On the Practical Use of LDPC Erasure Codes for Distributed Storage Applications

James S. Plank and Michael G. Thomason*

Department of Computer Science
University of Tennessee
[plank,thomason]@cs.utk.edu

Technical Report UT-CS-03-510
http://www.cs.utk.edu/~plank/plank/papers/CS-03-510.html

This paper has been submitted for publication.
Please see the above URL for current publication status.

September 23, 2003

Abstract

As peer-to-peer and widely distributed storage systems proliferate, the need to perform efficient erasure coding, instead of replication, is crucial to performance and efficiency. *Low-Density Parity-Check (LDPC)* codes have arisen as alternatives to standard erasure codes, such as Reed-Solomon codes, trading off vastly improved decoding performance for inefficiencies in the amount of data that must be acquired to perform decoding. The scores of papers written on LDPC codes typically analyze their *collective* and *asymptotic* behavior. Unfortunately, their practical application requires the generation and analysis of *individual* codes for *finite* systems.

This paper attempts to illuminate the practical considerations of LDPC codes for peer-to-peer and distributed storage systems. The three main types of LDPC codes are detailed, and a huge variety of codes are generated, then analyzed using simulation. This analysis focuses on the performance of *individual* codes for *finite* systems, and addresses several important heretofore unanswered questions about employing LDPC codes in real-world systems.

*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.

1 Introduction

Peer-to-peer and widely distributed file systems typically employ replication to improve both the performance and fault-tolerance of file access. Specifically, consider a file system composed of storage nodes distributed across the wide area, and consider multiple clients, also distributed across the wide area, who desire to access a large file. The standard strategy that file systems employ is one where the file is partitioned into n blocks of a fixed size, and these blocks are replicated and distributed throughout the system. Such a scenario is depicted in Figure 1, where a single file is partitioned into eight blocks numbered one through eight, and each block is replicated on four of eight storage servers. Three separate clients are shown accessing the file in its entirety by attempting to download each of the eight blocks from a nearby server.

Replicated systems such as these provide both fault-tolerance and improved performance over non-replicated storage systems. However, the costs are high. First, each block must be replicated m times to tolerate the failure of any $m - 1$ servers. Second, clients must find close copies of each of the file's blocks, which can be difficult, and the failure or slow access of any particular block can hold up the performance of the entire file's access. Aggressive or intelligent downloading strategies have been proposed and evaluated for such systems [AW03].

Erasure encoding schemes (schemes originally de-

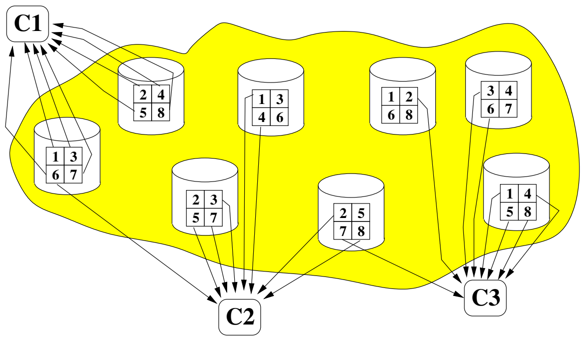


Figure 1: A widely distributed file system hosting a file partitioned into eight blocks, each block replicated four times. Three clients are depicted accessing the file from different network locations.

veloped for communication on the *binary erasure channel (BEC)*) improve both the fault-tolerance and downloading performance of replicated systems [WK02]. For example, with Reed-Solomon erasure encoding, instead of storing the blocks of the files themselves, $n + m$ encodings of the blocks are calculated, and these are stored instead. Now the clients need only download *any* n blocks, and from these, the n blocks of the file may be calculated. Such a scenario is depicted in Figure 2, where 32 encoding blocks, labeled **A** through **Z** and **a** through **f** are stored, and the clients need only access the eight closest blocks to compute the file.

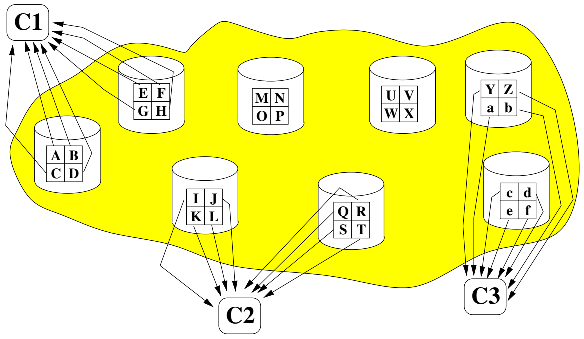


Figure 2: The same system as Figure 1, employing Reed-Solomon coding instead of replication. Again the file is partitioned into eight blocks, but now 32 encoding blocks are stored so that clients may employ *any* eight blocks to calculate the file.

Reed-Solomon coding has been employed effectively in distributed storage systems [RWE+01, KBC+00], and in related functionalities such as fault-tolerant data structures [LS00], disk arrays [BM93] and secret sharing [Rab89]. However, it is not without costs. Specifically, encoding involves breaking each block into words, and each word is calculated as the dot product of two length-n vectors under Galois Field arithmetic, which is more expensive than regular arithmetic. Decoding involves the inversion of an $n \times n$ matrix, and then each of the file's blocks is calculated with dot products as in encoding. Thus, as n grows, the costs of Reed-Solomon coding induce too much overhead [BLMR98].

In 1997, Luby *et al* published a landmark paper detailing a coding technique that thrives where Reed-Solomon coding fails [LMS+97]. Their codes, later termed "Tornado Codes," calculate m coding blocks from the n file blocks in linear time using only cheap exclusive-or (parity) operations. Decoding is also performed in linear time using parity; however, rather than requiring any n blocks for decoding as in Reed-Solomon coding, they require fn blocks, where f is an *overhead factor* that is greater than one, but approaches one as n approaches infinity. A content-distribution system called "Digital Fountain" was built on Tornado Code technology, and in 1998 its authors formed a company of the same name [Dig02].

Tornado Codes are instances of a class of codes called *Low-Density Parity-Check (LDPC)* codes, which have a long history dating back to the 60's [Gal63], but have received renewed attention since the 1997 paper. Since 1998, the research on LDPC codes has taken two paths – Academic research has resulted in many publications about LDPC codes [RGCV03, WK03, SS00, RU03], and Digital Fountain has both published papers [BLM99, Lub02, Sho03] and received patents [LSS+00b, LM01, LSS+00a, Lub01a, Lub01b] on various aspects of coding techniques. The issue of patent infringement is discussed in section 7 below.

LDPC codes are based on graphs, which are used to define codes based solely on parity operations. Nearly all published research on LDPC codes has had the same mission – to define codes that approach "channel capacity" asymptotically. In other words, they define codes where the overhead factor, f, approaches one as n approaches infinity. It has been shown [LMS+97] that codes based on regular graphs – those where each node has a constant incoming and outgoing cardinality – do not have this property. Therefore, the "best" codes are based on randomly generated irregular graphs. A class of irregular graphs is defined, based on probability distributions of node cardinalities, and then properties are proven about the ensemble characteristics of this class. The challenge then becomes to design probability distributions that generate classes of graphs that approach channel capacity. Hundreds of such distributions have been published in the literature and on the web (see Table 1 for 80 examples).

Although the probabilistic method [ASE92] with random graphs leads to powerful characterizations of LDPC ensembles, generating individual graphs from these probability distributions is a non-asymptotic, non-ensemble activity. In other words, while the properties of infinite collections of infinitely sized graphs is known, and while there has been important work in finite-length analysis [DPT+02], the properties of individual, finite-sized graphs, especially for small values of n, have not been explored to date. Moreover, these properties have profound practical consequences.

Addressing aspects of these practical consequences is the goal of this paper. Specifically, we detail how three types of LDPC graphs are generated from given probability distributions, and describe a method of simulation to analyze individual LDPC graphs. Then we generate a wide variety of LDPC graphs and analyze their performance in order to answer the following five practical questions:

1. **What kind of overhead factors (f) can we expect for LDPC codes for small and large values of n?**

2. **Are the three types of codes equivalent, or do they perform differently?**

3. **How do the published distributions fare in producing good codes for finite values of n?**

4. **Is there a great deal of random variation in code generation from a given probability distribution?**

5. **How do the codes compare to Reed-Solomon coding?**

In answering each question, we pose a challenge to the community to perform research that helps networking systems researchers make use of these codes. It is our hope that this paper will spur researchers on LDPC codes to include analyses of the non-asymptotic properties of individual graphs based on their research.

2 Three Types of LDPC Codes

Three distinct types of LDPC codes have been described in the academic literature. All are based on bipartite graphs that are randomly generated from probability distributions. We describe them briefly here. For detailed presentations on these codes, and standard encoding/decoding algorithms, please see other sources [LMS+97, JKM00, Sho00, RU03, WK03].

The graphs have $L + R$ nodes, partitioned into two sets – the *left* nodes, $l_1, \ldots, l_L$, and the *right* nodes, $r_1, \ldots, r_R$. Edges only connect left nodes to right nodes. A class of graphs G is defined by two probability distributions, Λ and P. These are vectors composed of elements $\Lambda_1, \Lambda_2, \ldots$ and $P_1, P_2, \ldots$ such that $\sum_i \Lambda_i = 1$ and $\sum_i P = 1$. Let g be a graph in G. Λ_i is the probability that a left node in g has exactly i outgoing edges, and similarly, P_i is the probability that a right node in g has exactly i incoming edges.[1]

Given L, R, Λ and P, generating a graph g is in theory a straightforward task [LMS+97], We describe our generation algorithm in section 4 below. For this section, it suffices that given these four inputs, we can generate bipartite graphs from them.

To describe the codes below, we assume that we have n equal-sized blocks of data, which we wish to encode into $n + m$ equal-sized blocks, which we will distribute on the network. The nodes of LDPC graphs hold such blocks of data, and therefore we will use the term "node" and "block" interchangeably. Nodes can either initialize their block's values from data, or they may calculate them from other blocks. The only operation used for these calculations is parity, as is common in RAID Level 5 disk arrays [CLG+94]. Each code generation method uses its graph to define an encoding of the n data blocks into $n + m$ blocks that are distributed on the network.

To decode, we assume that we download the fn closest blocks, $B_1, \ldots B_{fn}$, in order. From these, we can calculate the original n data blocks. Our hope is that f is very close to one, and that the encoding and decoding processes are efficient.

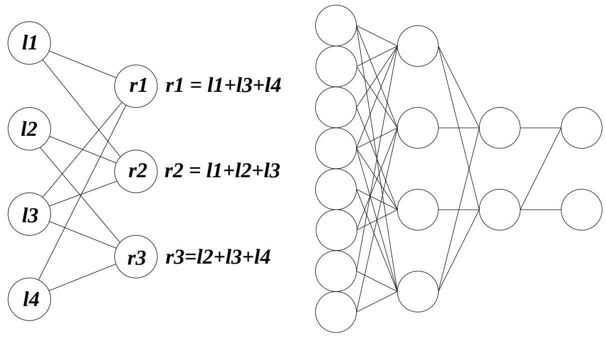
2.1 Systematic Codes

With Systematic codes, $L = n$ and $R = m$. Each left node l_i holds the i-th data block, and each right node r_i is calculated to be the exclusive-or of all the left nodes that are connected to it. A very simple example is depicted in Figure 3(a).

Systematic codes can *cascade*, by employing $d > 1$ levels of bipartite graphs, $g_1, \ldots, g_d$, where the right nodes of g_i are also the left nodes of g_{i+1}. The graph of level 1 has $L = n$, and those nodes contain the n data blocks. The remaining blocks of the encoding are right-hand nodes of the d graphs. Thus, $m = \sum_{i=1}^{d} R_i$. A simple three-level cascaded Systematic code is depicted in Figure 3(b).

Encoding and decoding of both regular and cascading Systematic codes are straightforward operations and are both linear time operations in the number of edges in the graph.

[1] An alternate and more popular definition is to define probability distributions of the edges rather than the nodes using two vectors λ and ρ. The definitions are interchangeable since (Λ, P) may be converted easily to (λ, ρ) and vice versa.

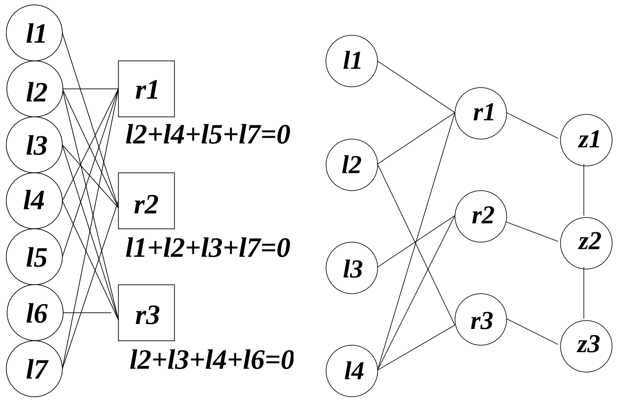


(a) (b)

Figure 3: (a) Example 1-level Systematic code for $n = 4$, $m = 3$. (b) Example 3-level Systematic code for $n = 8$, $m = 8$.

2.2 Gallager (Unsystematic) Codes

Gallager codes were introduced in the early 1960's [Gal63]. With these codes, $L = n + m$, and $R = m$. The first step of creating a Gallager code is to use g to generate a $(n + m) \times n$ matrix M. This is employed to calculate the $n + m$ encoding blocks from the original n data blocks. These blocks are stored in the left nodes of g. The right nodes of g do not hold data, but instead are *constraint* nodes — each r_i has the property (guaranteed by the generation of M) that the exclusive-or of all nodes incident to it is zero. A simple Gallager code is depicted in Figure 4(a).



(a) (b)

Figure 4: (a) Example Gallager code for $n = 4$, $m = 3$. Note that the right nodes define constraints between the left nodes, and do not store encoding blocks. (b) Example IRA code for $n = 4$, $m = 3$. The left and accumulator nodes are stored as the encoding blocks. The right nodes are just used for calculations.

Thus, encoding is an expensive operation, involving the generation of M, and calculation of the encoding blocks. Fortunately, if the graph is low density (i.e. the average cardinality of the nodes is small), M is a sparse matrix, and its generation and use for encoding and decoding is not as expensive as a dense matrix (as is the case with Reed-Solomon coding). Decoding is linear in the number of edges in the graph. Fortunately, M only needs to be generated once per graph, and then it may be used for all encoding/decoding operations.

2.3 IRA Codes

Irregular Repeat-Accumulate (IRA) Codes are Systematic codes, as $L = n$ and $R = m$, and the information blocks are stored in the left nodes. However, an extra set of m nodes, $z_1, \ldots, z_m$, are added to the graph in the following way. Each node r_i has an edge to z_i. Additionally, each node z_i has an edge to z_{i+1}, for $i < m$. These extra nodes are called *accumulator* nodes. For encoding, only blocks in the left and accumulator nodes are stored – the nodes in R are simply used to calculate the encodings and decodings, and these calculations proceed exactly as in the Systematic codes. An example IRA graph is depicted in Figure 4(b).

3 Asymptotic Properties of LDPC Codes

All three classes of LDPC codes have undergone asymptotic analyses that proceed as follows. A *rate* $\mathcal{R} = \frac{n}{n+m}$ is selected, and then Λ and P vectors are designed. From these, it may be proven that graphs generated from the distributions in Λ and P can asymptotically *achieve capacity*. In other words, they may be successfully decoded with fn downloaded blocks, where f approaches 1 from above as n approaches ∞.

Unfortunately, in the real world, developers of wide-area storage systems cannot break up their data into infinitely many pieces. Limitations on the number of physical devices, plus the fact that small blocks of data do not transmit as efficiently as large blocks, dictate that n may range from single digits into the thousands. Therefore, a major question about LDPC codes (addressed by **Question 1** above) is how well they perform when n is in this range.

4 Assessing Performance

Our experimental methodology is as follows. For each of the three LDPC codes, we have written a program to randomly generate a bipartite graph g that defines an instance of the code, given n, m, Λ, P, and a seed for a random number generator. The generation follows the methodology sketched in [LMS+97]:

For each left node l_i, its number of outgoing edges ξ_i is chosen randomly from Λ, and for each right node r_i, its number of incoming edges ι_i is chosen randomly from P. This yields two total number or edges, $T_L = \sum_{i=1}^{L} \xi_i$ and $T_R = \sum_{i=1}^{R} \iota_i$ which may well differ by $D > 0$. Suppose $T_L > T_R$. To rectify this difference, we select a "shift" factor s such that $0 \leq s \leq 1$. Then we subtract sD edges randomly from the left nodes (modifying each ξ_i accordingly), and add $(1-s)D$ edges randomly to the right nodes (modifying each ι_i accordingly). This yields a total of T total edges coming from the left nodes and going to the right nodes.

Now, we define a new graph g' with T left nodes, T right nodes and a random matching of T edges between them. We use g' to define g, by having the first ξ_1 edges of g' define the edges in g coming from l_1. The next ξ_2 edges in g' define the edges coming from l_2, and so on. The right edges of g are defined similarly by the right edges of g' and ι_i.

At the end of this process, there is one potential problem with g — there may be duplicate edges between two nodes, which serve no useful purpose in coding or decoding. We deal with this problem by deleting duplicate edges. An alternative method is to swap edges between nodes until no duplicate edges exist. We compared these two methods and found that neither outperformed the other, so we selected the edge deletion method since it is more efficient.

We evaluate each random graph by performing a Monte Carlo simulation of over 1000 random downloads, and calculating the average number of blocks required to successfully reconstruct the data. This is reported as the overhead factor f above. In other words, if $n = 100$, $m = 100$, and our simulation reports that $f = 1.10$, then on average, 110 random blocks of 200 total blocks are required to reconstruct the 100 original blocks of data from the graph in question.

Theoretical work on LDPC codes typically calculates the *percentage of capacity* of the code, which is $\frac{1}{f}100\%$. We believe that for storage applications, the overhead factor is a better metric, since it quantifies how many block downloads are needed on average to acquire a file.

5 Experiments

5.1 Code Generation

The theoretical work on LDPC codes gives little insight into how the Λ and P vectors that they design will perform for smaller values of n. Therefore we have performed a rather wide exploration of LDPC code generation. First, we have employed 80 different sets of Λ and P from published papers on asymptotic codes. We call the codes so generated *published* codes. These are listed in Table 1, along with the codes and rates for which they were designed. The WK03 distributions are for Gallager codes on AWGN (Additive White Gaussian Noise) channels, and the R03 distributions are for IRA codes on AWGN and binary symmetric channels. In other words, neither is designed for the BEC. We included the former as a curiosity and discovered that they performed very well. We included the latter because distributions for IRA codes are scarce.

Second, we have written a program that generates random Λ and P vectors, determines the ten best pairs that minimize f, and then goes through a process of picking random Λ's for the ten best P's and picking random P's for the ten best Λ's. This process is repeated, and the ten best Λ/P pairs are retained for subsequent iterations. Such a methodology is suggested by Luby *et al* [LMS98]. We call the codes generated from this technique *Monte Carlo* codes.

Third, we observed that picking codes from some probability distributions resulted in codes with an extremely wide range of overhead factors (see section 6.4 below). Thus, our third mode of attack was to take the best performing instances of the published and Monte Carlo codes, and use their left and right node cardinalities to define new Λ's and P's. For example, the Systematic code in Figure 3(a) can be generated from any number of probability distributions. However, it defines a probability distribution where $\Lambda =< 0, 0.75, 0.25 >$ and $P =< 0, 0, 1 >$. These new Λ's and P's may then be employed to generate new codes. We call the codes so generated *derived* codes.

5.2 Tests

The range of potential tests to conduct is colossal. As such, we limited it in the following way. We focus on three rates: $\mathcal{R} \in \{\frac{1}{3}, \frac{1}{2}, \frac{2}{3}\}$. In other words, $m = 2n$, $m = n$, and $m = \frac{n}{2}$. These are the rates most studied in the literature. For each of these rates, we generated the three types of codes from each of the 80 published distributions for all even n between 2 and 150, and for $n \in \{250, 500, 1250, 2500, 5000, 12500, 25000, 50000, 125000\}$[2]. For Systematic codes, we tested cascading levels from one to six.

For Monte Carlo codes, we tested all three codes with all three rates for even $n \leq 50$. As shown in section 6.2 below, this code generation method is only useful for small n.

Finally, for each value of n, we used distributions derived the best current codes for all three coding meth-

[2]One exception is $n = 125000$ for $\mathcal{R} = \frac{1}{3}$, due to the fact that these graphs often exceeded the physical memory of our machines.

Name	Source	# of Codes	Λ_{max}	P_{max}	Developed for	Rate: $[\frac{1}{3}, \frac{1}{2}, \frac{2}{3}]$
L97A	[LMS+97]	2	1,048,577	30,050	Systematic	[0,1,1]
L97B	[LMS+97]	8	8-47	16-28	Systematic	[0,4,4]
S99	[Sho99]	19	2-3298	6-13	Gallager	[4,7,8]
SS00	[SS00]	3	9-12	7-16	Gallager	[0,3,0]
M00	[McE00]	14	2-20	3-8	IRA	[0,6,8]
WK03	[WK03]	6	11-50	8-11	Gallager*	[0,6,0]
RU03	[RU03]	2	8-13	6-7	Gallager	[0,2,0]
R03	[RGCV03]	8	100	8	IRA*	[0,8,0]
U03	[Urb03]	22	6-100	6-19	Gallager	[6,9,7]

Table 1: The 80 published probability distributions (Λ and P) used to generate codes.

ods (and all six cascading levels of Systematic codes) to generate codes for the ten nearest values of n with the same rate. The hope is that good codes for one value of n can be employed to generate good codes for nearby values of n.

In sum, this makes for over 100,000 different data points, each of which was repeated with over 100 different random number seeds. The optimal code and overhead factor for each data point was recorded and the data is digested in the following section.

6 Results

Our computational engine is composed of 160 machines (Sun workstations running Solaris, Dell Pentium workstations running Linux, and a Macintosh PowerBook running OSX) which ran tests continuously for over a month. Tests continue to be run. We organize our results by answering each of the questions presented in Section 1 above.

6.1 Question 1

What kind of overhead factors can we expect for LDPC codes for small and large values of n?

All of our data is summarized in Figure 5. For each value of n and m, the coding and generation method that produces the smallest overhead factor is plotted.

All three curves of Figure 5 follow the same pattern. The overhead factor starts at 1 when $m = 1$ or $n = 1$, and the Systematic codes become simple replication/parity codes with perfect performance. Then the factor increases with n until n reaches roughly twenty at which point it levels out until n increases to roughly 100. At that point, the factor starts to decreases as n increases, and it appears that it indeed goes to one as n gets infinitely large.

Although we only test three rates, it certainly appears that the overhead factor grows as the rate approaches

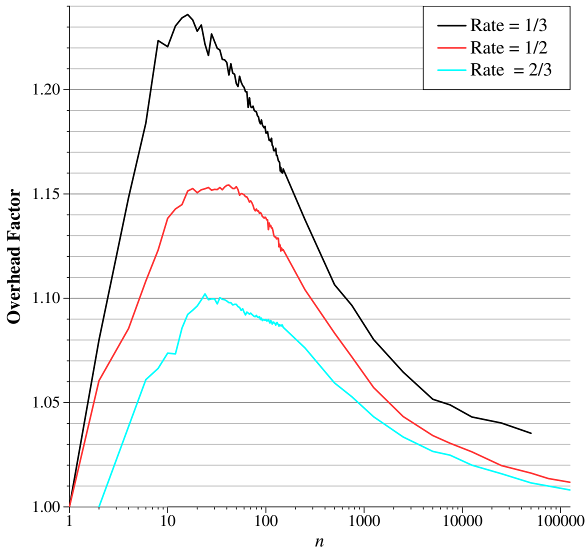


Figure 5: The best codes for all generation methods for $1 \leq n \leq 125,000$, and $\mathcal{R} = \frac{1}{2}, \frac{2}{3}$.

zero. This is intuitive. At one end, any code with a rate of one will have an overhead factor of one. At the other, consider a one-level Systematic code with $n = 3$ and $m = \infty$. There are only seven combinations of the left nodes to which a right node may be connected. Therefore, the right nodes will be partitioned into at most $\frac{m}{7}$ groups, where each node in the group is equivalent. In other words, any download sequence that contains more than one block from a node group will result in overhead. Clearly, this argues for a higher overhead factor.

Challenge for the Community: The shape of the curves in Figure 5 suggests that there is a lower bound for overhead factor as a function of n and m (or alternatively as a function of n and $\mathcal{R}$). It is a challenge to the theoretical community to quantify this lower bound for *finite* values of n and m, and then to specify exact methods for generating optimal or near optimal codes.

6.2 Question 2

Are the three types of codes equivalent, or do they perform differently?

They perform differently. Figures 6 - 8 show the best performing of the three different codes for the three rates. All three show a similar pattern – for small values of n, Systematic codes perform the best. However, when n roughly equals 100, the IRA codes start to outperform the others, and the Gallager codes start to outperform the Systematic codes. This trend continues to the maximum values of n.

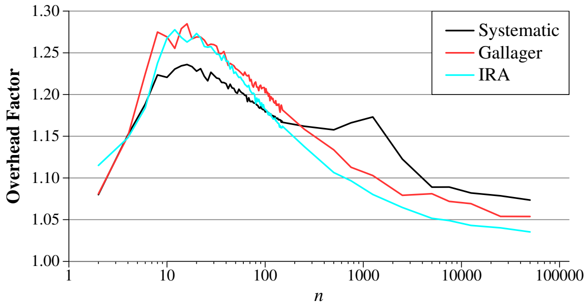


Figure 6: Comparing methods, $\mathcal{R} = \frac{1}{3}$

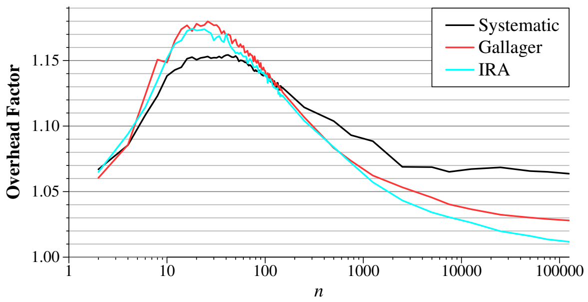


Figure 7: Comparing methods, $\mathcal{R} = \frac{1}{2}$

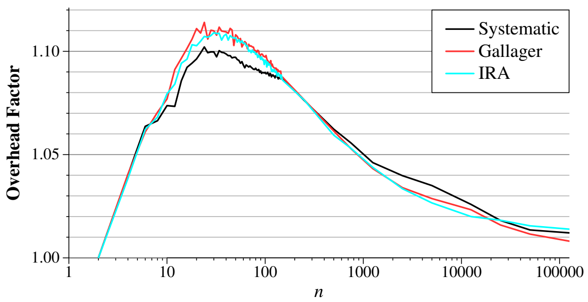


Figure 8: Comparing methods, $\mathcal{R} = \frac{2}{3}$

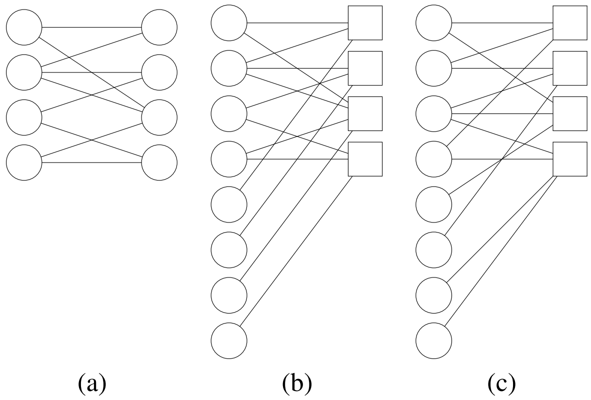


Figure 9: (a) Example 1-level Systematic code for $n = 4$, $m = 4$. (b) Equivalent Gallager code. (c) Gallager code generated from the same Λ and P as (b).

Unfortunately, since the theoretical work on LDPC codes describes only asymptotic properties, little insight can be given as to why this pattern occurs. One curious point is the relationship between one-level Systematic codes and Gallager codes. It is a trivial matter to convert a one-level Systematic code into an equivalent Gallager code by adding m left nodes, $l_{n+1}, \ldots, l_{n+m}$ to the Systematic graph, and m edges of the form (l_{n+i}, r_i) for $1 \leq i \leq m$. An example is the Systematic code in Figure 9(a), which is equivalent to the Gallager code in Figure 9(b). Both have overhead factors of 1.11. This fact would seem to imply that overhead factors for one-level Systematic codes would be similar to, or worse than Gallager codes. However, when $n < 50$, the one-level Systematic codes vastly outperform the others; the Gallager codes perform the worst. A clue to this behavior can be seen in Figure 9(c). This is a Gallager code whose nodes have the same cardinalities as the code in Figure 9(b), and thus would be generated by the same values of Λ and P. However, its overhead factor is 1.21! To hammer this point home further, we performed the same conversion on a Systematic graph where $n = m = 20$, and the overhead factor is 1.16. The node cardinalities of the equivalent Gallager graph were then used to generate values of Λ and P, which in turn were used to generate 500 new Gallager graphs with the exact same node cardinalities. The *minimum* overhead factor of these graphs was 1.31 (the average was 1.45, and the maximum was 1.58). What this suggests is that for smaller graphs, perhaps Λ and P need to be augmented with some other metric so that optimal codes can be generated easily.

Challenge to the community: A rigorous comparison of the practical utility of the three coding methods needs to be performed. In particular, a computationally attractive method that yields (near) optimal codes for finite n would be exceptionally useful. This is highlighted by the fact that one-level Systematic codes vastly outperform Gallager codes for small n, even though equivalent Gallager codes may be constructed from the Systematic codes.

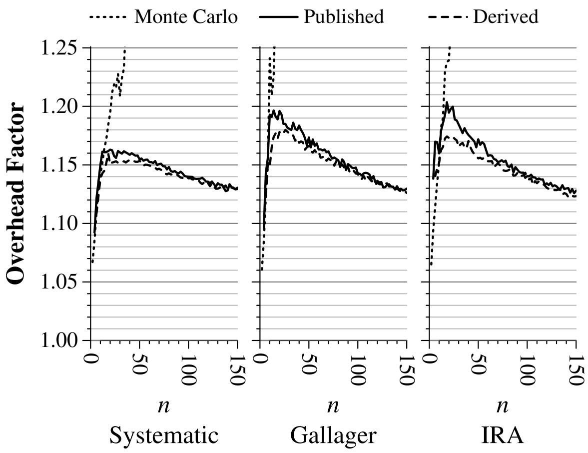


Figure 10: Performance of various codes for $n \leq 150$ when $\mathcal{R} = \frac{1}{2}$.

6.3 Question 3

How do the published distributions fare in producing good codes for finite values of n?

In the next two graphs, we limit our scope to $R = \frac{1}{2}$, as the results for the other two rates are similar. First, we present the performance of the three code generation methods for the three coding methods for small n in Figure 10. As in the other graphs, the best performing instance for each value of n is plotted.

In all coding methods, the Monte Carlo generation method produces better codes than the published distributions when n is roughly less than 15. At that point, the exponential number of possible Λ/P combinations drastically reduces the effectiveness of Monte Carlo code generation. From that point until n is in the high double-digits, the performance of the published codes is noticeably worse than the derived codes. As n grows past 100, the derived and published codes perform roughly equally. Thus, for small $n(< 100)$, the answer to Question 3 is clearly *inadequately*.

Figure 11 addresses which published distributions perform well in generating small codes. Each graph plots four curves – the best codes generated from distributions designed for the particular code and rate, the best codes generated from distributions designed for the particular code, but not for the rate, the best codes generated from distributions designed for other codes, and a reference curve showing the best codes from Figure 10.

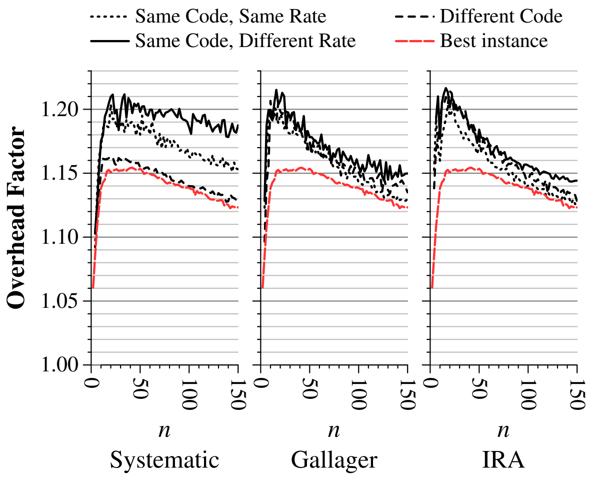


Figure 11: Performance of published distributions for $n \leq 150$ when $\mathcal{R} = \frac{1}{2}$.

Source	Designed for	Rate	Percentage
[McE00]	IRA	2/3	46.6%
[RU03]	Gallager	1/2	24.0%
[Urb03]	Gallager	1/3	12.3%
[Sho99]	Gallager	2/3	11.0%
[Sho99]	Gallager	1/2	4.1%
[Urb03]	Gallager	1/2	2.0%

Table 2: Distributions that generated the best Systematic codes for $n \leq 150$

In all three graphs, the worst codes were generated from distributions designed for the particular code, but for a different rate. In both the Gallager and IRA codes, the best codes were generated from distributions designed for the code and rate; and in the Systematic codes, the best codes were clearly derived from distributions designed for *other* codes. Probing further, Table 2 shows the breakdown of which distributions produced the best Systematic codes. The significance of this is none other than the fact that the derivation of good Systematic codes for small n is clearly not well understood at this point.

For large n, we plot the best published and derived codes for all rates and coding methods in Figure 12. Note that in each graph, the y-axis has a different scale. There are several interesting features of these graphs. In the middle graph, where $\mathcal{R} = \frac{1}{2}$, the published distributions perform best relative to the derived distributions. This is not surprising, since the bulk of the published distributions (46 of the 80) are for $\mathcal{R} = \frac{1}{2}$. For $\mathcal{R} = \frac{2}{3}$, all three coding methods perform similarly in their best instances. For $\mathcal{R} = \frac{1}{3}$, it is not surprising that the published distributions fare poorly in relation to the derived distributions, since only 10 of the 80 published distributions are for $\mathcal{R} = \frac{1}{3}$, and these are only for Gallager codes. It is interesting that given this fact, the derived IRA codes significantly outperform the others. It is also interesting that the published IRA codes for $\mathcal{R} = \frac{2}{3}$ perform so poorly in comparison to the derived codes.

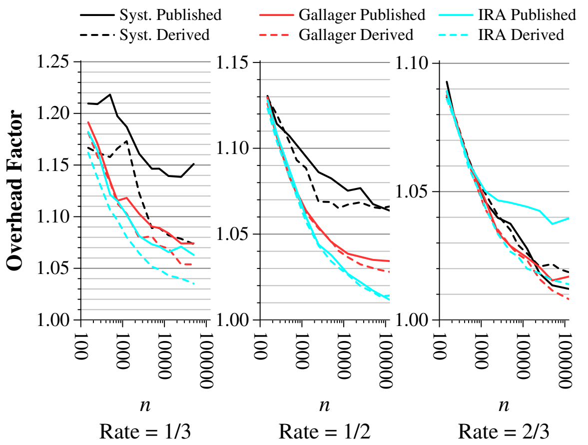


Figure 12: Performance of all codes and rates for large n.

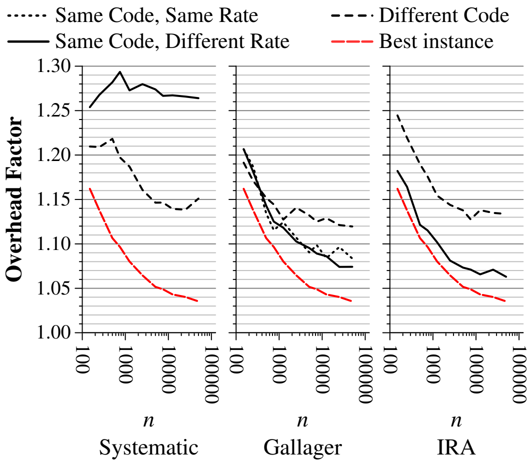


Figure 13: Published Distributions, $\mathcal{R} = \frac{1}{3}$.

Figures 13 - 15 address which published distributions perform well in generating large codes, much like Figure 11 addresses them for small codes. Significant features from these graphs are that only Gallager codes, and IRA codes for $\mathcal{R} = \frac{1}{2}$ benefit from the distributions designed specifically for them. The Systematic codes in particular perform poorly using their own published distributions. As displayed by the IRA graph of Figure 14, the only published distribution that performs optimally (where "optimally" is defined with respect to the codes in our tests) is for IRA codes with $\mathcal{R} = \frac{1}{2}$. This is the second of the eight RCV03 distributions, which, ironically, were not designed for this usage of IRA codes.

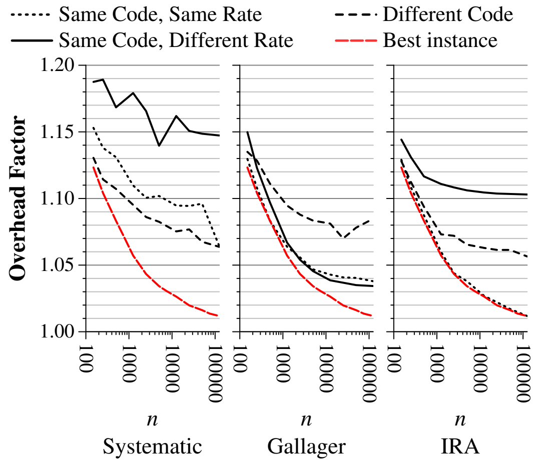


Figure 14: Published Distributions, $\mathcal{R} = \frac{1}{2}$.

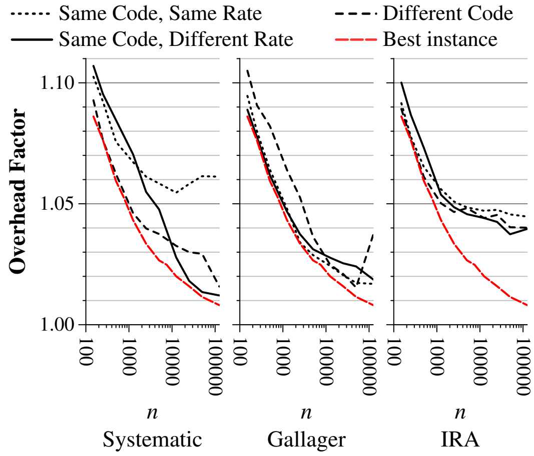


Figure 15: Published Distributions, $\mathcal{R} = \frac{2}{3}$.

Thus, for large n, our answer to Question 3 has to be that the published distributions perform poorly, certainly in relation to the derived codes. To explore this further, we decided to experiment with deriving IRA codes without *any* basis in the published codes. Figure 16 displays the results. In this graph, the dashed lines show our original best IRA codes, derived from the published distributions. The solid lines show the performance of

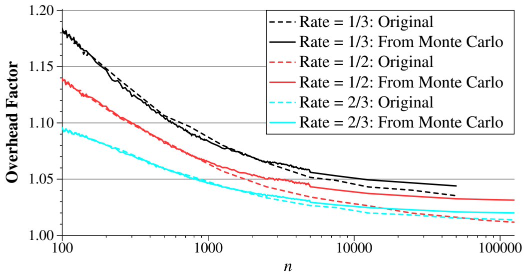


Figure 16: Derived IRA Codes.

Source	Code	Rate Designed	Rate Used	Λ range	P range
S99	Gallager	2/3	2/3	2	6
S99*	Gallager	2/3	1/2	2	6
RU03	Gallager	1/2	1/2	2-13	7
U03	Gallager	1/2	1/2	2-100	10-11
R03	IRA	1/2	1/2	2-100	8
L97A	Syst.	1/2	2/3	3-1M	11-30K

Table 3: Range of code generation for given probability distributions.

Figure 17: The variation in code generation for six selected distributions, $n = 125,000$.

codes derived solely from the Monte Carlo codes for even $n \leq 26$. This is just 13 starting points for each rate. To help in the derivation, we generate codes for the following values of n:

- Even values less than 150
- Multiples of 10 less than 1000
- Multiples of 50 less than 5000
- 7500, 12500, 25000, 50000, 75000 and 125000

As of the submission of this paper, the generations have been proceeding on 90 of our machines for 11 days. For all rates, when $n < 1000$, the codes derived from published distributions perform no better than the ones derived from the 13 starting points. As n progresses higher, the codes derived from published distributions outperform the others, although by less than 0.01 for $\mathcal{R} = \frac{1}{3}$ and $\frac{2}{3}$. As we continue to generate codes, we anticipate that the gap between the two methods will shrink.

6.4 Question 4

Is there a great deal of random variation in code generation from a given probability distribution?

Obviously, this depends on the distribution, and how the distribution is utilized. In Table 3, we select six probability distributions in order to test their variation in code generation. For each of the distributions, we generated over 1000 random codes for $n = 125,000$, and present a digest of the results in Figure 17. For each distribution we draw a Tukey plot [Tuf83], which shows the quartiles for the data and its mean.

The first distribution, S99, from [Sho99], is for a regular graph, where the left nodes each have two outgoing edges, and the right nodes have six incoming edges. As such, we expect little random deviation, which is borne out by the experiments. (We do expect some, because of the random nature of graph generation and of the downloading simulation).

S99* uses the same distribution, but for a different rate. As described in Section 4, when the total number of edges generated by the left and right nodes do not match, edges are added to or subtracted from random nodes until they do match. Thus, even a regular distribution such as this one, when employed for the wrong rate as in this instance, can generate a wide variety of graphs. It is interesting that this distribution produces better codes for the wrong rate, both in the best and median case, than the rate for which it is developed. It is also interesting that this regular graph, which theoretically should achieve an asymptotic overhead factor of $\frac{1}{0.68731} = 1.45$ for $\mathcal{R} = \frac{2}{3}$ [Sho99], in actuality achieves a far better one for both rates.

The next two distributions, RU03 and U03, are for Gallager graphs with rate $\frac{1}{2}$. RU03 is *right regular*, meaning all right-hand nodes have the same number of incoming edges, which is a desirable property, because it simplifies code analysis and distribution generation [Sho99, RU03]. U03 is nearly right regular. Both distributions generate codes with a large spread in performance; however, both have the desirable quality that their medians are very close to their minimum values. In other words, one does not have to generate many codes to get one that performs optimally or near optimally.

The next distribution, for IRA graphs, is also right regular, but has far less desirable generation properties, as it has a very large range of overhead factors, and its median is extremely high. The last distribution, for two-level Systematic codes, is one whose nodes have an exceptionally large range of cardinalities – over a million for left nodes (although with $n = 125,000$, the range is reduced to 32,769), and over 30,000 for right nodes. Interestingly, though, its range of overhead factors is less than R03, although it is still a large range.

While more distributions can be displayed, the bottom line remains the same — some distributions yield good codes with only a few iterations of code generation. Others require a longer time to generate good codes. Clearly, one must generate multiple instances of codes to find one that performs well for given values of n and m.

Challenge To The Community: Besides asymptotic performance, some measure of how quickly a distribution yields good codes in practice should be developed. While distributions such as R03 for IRA graphs and L97A for Systematic graphs do produce excellent codes, they only do so in relatively rare cases, and thus are difficult to employ.

6.5 Question 5

How do the codes compare to Reed-Solomon coding?

The Digital Fountain authors have demonstrated that Tornado codes greatly outperform Reed-Solomon codes for $\mathcal{R} = \frac{1}{2}$ and large n [BLMR98]. On the other end of the spectrum, the implementers of OceanStore, a highly distributed file system, selected Reed-Solomon coding for their erasure-based archival service, employing small values of n and a rate of $\frac{1}{4}$ [RWE$^+$01]. In the middle of these two extremes resides a gray area, which is exacerbated by the fact that the literature on LDPC codes is heavy on theory and light on practice.

Answering this question thoroughly is beyond the scope of this paper. However, in this section, we perform a high-level analysis that should aid researchers faced with the decision of choosing between Reed-Solomon and LDPC codes. Our methodology is as follows. We employ Reed-Solomon coding based on Vandermonde matrices, as described in [Pla97, Riz97, PD03], as there exists a simple public domain implementation in C [Pla03]. We assume that the following metrics can be measured:

- S_{xor}: The average rate of performing bitwise exclusive-or, in MB/s.
- S_{down}: The average rate of downloading blocks, in MB/s.
- S^x_{GF}: The average rate (in MB/s) of multiplying a block of data by a constant, using Galois Field arithmetic $GF(2^x)$, which is employed by Reed-Solomon coding. For practical purposes, we restrict ourselves to $x \in \{8, 16\}$. With $GF(2^8)$, one encodes 8-bit words, and $n + m$ must be less than 256. With $GF(2^{16})$, one encodes 16-bit words, and $n + m < 65536$.
- $f^x_{inv}(n)$: The time to invert an $n \times n$ matrix in $GF(2^x)$.

Given these metrics, the time to download n blocks of size B MB, encoded with Reed-Solomon coding, and then to use them to regenerate an original file is:

$$\frac{Bn}{S_{down}} + f^x_{inv}(n) + n * \left(\frac{Bn}{S^x_{GF}} + \frac{Bn}{S_{xor}}\right). \quad (1)$$

The first term is the download time. The second is the time to invert the encoding matrix, and the third is the time to recalculate the n data blocks (each block is recalculated by a Galois Field dot product of length n vectors).

Given an LDPC code based on a graph with E edges and an overhead factor of f, the time to download fn blocks of size B MB, encoded with the LDPC code, and then to use them to regenerate an original file is roughly:

$$\frac{fBn}{S_{down}} + \frac{EB}{S_{xor}}. \quad (2)$$

For Gallager codes, this does not include the time to manipulate the matrix M.

S_{xor}	637 MB/s
S^8_{GF}	218 MB/s
S^{16}_{GF}	20.2 MB/s
$f^8_{inv}(n)$	$3.83 \times 10^{-8} n^3$ s
$f^{16}_{inv}(n)$	$1.18 \times 10^{-7} n^3$ s
S_{down}	[45.75,1.08,0.256] MB/s

Table 4: Measured metrics for comparing Reed-Solomon and LDPC coding

To compare the various codes, we used the metrics summarized in Table 4. The coding metrics were measured on a Dell Precision 330, with an Intel Pentium 4 processor running Linux, using the public-domain Reed-Solomon coding routines from GFLIB [Pla03]. There are three sets of download times, measured by downloading 1 GB files from various hosts on the Internet.

- **Fast**: This is when the client is connected via 100 Mb Ethernet to our institution's gateway network, whose connectivity is up to 500 Mb to the commodity Internet. Given that, we have observed average download speeds of 45.75 MB/s.

- **Medium**: This is when the client is connected to our institution's wireless network. In this situation, we have observed download speeds of 1.08 MB/s.
- **Slow**: This is when the client is using a private, home wireless network, connected to the Internet via a cable modem to a commercial ISP. In this situation, the download speeds are 0.256 MB/s.

Using these metrics and equations (1) and (2), we calculate the average time to download a 1 GB file using Reed-Solomon coding and the three kinds of LDPC codes. For each value of n, we set B to $\frac{1024}{n}$ MB, and use the values of f and E for the best instances of each code from our earlier tests. In Figures 18 - 20 we show results for each value of S_{down}, with $n \leq 150$. We include the "perfect" download time in these graphs – this is simply (1024 MB)/S_{down}, and represents the time to download the file without coding (or erasures).

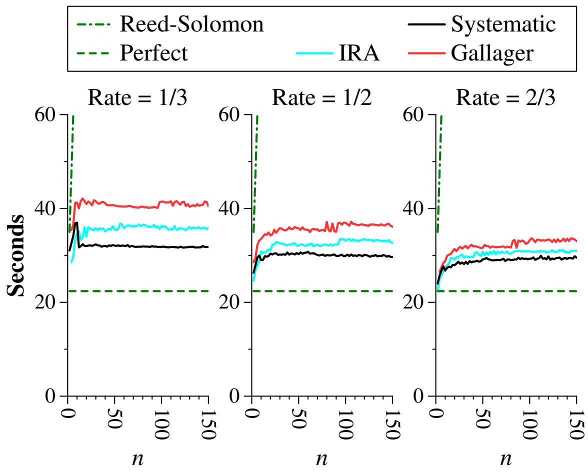


Figure 18: Download times on the **Fast** client.

On the **Fast** client, the LDPC codes significantly outperform Reed-Solomon coding for *all* values of n. This is because the computational overhead of parity and Galois Field arithmetic is the dominant factor in the download's performance, and Reed-Solomon coding, even for small n, is penalized by the expensive Galois Field arithmetic operations. Focusing solely on the LDPC codes, the Systematic codes outperform the IRA codes, which in turn outperform the Gallager codes. This is due to the fact that the underlying graphs have fewer edges – the Gallager graphs have m more left nodes than the others, and the IRA graphs have an extra $2m - 1$ edges for the accumulator nodes.

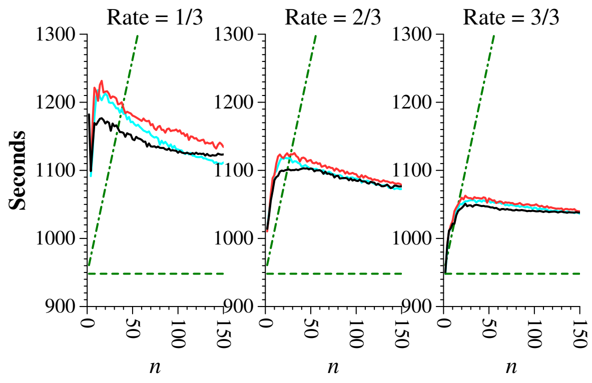


Figure 19: Download times on the **Medium** client.

For the **Medium** client, individual downloads are roughly 40 times slower than the **Fast** client. Thus, the higher overhead factors of LDPC codes have a greater impact. For all rates, Reed-Solomon coding outperforms the LDPC codes for small values of n ($n < 36$ for $\mathcal{R} = \frac{1}{3}$, $n < 26$ for $\mathcal{R} = \frac{1}{2}$, and $n < 12$ for $\mathcal{R} = \frac{2}{3}$). However, its performance rapidly deteriorates because of the expensive arithmetic operations and the $O(n^3)$ matrix inversion. As with the **Fast** client, Systematic codes outperform the other LDPC codes, but this difference becomes less pronounced as n increases, because the download cost, and not the number of edges, becomes the dominant factor.

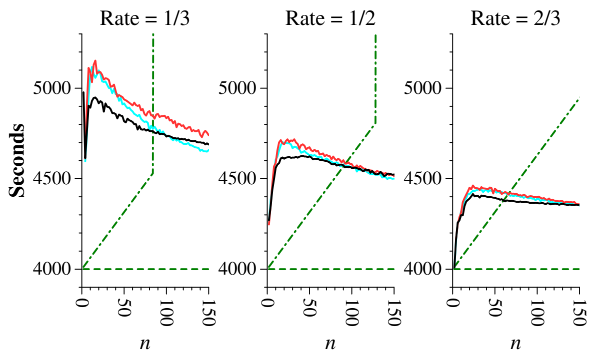


Figure 20: Download times on the **Slow** client.

Finally, on the **Slow** client, the slow download speeds severely penalize the performance of the LDPC codes in favor of Reed-Solomon codes for small values of n. However, as n grows, the matrix inversion time of Reed-Solomon coding again penalizes it to the point where LDPC codes vastly improve upon them. The graphs for $\mathcal{R} = \frac{1}{3}$ and $\mathcal{R} = \frac{1}{2}$ show the points at which $n + m$ becomes greater than 256, and Reed-Solomon coding requires the use of more expensive arithmetic in $GF(2^{16})$.

Thus, the answer to Question 5 is that as download speeds improve, rates increase, and n increases, LDPC codes compare favorably to Reed-Solomon codes. However, when n is small and the download speeds are slow, Reed-Solomon coding can significantly outperform LDPC codes. For example, when downloading to the slow client with $n = m = 10$, Reed-Solomon coding outperforms LDPC codes by over eight minutes.

Challenge to the Community: The analysis in this section is simplistic in that it employs simple equations based on average values. In reality, download speeds can vary greatly from block to block, following non-trivial probability distributions. A better analysis of Question 5 needs to be performed, which either performs actual downloads of data on the wide-area network, or employs download traces to simulate downloads. Additionally, Cauchy Reed-Solomon coding [BKK$^+$95] should be evaluated, as it performs arithmetic operations in $GF(2^1)$ for all values of n and m.

7 Of Patents and Publishing

Due to the extremely legal wording common to most patents, and the fact that patent infringement is only truly proven in court, it is somewhat hazy to us whether licenses from Digital Fountain (from patents [LSS$^+$00b, LSS$^+$00a, LMS$^+$00, LM01]) need to be obtained by those who generate and utilize the LDPC codes described in this paper. Until recently, the "experimental use" exception has protected university researchers from infringing patents when the research conducted is not for profit. However, this exception has recently been brought into question [ES03]. The fact remains that the myriad articles (e.g. [McE00, JKM00, RGCV03]) and books (e.g. [RU03, WK03]) written on LDPC codes based on randomly generated bipartite graphs can likely be brought under fire by Digital Fountain, but whether they are guilty of patent infringement can only be proven in court. Since this paper evaluates, but does not publish codes, it should not infringe these patents.

We have had conversations with members of Digital Fountain's executive team and licensing department, and while we have no official words from them, they appear to be supportive of academic research on LDPC codes, and focus their concern on commercial infringement. That said, it is unclear whether publishing the best codes from our testing on our web site, or our code generation programs, or even the best probability distributions constitutes a threat to Digital Fountain. Our hope is to work with Digital Fountain to be able to clarify their position by the time this paper completes review.

8 Conclusion

This paper has performed a practical exploration of the utility of LDPC codes for wide-area network storage applications. While the asymptotic properties of these codes have been well studied, we have attempted to illuminate their performance for finite systems. The bottom line is that in terms of downloading overhead, the codes display their worst performance for $10 < n < 100$. Then their performance indeed improves asymptotically. In most cases, especially for smaller n, it appears that generating good instances of the codes is a black art, as deriving probability distributions in a more or less evolutionary way is more successful than using the theoretically based published probability distributions. Clearly there is an opportunity for theoretical research on codes for small n to have very wide-reaching impact.

Our cursory analysis in section 6.5 sheds some light on the tradeoffs between Reed-Solomon and LDPC coding. A significant conclusion from this analysis is that while there are indeed practical situations where Reed-Solomon coding clearly outperforms LPDC codes, there are also practical situations where LPDC codes outperform Reed-Solomon coding for *all* values of n. It is a subject of future work to perform a more thorough analysis of the tradeoff between these two types of codes.

One limitation of the LDPC codes in this paper is that they have not been designed to adjust to different rates. It is easy to envision a situation where a file already broken into n blocks is spread among $n + m$ storage servers, and then m' new servers are added to the system. If the coding method that stores the original $n + m$ blocks can adapt efficiently to a rate of $\frac{n}{n+m+m'}$, then adding new coding blocks to the system is a straightforward and efficient operation. However, if the coding technique must be altered to accommodate the new rate, then old coding blocks must be discarded, and new ones calculated in their place, which will be inefficient. Reed-Solomon codes have the feature that they adapt to any rate, although the same inefficiencies pointed out by section 6.5 apply. New codes called LT codes and Raptor codes, that adapt to any rate with optimal asymptotic performance have been developed by Luby and Shokrollahi [Lub02, Sho03]. It is a subject of future work to perform a practical analysis of these codes.

References

[ASE92] N. Alon, J. W. Spencer, and P. Erdos. *The Probabilistic Method.* John Wiley & Sons, New York, 1992.

[AW03] 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.

[BKK+95] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, August 1995.

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

[BLMR98] 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.

[BM93] W. A. Burkhard and J. Menon. Disk array storage system reliability. In *23rd International Symposium on Fault-Tolerant Computing*, pages 432–441, Toulouse, France, June 1993.

[CLG+94] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[Dig02] Digital Fountain, Inc. Next generation data transfer: the meta-content revolution. A Digital Fountain White Paper, `www.digitalfountain.com`, 2002.

[DPT+02] C. Di, D. Proietti, I. E. Telatar, T. J. Richardson, and R. L. Urbanke. Finite-length analysis of low-density parity-check codes on the binary erasure channel. *IEEE Transactions on Information Theory*, 48:1570–1579, June 2002.

[ES03] E. Ergenzinger and M. Spruill. Basic science in US Universities can infringe patents. *The Scientist*, 17(5), March 2003.

[Gal63] R. G. Gallager. *Low-Density Parity-Check Codes*. MIT Press, Cambridge, MA, 1963.

[JKM00] H. Jin, A. Khandekar, and R. McEliece. Irregular repeat-accumulate codes. In *2nd International Symposium on Turbo codes and Related Topics,*, Brest, France, September 2000.

[KBC+00] 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.

[LM01] M. Luby and M. D. Mitzenmacher. Loss resilient code with double heavy tailed series of redundant layers. U.S. Patent #6,195,777, 2001.

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

[LMS98] M. Luby, M. Mitzenmacher, and A. Shokrollahi. Analysis of random processes via and-or tree evaluation. In *9th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1998.

[LMS+00] M. Luby, M. D. Mitzenmacher, M. A. Shokrollahi, D. A. Spielman, and V. Stemann. Message encoding with irregular graphing. U.S. Patent #6,163,870, 2000.

[LS00] W. Litwin and T. Schwarz. Lh*rs: a high-availability scalable distributed data structure using Reed Solomon codes. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 237–248, 2000.

[LSS+00a] M. Luby, M. A. Shokrollahi, V. Stemann, M. D. Mitzenmacher, and D. A. Spielman. Irregularly graphed encoding technique. U.S. Patent #6,081,909, 2000.

[LSS+00b] M. Luby, M. A. Shokrollahi, V. Stemann, M. D. Mitzenmacher, and D. A. Spielman. Loss resilient decoding technique. U.S. Patent #6,073,250, 2000.

[Lub01a] M. Luby. Information additive code generator and decoder for communication systems. U.S. Patent #6,373,406, 2001.

[Lub01b] M. Luby. Information additive group code generator and decoder for communications systems. U.S. Patent #6,320,520, 2001.

[Lub02] M. Luby. LT codes. In *IEEE Symposium on Foundations of Computer Science*, 2002.

[McE00] R. J. McEliece. Achieving the Shannon Limit: A progress report. Plenary Talk, 38th Allerton Conference, October 2000.

[PD03] 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.

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

[Pla03] J. S. Plank. GFLIB - C procedures for Galois Field arithmetic and Reed-Solomon coding. `http://www.cs.utk.edu/~plank/plank/gflib/index.html`, 2003.

[Rab89] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the Association for Computing Machinery*, 36(2):335–348, April 1989.

[RGCV03] A. Roumy, S. Guemghar, G. Caire, and S. Verdu. Design methods for irregular repeat accumulate codes. In *IEEE International Symposium on Information Theory*, Yokohoma, Japan, 2003.

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

[RU03] T. Richardson and R. Urbanke. Modern coding theory. Draft from `lthcwww.epfl.ch/papers/ics.ps`, August 2003.

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

[Sho99] M. A. Shokrollahi. New sequences of linear time erasure codes approaching the channel capacity. In *Proceedings of AAECC-13, Lecture Notes in CS 1719*, pages 65–76, New York, 1999. Springer-Verlag.

[Sho00] M. A. Shokrollahi. Codes and graphs. *Lecture Notes in Computer Science*, 1770, 2000.

[Sho03] A. Shokrollahi. Raptor codes. Technical Report DR2003-06-001, Digital Fountain, 2003.

[SS00] M. A. Shokrollahi and R. Storn. Design of efficient erasure codes with differential evolution. In *IEEE International Symposium on Information Theory*, Sorrento, Italy, 2000.

[Tuf83] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, Connecticut, 1983.

[Urb03] R. Urbanke *et al*. LdcpOpt - a fast and accurate degree distribution optimizer for LPDC ensembles. `http://lthcwww.epfl.ch/research/ldpcopt/index.php`, 2003.

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

[WK03] S. B. Wicker and S. Kim. *Fundamentals of Codes, Graphs, and Iterative Decoding*. Kluwer Academic Publishers, Norwell, MA, 2003.