomon vs. LDPC coding (performance including both download and decoding times) 23
3.7 Number of Blocks Started and Finished in Best Performances of Reed-Solomon (RS) and LDPC coding 25
3.8 Total Download Time (beginning of first IBP load to ending of last) in Best Performances . 26

Chapter 1

Introduction

1.1 Motivation

In wide-area, peer-to-peer and Grid file systems [9–11, 13, 16, 24, 28–30], 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, 23], 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* [23].
- 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 [23] 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.

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 inter-operation 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.

In Chapter 2, we explore this framework assuming that the blocks of the file are simply replicated. In Chapter 3, we extend the work of Chapter 3 to include tests where erasure coding is used instead of replication.

1.2 Experimental Testbed

The testbed used for all of our experiments is the Internet Backplane Protocol (IBP). This section provides a brief sketch of IBP and related software.

1.2.1 Logistical Networking

Logistical networking attempts to enhance data movement, computation, and storage by leveraging resources that are available in large scale networks [7]. The label "logistical" plays off the similarities between digital data movement over networks and physical content movement between producers, warehouses, and consumers. One of the goals of logistical networking is to aggregate shared resources in order to provide services that improve the performance of existing applications and in some cases make possible applications that would not be achievable otherwise. Resources utilized by logistical networking must be generic so they can be used for many applications, and they must be scalable so they can grow with the network.

1.2.2 Internet Backplane Protocol (IBP)

IBP was developed to fulfill the needs of logistical networking by providing a generic, best-effort storage service [5, 7]. IBP consists of a server daemon software and client API that allow IBP storage servers, or depots, to accept requests from IBP clients to allocate, store, load and manage data. More recent versions of IBP also allow the client to request transformations to storage allocations [6]. IBP is "best-effort" because it does not make guarantees about the correctness of data or the persistence of storage. Instead, higher level storage requirements such as replication, encryption, and checksumming, are pushed to the endpoints of communication allowing IBP to remain simple and scalable.

1.2.3 EXnode, LBone, and LoRS

Several services depicted in figure 1.1 have been developed on top of IBP to improve the reliability and ease of use for the client. IBP is the lowest layer from which clients can access globally sharable storage. At the next level, the eXnode is used to represent a *network file* that may be made up of several blocks stored in separate IBP allocations, each of which may be replicated to protect against server failures. The Logistical Backbone (L-Bone) maintains an IBP server database that contains information about each server's status, capacity, location, etc. L-Bone servers may be queried by a client to determine which IBP servers can best serve that client's needs. The Logistical Runtime System (LoRS) provides storage tools using all of the underlying layers and provide clients with a high level set of storage features that are conspicuously absent in IBP but are desirable in storage such as encryption, checksumming, replication or erasure coding, and buffering. Using LoRS, users may create and manipulate eXnodes as if they are simply files stored on the network.

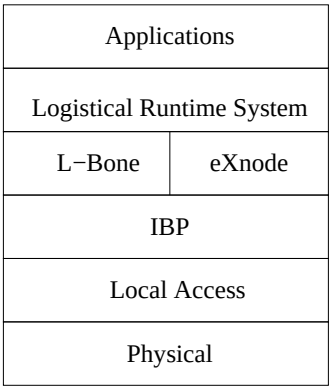


Figure 1.1: Network Storage Stack

1.2.4 This Work in the Context of Logistical Networking

In this work, IBP is used to create a number of different eXnodes that disperse the blocks of a large file across the wide-area in several different geographical distributions. The framework of downloading algorithms presented should be equally applicable to LoRS and to other distributed storage systems that break files into blocks, use replication or erasure coding to add failure protection to the storage, and spread the blocks across the wide-area.

Chapter 2

Downloading Replicated Wide-Area Files

2.1 Framework

In this chapter, a framework is built under which Progress-Driven Redundancy and the Bandwidth-Prediction Strategy can both reside [12]. 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 perform as well as optimally chosen static timeouts [2, 27], 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 thus, 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 2.2.3

2.2 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 2.2.1 and 2.2.2. 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.

2.2.1 Progress-Driven Redundancy

As originally defined [23], 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 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 Progress-Driven Redundancy was initially presented, it was assumed that the file was fully replicated at every site, and threads were assigned to individual servers [23]. This was augmented in [1] so that server selection was performed randomly. In this work, we explore a variety of server selection algorithms.

2.2.2 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 [32] 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.

2.2.3 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 $time * (l/2 + 1)$.

2.3 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 2.1. 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 [24] 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

Table 2.1: Ranges of parameters explored

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

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 one MB blocks. 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 2.2. Note, there are multiple servers in each region, and since these are live servers in 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 used for the hodgepodge distribution along with a more precise description of the distribution is available in the Appendix. 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.

2.4 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.

2.4.1 Broad Trends for Each Dimension

Figures 2.1 and 2.2 show the best performing downloads when parameters for each dimension are fixed. For example, in the leftmost graph of figure 2.1, T ranges from one to thirty, and for each value of T, the combination of P, R and scheduling algorithm that yields the best average download performance is plotted.

Two 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. 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 2.2, 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.

Table 2.2: Regions used in regional distribution

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

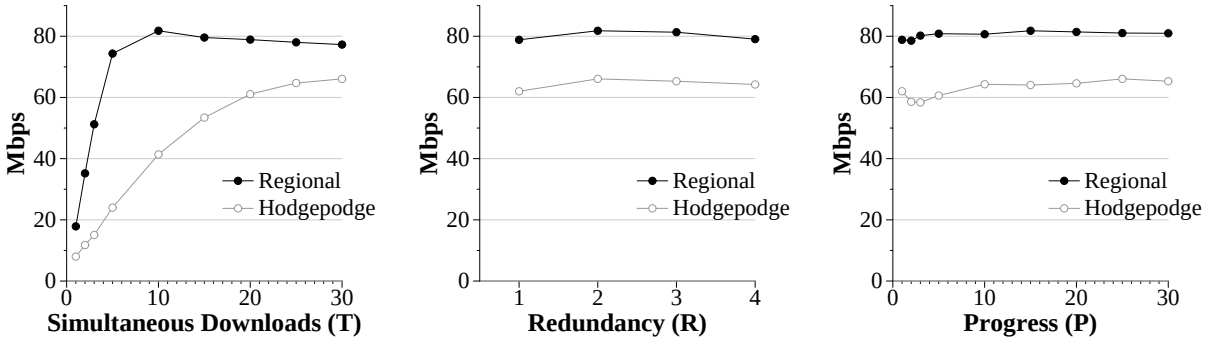


Figure 2.1: The best performance of threads, redundancy and progress

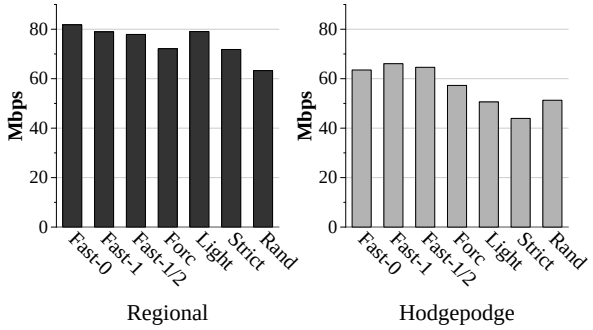


Figure 2.2: The best performance of each server scheduling algorithm

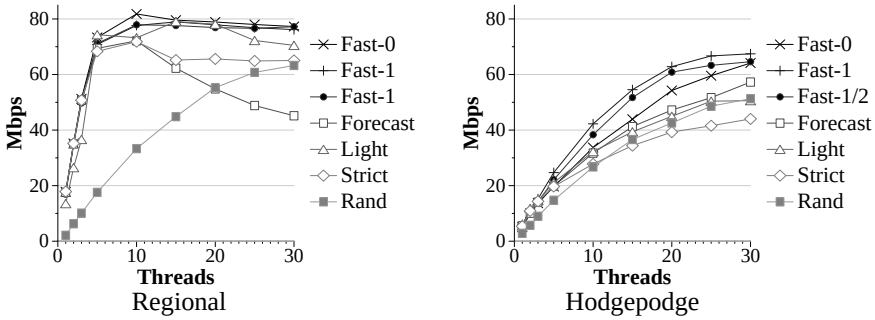


Figure 2.3: Best performance of each server scheduling algorithm plotted over threads

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 may have better performance, and it is possible that the coarse forecaster would perform better 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.

2.4.2 Interaction of Server Selection and Threads

Figure 2.3 gives a more in-depth picture of the interaction of the scheduling algorithms and the number of threads. The best performance of each algorithm given the number of threads is plotted. The overall trends in figure 2.2 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.

2.4.3 Where do the blocks come from?

Figures 2.4 and 2.5 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, the **strict-load** algorithm is forced 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 2.5 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.

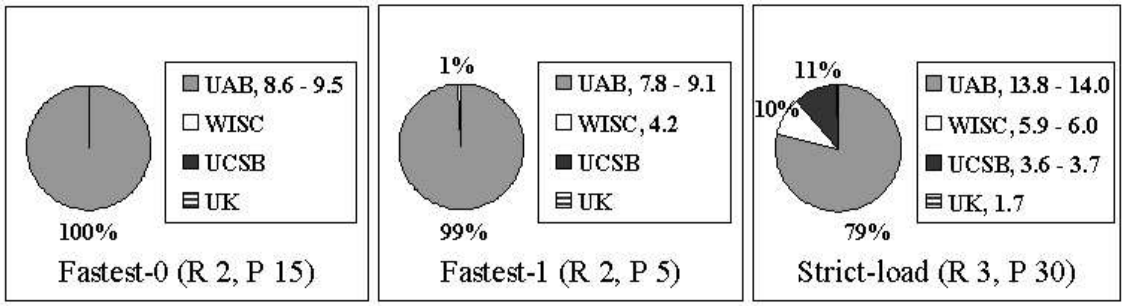


Figure 2.4: Breakdown of where blocks came from best performances of the fastest$_0$, fastest$_1$, and strict-load algs. with 10 threads (range of average download speeds in Mbps is listed in legend)

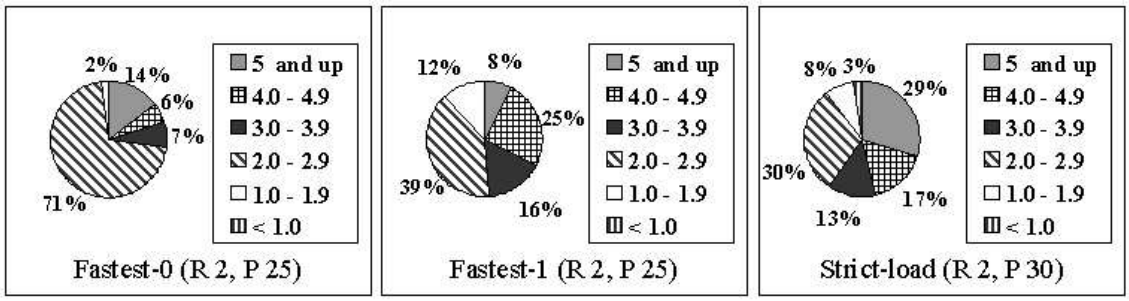


Figure 2.5: Breakdown of where blocks came from in the best performances of the fastest$_0$, fastest$_1$, and strict-load algs. with 30 threads (range of average download speeds in Mbps is listed in legend)

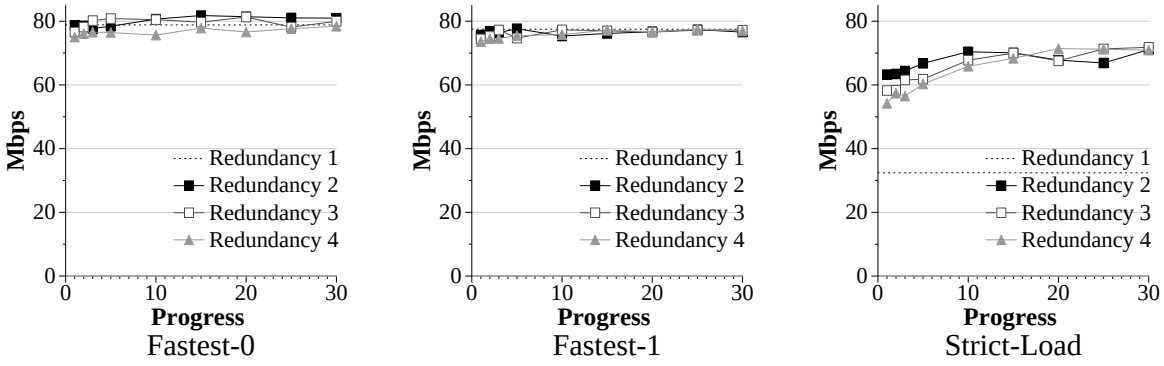


Figure 2.6: Relationship of progress and redundancy with 10 threads over the regional distribution

2.4.4 The Interaction of P and R

The interaction of progress with redundancy is shown in figures 2.6 and 2.7. 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 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.

2.4.5 When is Aggressive Failover useful?

Given that it is sometimes advantageous to make retries, how often is a failover necessary? Figures 2.8 and 2.9 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.

2.5 Conclusions

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

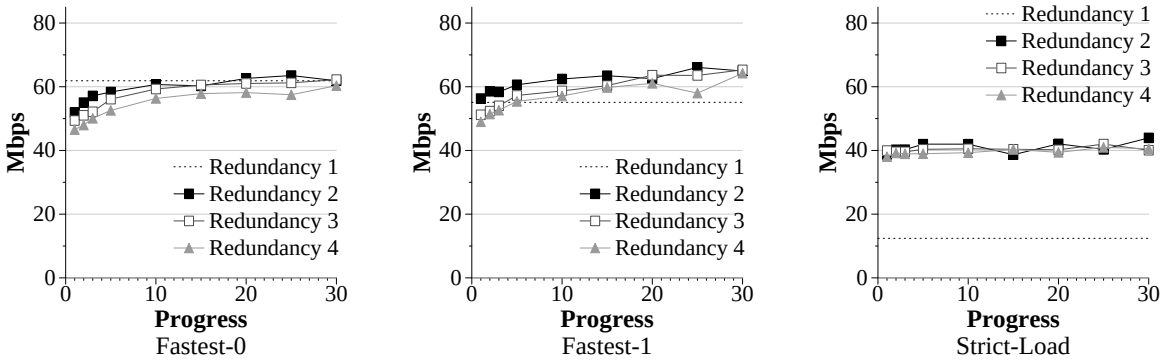


Figure 2.7: Relationship of progress and redundancy with 30 threads over the hodgepodge distribution

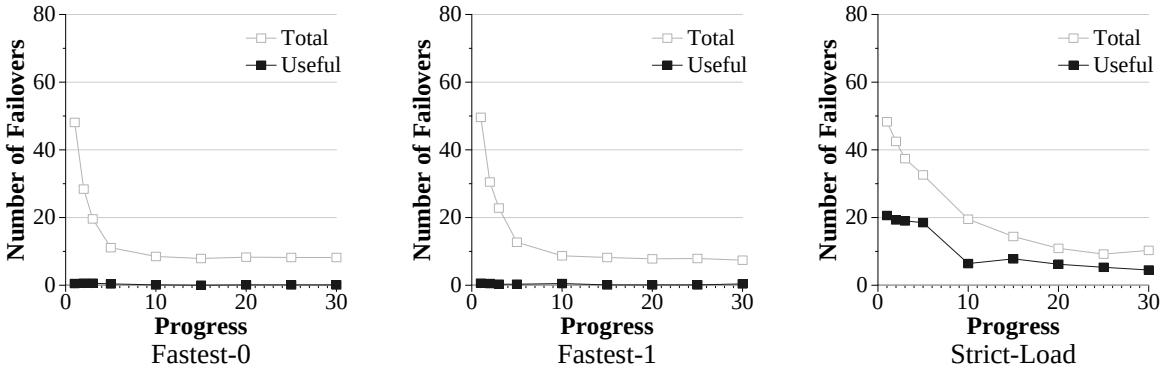


Figure 2.8: Number of failovers with 10 threads, R=2, over the regional distribution

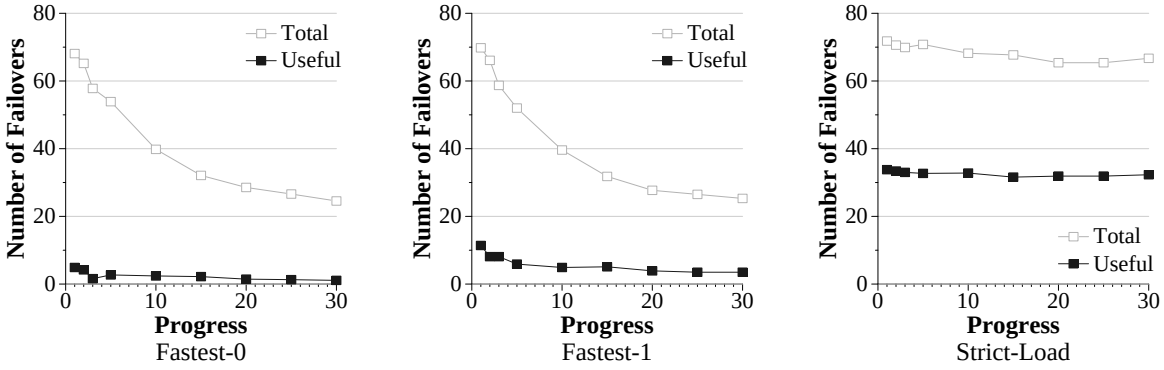


Figure 2.9: Number of failovers with 30 threads, R=2, over the hodgepodge distribution

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 [5], 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 [14].

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.

Finally, we did not assess the impact of timeout-based strategies, which have been shown to be important in some situations [1, 27]. 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.

Chapter 3

Erasure Codes in the Wide-Area

3.1 Introduction

The most natural method of adding redundancy to a file's storage is replication. When a file is broken into blocks and replicas of its blocks are stored in a wide-area file system, if every block has m copies, then $m-1$ server failures can occur before the file becomes unavailable. Though straightforward, replication is expensive in terms of physical storage, and it is limiting in the sense that a client can only retrieve a block by retrieving one of the block's copies. Erasure codes have arisen as a viable alternative to replication for both caching and fault-tolerance in wide-area file systems [8, 26, 31, 33]. In this chapter, we intend to see how the downloading algorithms apply to file systems based on erasure codes, what additional considerations apply, and what the performance impact is.

With erasure coding, n data blocks are used to construct m coding, or check, blocks, where data and check blocks have the same size. The encoding *rate* is $\frac{n}{n+m}$, and some subset of the data and coding blocks may be used to reconstruct the original set of data blocks. Ideally, this subset is made up of any n data or check blocks, though this is not always the case. In comparison to replication, erasure coding techniques reduce the burden of physical storage required to maintain high levels of fault tolerance. However, erasure coding introduces computationally intensive encoding and decoding operations. We focus mainly on download time as a performance metric and not so much on fault tolerance, though performance with respect to both download time and fault tolerance is related. Consider the following: assuming that server speeds follow a uniform distribution, if n remains fixed, then while the m increases, more blocks are being used for the storage of an n-sized data set - improving fault tolerance. On the other hand, since the set can be reconstructed from any n (or approximately any n) blocks, as m increases, the average speed of downloading the fastest n blocks in the entire set will also increase - most likely improving download time. Given an encoding rate, the coding schemes and choice of n and m that result in the best download time may vary depending on underlying system properties such as network and processor speeds. Note that erasure coding is actually a generalization of replication, since replication corresponds to the case when $n = 1$ and decoding and encoding are identity operations.

In the experiments in Chapter 2 that use replication, a file is broken into blocks and four replicas of each block are stored across the wide-area. Erasure coding can be applied to a file broken into blocks by first partitioning the blocks of the file into several sets of size n blocks, and then computing m check blocks for each set independently.

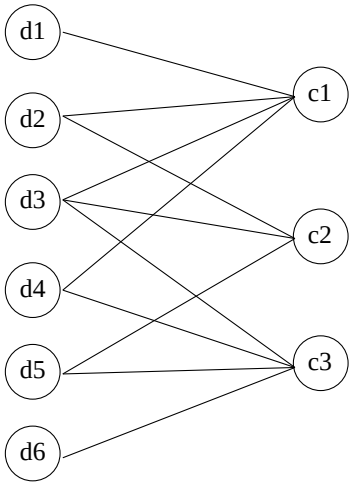


Figure 3.1: Example LDPC code – data blocks on the left and check blocks on the right

3.2 Erasure Coding

We explore erasure codes in the wide-area primarily with Reed-Solomon codes, because they are widely understood and used in a number of applications [15, 17, 18, 21, 33], and because generalizing replication to Reed-Solomon codes in our downloading framework is relatively uncomplicated. Next, we briefly compare Reed-Solomon coding to Low Density Parity Check (LDPC) coding, because LDPC codes have recently arisen that greatly outperform Reed-Solomon codes in some cases [19, 20]. In sections 3.2.1 and 3.2.2, we briefly sketch Reed-Solomon and Low Density Parity Check coding.

3.2.1 Reed-Solomon Coding

Reed-Solomon coding creates m check blocks from n data blocks where any n blocks of either type may be used to reconstruct the set. The check blocks are computed by treating the data blocks like a vector and taking the dot product of the n length data vector and m encoding vectors that compose a Vandermonde-derived matrix. Decoding the data blocks from a set of n data and check blocks requires taking the inverse of an nxn matrix, and computing dot products for each of the missing data blocks. All operations are Galois-field operations. A tutorial written by Plank [21, 25] provides a more complete description of the mathematical details of Reed-Solomon coding.

3.2.2 Low Density Parity Check Coding

Low Density Parity Check (LDPC) coding creates check blocks using only the XOR operation. Figure 3.1 shows an example of a bipartite graph or code that represents the relationship between the data and check blocks. In this example, the value of check block $c1$ is $d1 \oplus d2 \oplus d3 \oplus d4$, $c2$ is $d2 \oplus d3 \oplus d5$, and $c3$ is $d3 \oplus d4 \oplus d5 \oplus d6$. A check block can only be used to reconstruct data blocks that it is adjacent to in the graph. For example, $c1$ is only useful for decoding $d1, d2, d3$, or $d4$. Given these limitations, if we assume that the blocks are downloaded randomly, sometimes more than n blocks are required to reconstruct all of the data blocks. A graph G's overhead, $o(G)$, is the average number of blocks required to decode the entire set. Unlike Reed-Solomon coding, there can be many different codes for the same m and n. Clearly, a code that

Table 3.1: Reed-Solomon Experiment Space

Parameter	Range of Parameters
Simultaneous Downloads	$T \in [20, 30]$
Work Replication and Failover Strategy	$\{R, P\} \in [\{1, 1\}, \{2, (30/n)\}]$, static timeouts
Server Selection	**Fastest**$_0$, **Fastest**$_1$
Coding	$n \in [1, 2, 3, 4, 5, 6, 7, 8, (9)]$, $m \in [1, 2, 3, 4, 5, 6, 7, 8, (9)]$, such that $m/n <= 3$

minimizes $o(G)$ for m and n is desirable, but optimal codes are currently known only for small values of n and m [22].

3.3 Reed-Solomon Experiments

During August 2004, we extended the experiments detailed in Chapter 2 to include erasure coding in addition to replication. Based on the previous results and some preliminary testing, we experimented over the range of parameters in the four dimensions of the framework that were expected to produce the best results; they are listed in table 3.1. We employ regional and hodgepodge network files as described in section 2.3, except that in our regional network file, the UK region is replaced with a Texas region. Also tested is a third network file made up of four regions that are slow relative to the University of Tennessee: Southeastern Canada, Western Europe, Singapore, and Korea. The regional distributions are slightly different from those of replicated network files since each block no longer has 4 copies that can be distributed among the 4 regions. Instead, the blocks of a set are distributed among the regions in a round robin fashion, where region order is chosen randomly for each set in the file.

3.3.1 Expected Trends

There are several trends that we anticipate in our experiments with Reed-Solomon codes:

- Performance should improve as m increases while n is fixed. Having more redundancy means that there is more choice of which n blocks can be used to decode, and more choice means that the average speed of the fastest n blocks in the set is probably faster. A related trend is that performance should decline as n increases while m remains fixed.
- As the set increases and the encoding rate remains fixed, performance will be better or worse based on the impact of two major factors:
    1. A larger set implies a larger server pool from which n blocks can be retrieved, and should improve performance for the following reason: if it is assumed that slow and fast servers are distributed randomly among the sets, then when sets are larger, it is less likely that any one set is composed entirely of slow servers.
    2. Sets with larger n have larger decoding overheads, and we expect performance to decrease when n gets too big. This is somewhat masked for smaller sets because there are many sets in a file and the decoding overhead of one set can overlap the data movement of future sets. However, as n increases, the time it takes to decode a set will eventually overtake the time it takes to download a set. Furthermore, sets near the end of the file often end up being decoded after all of

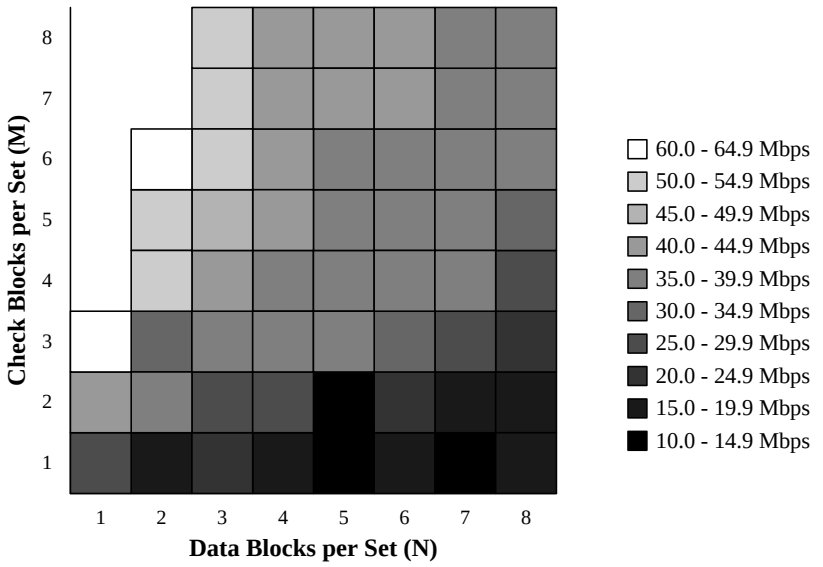


Figure 3.2: Best overall performance given n and m over the Slow Regional Distribution

the downloading has taken place - with larger n, this amounts to larger leftover computation that cannot overlap any data movement.

In the rest of this section, empirical results are presented and compared to the anticipated trends.

3.3.2 Results

Figures 3.2, 3.3, and 3.4 show the best performance of the three distributions when Reed-Solomon coding is used. The trends proposed in section 3.3.1 emerge in several ways, but the different distributions of the blocks appear to strengthen and weaken different trends.

The slow regional distribution shown in figure 3.2 produces a very regular pattern with diagonal bands of performance across the grid. The two trends that dominate are that performance improves as m increases and n remains fixed and performance degrades as n increases and m remains fixed. Furthermore, both trends seem to have equal weight. Performance improves whenever the rate of encoding decreases, but does not show significant changes when the set size increases and the encoding rate stays the same - more specifically, increased set size does not significantly harm performance in smaller sets, nor does it improve performance due to a more advantageous block distribution.

The only difference between the regional distribution and the slow regional distribution is that all of the blocks in the regional distribution are nearby, and none of the blocks in the slow regional distribution are nearby. As such, the performance per block of the regional network file is quite good, and the only trend that emerges in figure 3.3 is that performance degrades as n, and thus the decoding time per set, increases.

The hodgepodge distribution exhibits more interesting behavior in figure 3.4 than either of other distributions. In general, performance improves as m increases and n remains fixed; the columns where $n = 5$ and 6 are good examples of this behavior. Performance also diminishes as n increases; observe for example, the rows where $m = 5$ and 8. These are the same trends that turned up in the slow regional distribution; however, the upper right quadrant of the grid has better performance relative to the rest of the grid than the upper right

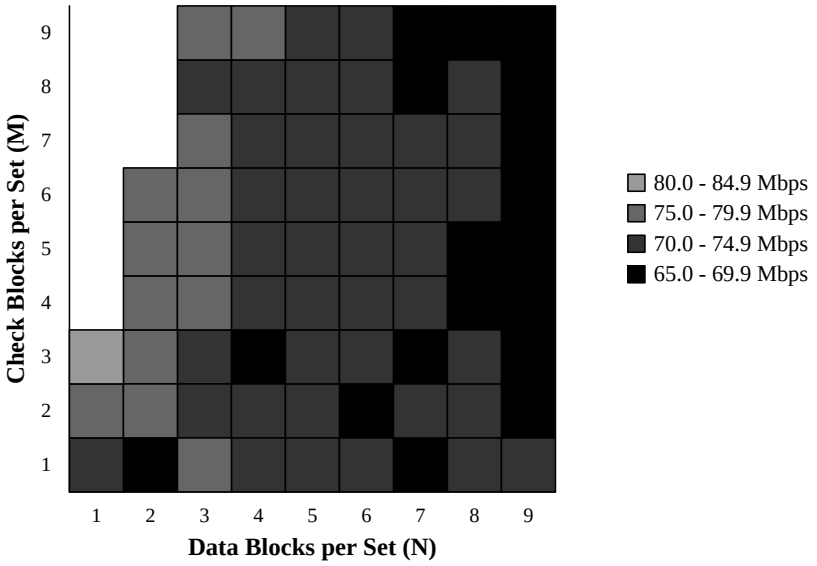


Figure 3.3: Best overall performance given n and m over the Regional Distribution

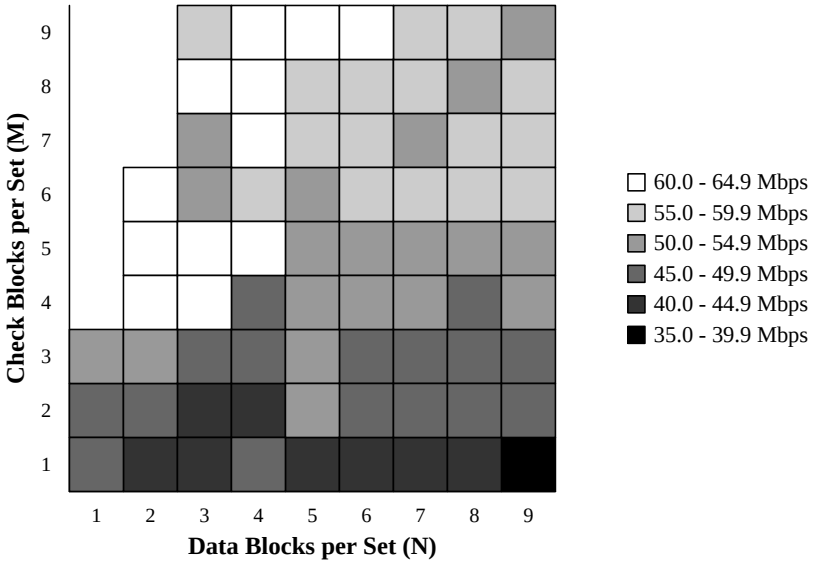


Figure 3.4: Best overall performance given n and m over the Hodgepodge Distribution

Table 3.2: Summary of Reed-Solomon and LDPC Coding

	Reed-Solomon	**LDPC**
Primary Operation	Galois-Field Dot Products	XOR
Quality of Encoding	Optimal	Suboptimal
Decoding Complexity	O(n^3)	O($n \ln(1/\epsilon)$), where $\epsilon \in \mathbb{R}$ [19]

Table 3.3: LDPC vs. Reed-Solomon Experiment Space

Parameter	**Range of Parameters**
Simultaneous Downloads	$T \in [20, 30]$
Work Replication and Failover Strategy	$\{R, P\} \in [\{1, 1\}, \{2, (30/n)\}]$, static timeouts
Server Selection	**Fastest**$_0$, **Fastest**$_1$
Block Selection	**Db-first**, **Dont-care**
Coding	$\{n, m\} \in [\{5, 5\}, \{10, 10\}, \{20, 20\}, \{50, 50\}, \{100, 100\}]$

quadrant of the slow regional grid. There are two possible reasons for this: first, the performance is improving due to distribution advantages that arise in larger sets, and second, the trend that performance improves as m increases is stronger than the trend that performance declines as n increases. (these observations are entirely focused with "small" sets; as n increases beyond a point, decoding will take much longer than downloading, regardless of set size or distribution) The hodgepodge distribution spreads the blocks across 50 servers that are not related to each other in any way, while the regional and slow regional distributions spread the blocks across only 4 regions that typically contain less that 50 servers. Thus, it is likely that the loads of servers in the same region interfere with each other, and the hodgepodge distribution may have performance advantages because of this that are not visible in the regional and slow regional network files.

3.4 Reed-Solomon vs. LDPC

Table 3.2 summarizes some of the key differences between Reed-Solomon and LDPC coding. When comparing the two types of coding, the properties of key importance are the encoding and decoding times, and the average number of blocks that are necessary to reconstruct a set. LDPC codes have a great advantage in terms of encoding and decoding time over Reed-Solomon codes; in addition, Reed-Solomon decoding requires n blocks from a set before decoding can begin, while LDPC decoding can take place on-the-fly. However, for small n, the extra blocks that LDPC codes incur can cause substantial performance degradation, and for systems where the network connection is slow, Reed-Solomon codes can sometimes outperform LDPC codes despite the increased decoding penalty [26].

Table 3.3 shows the parameter space explored in the next set of experiments, which compare Reed-Solomon coding to LDPC coding. Since LDPC coding may fare poorly in very small sets, sets up to size 200 were tested. The experiments test the same values for T, R, P, and server selection algorithm that were used in the Reed-Solomon experiments. In addition, a new block selection criteria based on the type of block is introduced, and will be detailed in section 3.4.1.

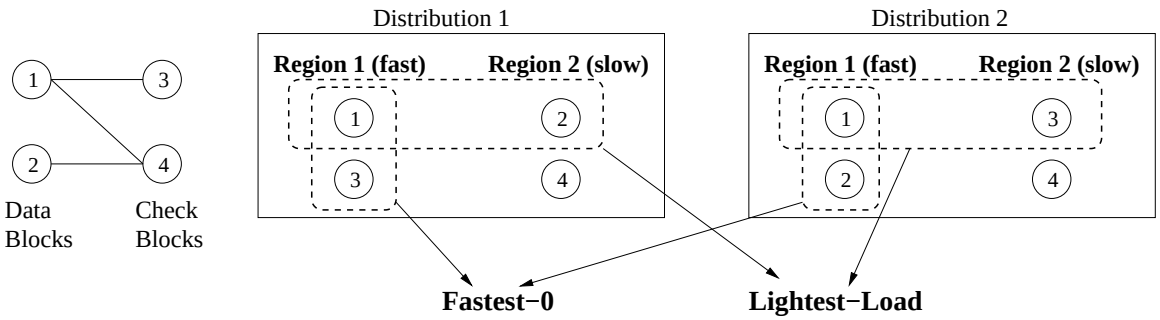


Figure 3.5: Distribution Woes. The performance of different server scheduling algorithms can vary greatly depending on the distribution of a file with LDPC coding. In distribution 1, the **Lightest-load** algorithm performs best, while in distribution 2, the **Fastest**$_0$ algorithm performs best.

3.4.1 Subtleties of LDPC Implementation

The implementation of LDPC coding involves several subtleties that are not present in that of Reed-Solomon coding. First, the fact that LDPC coding sometimes requires more than n blocks affects not only *how many* blocks must be retrieved, but also limits *which* blocks the client application can choose. A wide range of block scheduling strategies may be applied to an LDPC coding set. At one extreme, blocks are downloaded randomly, until enough blocks have been retrieved to decode the set. It is likely that some of the coding blocks will become useless by the time they are retrieved, and that some data blocks may be decoded before they are retrieved. Both of these possibilities increase the number of blocks that are needlessly downloaded. At the other extreme, a download is simulated in order to determine an "optimal" set of blocks that can be used to decode the set. Note that it is possible every time to choose a set of exactly n blocks that can be used for decoding. The difficulty here is that it must be determined up front which blocks are coming from the fastest servers. In our previous experiments we tested a number of different server selection algorithms that judged servers based on speed and on load. The speed of servers remains fixed in most of the server scheduling algorithms, but the load is always dynamic, and any optimal schedule would have to approximate the load of servers not only throughout the download of a given set, but between the downloads of different sets in the file, since they often overlap. Moreover, such a scheduling algorithm is somewhat complicated to implement and may not offer significant performance enhancements once its own computation time is factored into performance. The following experiments use a compromise between the two extreme scheduling options: when selecting a block to download, the downloading algorithm will skip over check blocks that can no longer contribute to decoding and data blocks that are already decoded. The algorithm also allows one of the two block preference to be specified:

- Data blocks first (**db-first**): data blocks are always preferred over check blocks
- Don't care (**dont-care**): the type of blocks is ignored, and blocks are chosen solely based on the speed and load of the servers on which they reside

In the previous experiments over only Reed-Solomon codes, only the **dont-care** algorithm is used.

The second major subtlety in the implementation of LDPC coding is that the distribution of the file can have a great impact on the performance of different server scheduling algorithms. Consider the example depicted in figure 3.5, where $n = 2$, and $m = 2$, and there are two regions. Figure 3.5 shows two possible distributions of the blocks, and which blocks would be chosen from each distribution by the **Fastest**$_0$ and **Lightest-load** server scheduling algorithms. Depending on the distribution, one of the algorithms results

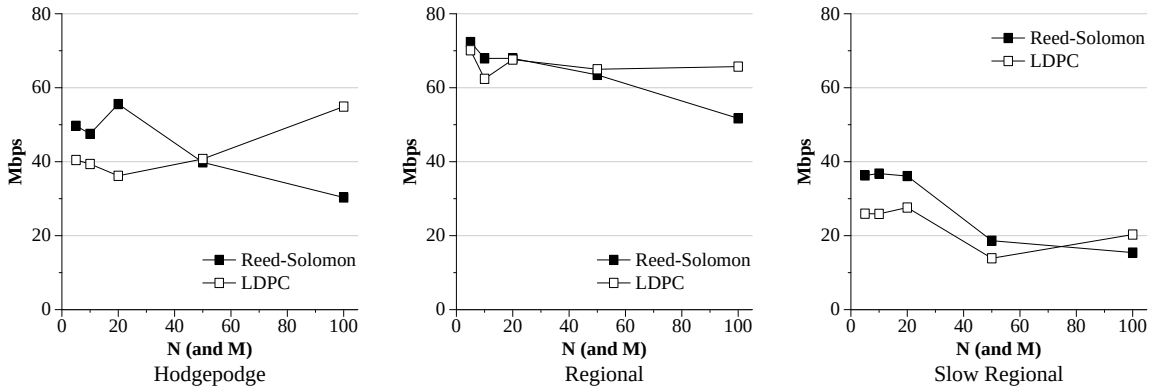


Figure 3.6: Reed-Solomon vs. LDPC coding (performance including both download and decoding times)

in two blocks that can be used to reconstruct the entire set, and the other does not. The unfortunate consequence of this characteristic is that the performance can vary greatly between different server scheduling algorithms not because of the algorithms themselves, but because of the file's distribution. The following experiments use the **Fastest**$_0$, and **Fastest**$_1$ server scheduling algorithms, and same distributions described in section 3.3, which do not address the distribution subtleties of LDPC coding - a decision based on time and sanity constraints.

3.4.2 Broad Trends

The best performing instances of Reed-Solomon and LDPC coding are shown in figure 3.6. LDPC coding performs no better than, and in some cases much worse than Reed-Solomon coding when n is less than 50; however, when n is greater than 50, LDPC coding vastly outperforms Reed-Solomon coding. As n increases past 20, the performance of Reed-Solomon coding steadily declines, while the performance of LDPC coding tends to improve, as in the hodgepodge distribution, or level off, as in the regional distribution. Concerning the best overall performance, the best data point over all set sizes in the hodgepodge distribution occurs at $n = m = 100$ for LDPC coding, while in both the regional and the slow regional distributions, Reed-Solomon achieves the best data point over all at $n = m = 5$ and $n = m = 10$, respectively. When judging the merits of either type of coding scheme in this particular application, it is important to remember that any size set can scale to arbitrarily large files without incurring additional overhead per set. In general, given an application and a choice between Reed-Solomon or LDPC coding it is probably best to choose the scheme and set n,m that:

- is able to scale in the future along with the application.
- does not exceed physical storage limitations.
- meets desired levels of fault tolerance; note that Reed-Solomon coding has stronger guarantees than LDPC coding in this respect.
- satisfies each of the three previous criteria with the best performance where performance consists of both download time and decoding time.

3.4.3 Block Preferences

In these experiments, blocks can be downloaded selectively based on their type. Data blocks are generally preferable to check blocks since fewer data blocks require less decoding, but when check blocks are much closer than data blocks, sometimes the advantage of getting close blocks is worth the decoding penalty. Table 3.4 shows which block preference algorithm is used in the best performing instances of the codes over the three distributions. A trade-off between download time and decoding time is apparent: the **dont-care** algorithm is favored in the Slow Regional distribution and in the Hodgepodge distribution, while the **db-first** algorithm is favored in the Regional distribution; in addition, when $n \geq 50$ (i.e., the decoding time is greater), the **db-first** algorithm is slightly favored.

3.4.4 How Bad is the LDPC Block Overhead?

With LDPC coding, if blocks are downloaded randomly, sometimes more than n blocks must be retrieved to decode a set. As discussed earlier, scheduling can be applied in varying degrees to reduce the number of extra blocks, and hopefully improve performance. The experiments presented here simply keep track of which data blocks are already downloaded, and which check blocks can no longer aid in decoding, and these blocks are not downloaded, or are halted in the case that they have already begun. Figure 3.7 shows the total number of blocks started and finished in both of the coding schemes. LDPC codes start many more blocks than Reed-Solomon codes start, and LDPC codes usually require more than 100 blocks while Reed-Solomon codes only retrieve more than 100 blocks when the downloads of two blocks end at the same time when redundancy has been added to a set. But even though LDPC codes seem to be downloading quite a few more blocks than Reed-Solomon codes, the downloads of extraneous blocks that are started and not finished do not seem to live very long. Figure 3.8 shows the actual time spent downloading for each of the coding schemes measured from the beginning of the first **IBP_load()** to the ending of the last. Note that unlike figure 3.6, figure 3.8 shows only the time taken to download the blocks, and does not include any additional time required to finish decoding the file. In both the Hodgepodge and the Regional distribution, the time spent downloading by the LDPC codes is actually less than the time spent by the Reed-Solomon codes when $n = 100$, which is a by-product of the **db-first** preference that dominates Reed-Solomon performance when n gets large. When a **dont-care** block preference is used along with server selection algorithms that strongly favor the fastest servers, any additional blocks that must be downloaded will typically come from slow servers. So even though more blocks are started and finished when LDPC coding is used, the overall difference in download time is not that great.

3.5 Conclusions

When downloading algorithms are applied to a wide-area file system based on erasure codes, additional considerations must be taken into account. First, performance depends greatly on the interactions of encoding rate, set size, and file distribution. The predominant trend in our experiments is that performance improves as the rate of encoding decreases, and that performance ultimately diminishes as n increases, but to a somewhat lesser extent, it appears that larger sets do have an advantage in terms of download time depending on the distribution of the file. Performance amounts to a balance between the time it takes to download the necessary number blocks and the time it takes to decode the blocks that have been retrieved, and thus distribution, which is intimately related to download time, strengthens and weakens the trends that arise from encoding rate and set size.

Though less thoroughly explored in this work, the type of erasure codes being used also has interactions with encoding rate, set size, and distribution, that shift performance. LDPC codes outperform Reed-Solomon

Table 3.4: Block Preferences

Distribution n, m	LDPC preference	RS preference
Slow Regional 5,5	**dont-care**	**dont-care**
Slow Regional 10,10	**dont-care**	**dont-care**
Slow Regional 20,20	**dont-care**	**dont-care**
Slow Regional 50,50	**db-first**	**dont-care**
Slow Regional 100,100	**dont-care**	**dont-care**
Hodgepodge 5,5	**dont-care**	**dont-care**
Hodgepodge 10,10	**dont-care**	**dont-care**
Hodgepodge 20,20	**dont-care**	**dont-care**
Hodgepodge 50,50	**db-first**	**db-first**
Hodgepodge 100,100	**dont-care**	**db-first**
Regional 5,5	**db-first**	**db-first**
Regional 10,10	**db-first**	**db-first**
Regional 20,20	**db-first**	**dont-care**
Regional 50,50	**db-first**	**db-first**
Regional 100,100	**db-first**	**db-first**

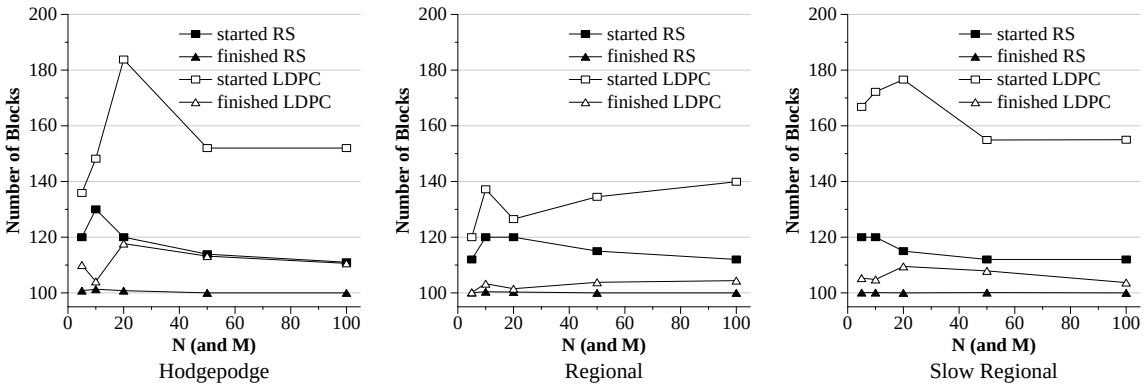


Figure 3.7: Number of Blocks Started and Finished in Best Performances of Reed-Solomon (RS) and LDPC coding

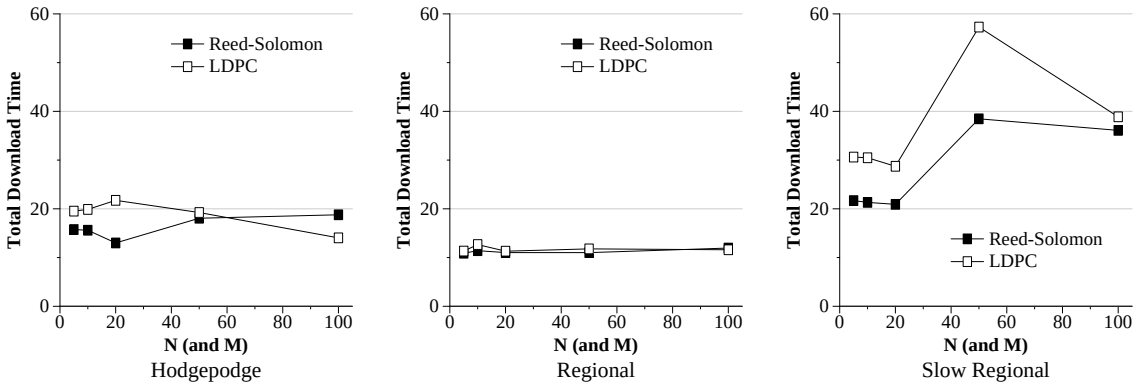


Figure 3.8: Total Download Time (beginning of first IBP load to ending of last) in Best Performances

codes in large sets because LDPC decoding is very inexpensive, but LDPC codes also require more blocks than Reed-Solomon codes and mildly limit which blocks can be used.

In the end, given a specific application or wide-area file system based on erasure codes, decisions about set size, encoding rate, and coding scheme should be based on the following issues: first, what kind of distribution is being used, and can it be changed; second, what level of fault tolerance is necessary, and what set size and coding scheme can achieve this level given the available physical storage; last, given the set size and coding scheme pairs that meet storage constraints and fault tolerance requirements, which has the best performance and the best ability to scale in ways that the file system is likely to scale in the future.

Chapter 4

Conclusions

When downloading a file that is broken into blocks and replicated across a wide-area file system, a client must make decisions along four dimensions:

- How many blocks should be retrieved in parallel?
- Overall, what percentage of the work should be redundant?
- When do we decide that a block must be retried?
- Which replica of a block should be retrieved?

The results in Chapter 2 show that the number of simultaneous downloads and the server selection algorithm have the greatest influence on performance of the first four dimensions. With regard to the two previously studied download strategies, *Bandwidth Prediction* and *Progress-Driven Redundancy*, some form of bandwidth prediction proves advantageous, and while little difference occurs between algorithms that used additional redundancy and those that used no additional redundancy in their best performing instances, in average and worst performing instances, a small amount additional redundancy can improve performance with virtually no cost.

Beneath the downloading framework and seeping into every one of the results is the distribution of the file. What is the mixture of geographical distance and underlying network capacity and speed? How close are the blocks? And what kind of pattern do they have relative to the client? In every instance, distribution affected the overall performance, and in many cases it also had an effect on the performance trends. The client may or may not have control over the distribution of the file, and similarly, the client may or may not have control over whether replication or erasure codes are used, and if erasure codes, what kind.

Chapter 3 explores the implications and additional considerations that arise when erasure codes are used instead of replication. Encoding rate and set size are the parameters of interest, and now the blocks stored on the network are classified as either data blocks or check blocks. Performance tends to improve as the encoding rate decreases, since more choices are available in the network, though as the size of the set increases, the best performance is often achieved by discarding the advantage of close check blocks in favor of data blocks that do not require decoding. Finally, the type of coding scheme also has a great impact on performance because as the size of n increases beyond a point, Reed-Solomon decoding becomes very inefficient, while the complexity of LDPC decoding grows almost linearly with n. Though LDPC codes outperform Reed-Solomon codes whenever n gets very large, there are instances in which Reed-Solomon codes are still the best choice. The results of these experiments provide insight not only into what the client should do when faced with a file that is stored with erasure coding across a wide-area network, but also what size set and distribution that the distributor of data should use.

A topic not addressed in this work, but certainly applicable to erasure codes in the wide-area is the use of distributed computation to decode sets as they travel across the network to the client [6]. Reed-Solomon codes have an advantage with smaller sets since they require fewer blocks for decoding, but LDPC codes have both the advantage of a very simple computation that is easier to deploy, and also the ability to decode on the fly.

Future directions of research in this area could include more extensive comparisons of erasure codes and distributions. In addition, much of the performance is dependent on a balance of data movement and computation; testing the algorithms on several different architectures would shed light into which aspects of the system have the greatest clout. Finally, *Progress Driven Redundancy* has been applied to dynamic job scheduling in a parallel computing application [4]; an exploration of what other aspects of the dynamic data movement algorithms presented here could apply to distributed computing may prove fruitful.

Bibliography

Bibliography

[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, J. Dongarra, J. Huang, T. Moore, and J. Plank. Active logistical state management in the gridsolve/l. In *4th International Symposium on Cluster Computing and the Grid (CCGrid 2004)*, April 2004.

[5] 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.

[6] M. Beck, T. Moore, and J. S. Plank. An end-to-end approach to globally scalable programmable networking. In *FDNA-03: Workshop on Future Directions in Network Architecture, in conjunction with ACM SIGCOMM 2003*, Karlsruhe, Germany, August 2003.

[7] M. Beck, T. Moore, J. S. Plank, and M. Swany. Logistical networking: Sharing more than the wires. In C. A. Lee S. Hariri and C. S. Raghavendra, editors, *Active Middleware Services*. Kluwer Academic, Norwell, MA, 2000.

[8] 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.

[9] 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.

[10] 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.

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

[12] R. L. Collins and J. S. Plank. Downloading replicated, wide-area files – a framework and empirical evaluation. In *3rd IEEE International Symposium on Network Computing and Applications (NCA-2004)*, Cambridge, MA, August 2004.

[13] 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.

[14] 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.

[15] Paul J.M. Havinga. Energy efficiency of error correction on wireless systems, 1999.

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

[17] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[18] Witold Litwin and Thomas Schwarz. LH* RS : A high-availability scalable distributed data structure using reed solomon codes. In *SIGMOD Conference*, pages 237–248, 2000.

[19] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman. Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47(2):569–584, February 2001.

[20] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. Practical loss-resilient codes. pages 150–159, 1997.

[21] 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.

[22] J. S. Plank. Optimal, small, systematic parity-check erasure codes – a brief presentation. Technical Report UT-CS-04-528, Department of Computer Science, University of Tennessee, July 2004.

[23] 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.

[24] 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.

[25] J. S. Plank and Y. Ding. Note: Correction to the 1997 tutorial on reed-solomon coding. Technical Report CS-03-504, University of Tennessee, April 2003.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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

Hodgepodge Distribution Specifics

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

Vita

Rebecca Collins pursued a Bachelor of Science at the University of Tennessee from 1999 to 2003. She graduated Summa Cum Laude with a major in Computer Science and a minor in Mathematics from the College of Arts and Sciences. Subsequently she enrolled in the University of Tennessee's Graduate School to seek a Master of Science Degree in Computer Science.

During the summer of 2002, Rebecca was awarded a fellowship by Pacific Northwest National Laboratory, where she worked as an intern for the System Administration Group of the Environmental Molecular Sciences Lab. She also worked as a research assistant for the Logistical Computing and Internetworking Lab during her senior year as an undergraduate and developed a prototype implementation of the Content-Addressable IBP (IBPCA).

She worked as a research assistant for the Computational Theory and Algorithms Group during her first year as a graduate student, where she developed code for a linear kernelization method for Vertex Cover, and code for applying dynamic load balancing to the fixed parameter tractable (FPT) formulation of Vertex Cover. Later in her graduate studies, she again worked for the Logistical Computing and Internetworking Lab, where she studied algorithms for downloading files that are distributed across wide-area file systems as well as theory in small optimal parity-check erasure codes.

After completing her Master of Science degree, Rebecca intends to pursue a Ph.D. in Computer Science.