A Digital Fountain Approach to Reliable Distribution of Bulk Data

John W. Byers [*] Michael Luby[†] Michael Mitzenmacher[‡] Ashutosh Rege [§]

Abstract

The proliferation of applications that must reliably distribute bulk data to a large number of autonomous clients motivates the design of new multicast and broadcast protocols. We describe an ideal, fully scalable protocol for these applications that we call a digital fountain. A digital fountain allows any number of heterogeneous clients to acquire bulk data with optimal efficiency at times of their choosing. Moreover, no feedback channels are needed to ensure reliable delivery, even in the face of high loss rates.

We develop a protocol that closely approximates a digital fountain using a new class of erasure codes that for large block sizes are orders of magnitude faster than standard erasure codes. We provide performance measurements that demonstrate the feasibility of our approach and discuss the design, implementation and performance of an experimental system.

1 Introduction

A natural solution for software companies that plan to efficiently disseminate new software over the Internet to millions of users simultaneously is multicast or broadcast transmission [24]. These transmissions must be fully reliable, have low network overhead, and support vast numbers of receivers with heterogeneous characteristics. Other activities that have similar requirements include distribution of popular images, database replication and popular web site access. These applications require more than just a reliable multicast protocol, since users wish to access the data at times of their choosing and these access times will overlap with those of other users.

While unicast protocols successfully use receiver initiated requests for retransmission of lost data to provide reliability, it is widely known that the multicast analog of this solution is unscalable. For example, consider a server distributing a new software release to thousands of clients. As clients lose packets, their requests for retransmission can quickly overwhelm the server in a process known as feedback implosion. Even in the event that the server can handle the requests, the retransmitted packets are often of use only to a small subset of the clients. More sophisticated solutions that address these limitations by using techniques such as local repair, polling, or the use of a hierarchy have been proposed [5, 10, 15, 16, 27], but these solutions as yet appear inadequate [19]. Moreover, whereas adaptive retransmission-based solutions are at best unscalable and inefficient on terrestrial networks, they are unworkable on satellite networks, where the back channel typically has high latency and limited capacity, if it is available at all.

The problems with solutions based on adaptive retransmission have led many researchers to consider applying Forward Error Correction based on erasure codes[1] to reliable multicast [6, 17, 18, 20, 22, 23, 24, 25]. The basic principle behind the use of erasure codes is that the original source data, in the form of a sequence of k packets, along with additional redundant packets, are transmitted by the sender, and the redundant data can be used to recover lost source data at the receivers. A receiver can reconstruct the original source data once it receives a sufficient number of packets. The main benefit of this approach is that different receivers can recover from different lost packets using the same redundant data. In principle, this idea can greatly reduce the number of retransmissions, as a single retransmission of redundant data can potentially benefit many receivers simultaneously. (In other applications, such as real-time video, retransmission may also be undesirable due to timing constraints; we emphasize that we are not considering real-time applications here.)

The recent work of Nonnenmacher, Biersack and Towsley [20] defines a hybrid approach to reliable multicast, coupling requests for retransmission with transmission of redundant codewords, and quantifies the benefits of this approach in practice. Their work, and the work of many other au-

[*] UC Berkeley and International Computer Science Institute, Berkeley, California. Research supported in part by National Science Foundation operating grant NCR-9416101. **Email:** byers@icsi.berkeley.edu

[†] International Computer Science Institute, Berkeley, California. Research supported in part by National Science Foundation operating grant NCR-9416101. **Email:** luby@icsi.berkeley.edu

[‡] Digital Systems Research Center, Palo Alto, California. **Email:** michaelm@pa.dec.com

[§] International Computer Science Institute, Berkeley, California. Research supported in part by National Science Foundation operating grant NCR-9416101. **Email:** rege@icsi.berkeley.edu

[1] Erasure codes are sometimes called forward-error correcting codes (FEC codes) in the networking community. However, FEC often refers to codes that detect and correct errors, and these codes are typically implemented in special purpose hardware. To avoid confusion, we always refer to the codes we consider as erasure codes.

thors, focus on erasure codes based on Reed-Solomon codes [7, 16, 17, 18, 22, 23, 24]. The limitation of these codes is that encoding and decoding times are slow on large block sizes, effectively limiting k to small values for practical applications. Hence, their solution involves breaking the source data into small blocks of packets and encoding over these blocks. Receivers that have not received a packet from a given block request retransmission of an additional codeword from that block. They demonstrate that this approach is effective for dramatically reducing the number of retransmissions, when packet loss rates are low (they typically consider 1% loss rates). However, this approach cannot eliminate the need for retransmissions, especially as the number of receivers grows large or for higher rates of packet loss. Their approach also does not enable receivers to join the session dynamically.

To eliminate the need for retransmission and to allow receivers to access data asynchronously, the use of a *data carousel* or broadcast disk approach can ensure full reliability [1]. In a data carousel approach, the source repeatedly loops through transmission of all data packets. Receivers may join the stream at any time, then listen until they receive all distinct packets comprising the transmission. Clearly, the reception overhead at a receiver, measured in terms of unnecessary receptions, can be extremely high using this approach. As shown in [23, 24], adding redundant codewords to the carousel can dramatically reduce reception overhead. These papers advocate adding a fixed amount of redundancy to blocks of the transmission using Reed-Solomon codes. The source then repeatedly loops through the set of blocks, transmitting one data or redundant packet about each block in turn until all packets are exhausted, and then repeats the process. This interleaved approach enables the receiver to reconstruct the source data once it receives sufficiently many packets from each block. The limitation of using this approach over lossy networks is that the receiver may still receive many unnecessary packets from blocks that have already been reconstructed while waiting for the last packets from the last few blocks it still needs to reconstruct.

The approaches described above that eliminate the need for retransmission requests can be thought of as weak approximations of an ideal solution, which we call a *digital fountain*. A digital fountain is conceptually simpler, more efficient, and applicable to a broader class of networks than previous approaches. A digital fountain injects a stream of distinct encoding packets into the network, from which a receiver can reconstruct the source data. The key property of a digital fountain is that the source data can be reconstructed intact from *any* subset of the encoding packets equal in total length to the source data. The digital fountain concept is similar to ideas found in the seminal works of Maxemchuk [13, 14] and Rabin [21]. Our approach is to construct better approximations of a digital fountain as a basis for protocols that perform reliable distribution of bulk data.

We emphasize that the digital fountain concept is quite general and can be applied in diverse network environments. For example, our framework for data distribution is applicable to the Internet, satellite networks, and wireless networks with mobile agents. These environments are quite different in terms of packet loss characteristics, congestion control mechanisms, and end-to-end latency; we strive to develop a solution independent of these environment-specific variables. These considerations motivate us to study, for example, a wide range of packet loss rates in our comparisons.

The body of the paper is organized as follows. In the next section, we describe in more detail the characteristics of the problems we consider. In Section 3, we describe the digital fountain solution. In Section 4, we describe how to build a good theoretical approximation of a digital fountain using erasure codes. A major hurdle in implementing a digital fountain is that standard Reed-Solomon codes have unacceptably high running times for these applications. Hence, in Section 5, we describe Tornado codes, a new class of erasure codes that have extremely fast encoding and decoding algorithms. These codes generally yield a far superior approximation to a digital fountain than can be realized with Reed-Solomon codes in practice, as we show in Section 6. Finally, in Section 7, we describe the design and performance of a working prototype system for bulk data distribution based on Tornado codes that is built on top of IP Multicast. The performance of the prototype bears out the simulation results, and it also demonstrates the interoperability of this work with the layered multicast techniques of [25]. We conclude with additional research directions for the digital fountain approach.

2 Requirements for an Ideal Protocol

We recall an example application in which millions of clients want to download a new release of software over the course of several days. In this application, we assume that there is a distribution server, and that the server will send out a stream of packets (using either broadcast or multicast) as long as there are clients attempting to download the new release. This software download application highlights several important features common to many similar applications that must distribute bulk data. In addition to keeping network traffic to a minimum, a scalable protocol for distributing the software using multicast should be:

- **Reliable:** The file is guaranteed to be delivered in its entirety to all receivers.
- **Efficient:** Both the total number of packets each client needs to receive and the amount of time required to process the received packets to reconstruct the file should be minimal. Ideally, the total time for the download for each client should be no more than it would be had point-to-point connections been used.
- **On demand:** Clients may initiate the download at their discretion, implying that different clients may start the download at widely disparate times. Clients may sporadically be interrupted and continue the download at a later time.
- **Tolerant:** The protocol should tolerate a heterogeneous population of receivers, especially a variety of end-to-end packet loss rates and data rates.

We also state our assumptions regarding channel characteristics. IP multicast on the Internet, satellite transmission, wireless transmission, and cable transmission are representative of channels we consider. Perhaps the most important property of these channels is that the return feedback channel from the clients to the server is typically of limited capacity, or is non-existent. This is especially applicable to satellite transmission. These channels are generally packet based, and each packet has a header including a unique identifier. They are best-effort channels designed to attempt to deliver all packets, but frequently packets are lost or corrupted. Wireless networks are particularly prone to high rates of packet loss and all of the networks we describe are prone to bursty loss periods. We assume that error-correcting codes are used to correct and detect errors within a packet. But if a packet contains more errors than can be corrected, it is discarded and treated as a loss.

The requirement that the solution be reliable, efficient, and on demand implies that client robustness to missing packets is crucial. For example, a client may sporadically be interrupted, continuing the download several times before completion. During the interruptions the server will still be sending out a stream of packets that an interrupted client will miss. The efficiency requirement implies that the total length of all the packets such a client has to receive in order to recover the file should be roughly equal to the total length of the file.

3 The Digital Fountain Solution

In this section, we outline an idealized solution that achieves all the objectives laid out in the previous section for the channels of interest to us. In subsequent sections, we describe and measure a new approach that implements an approximation to this ideal solution that is superior to previous approaches.

A server wishes to allow a universe of clients to acquire source data consisting of a sequence of k equal length packets. In the idealized solution, the server sends out a stream of distinct packets, called encoding packets, that constitute an encoding of the source data. The server will transmit the encoding packets whenever there are any clients listening in on the session. A client accepts encoding packets from the channel until it obtains exactly k packets. In this idealized solution, the data can be reconstructed regardless of which k encoding packets the client obtains. Therefore, once k encoding packets have been received the client can disconnect from the channel. We assume that in this idealized solution that there is very little processing required by the server to produce the encoding of packets and by the clients to recover the original data from k encoding packets.

We metaphorically describe the stream of encoding packets produced by the server in this idealized solution as a *digital fountain.* The digital fountain has properties similar to a fountain of water for quenching thirst: drinking a glass of water, irrespective of the particular drops that fill the glass, quenches one's thirst. The digital fountain protocol has all the desirable properties listed in the previous section and functions over channels with the characteristics outlined in the previous section.

4 Building a Digital Fountain with Erasure Codes

An ideal way to implement a digital fountain is to directly use an erasure code that takes source data consisting of k packets and produces sufficiently many encoding packets to meet user demand. Indeed, standard erasure codes such as Reed-Solomon erasure codes have the ideal property that a decoder at the client side can reconstruct the original source data whenever it receives any k of the transmitted packets. But erasure codes are typically used to stretch a file consisting of k packets into n encoding packets, where both k and n are input parameters. We refer to the ratio n/k as the *stretch factor* of an erasure code. While this finite stretch factor limits the extent to which erasure codes can approximate a digital fountain, a reasonable approximation proposed by other researchers (e.g., [18, 22, 23, 25]), is to set n to be a multiple of k, then repeatedly cycle through transmission of the n encoding packets. The limitation is that for any pre-specified value of n, under sufficiently high loss rates a client may not receive k out of n packets in one cycle. Thus in lossy environments, a client may receive useless duplicate transmissions before reconstructing the source data, decreasing the channel efficiency. But in practice, our experimental results indicate that this source of inefficiency is not large even under very high loss rates and when n is set to be a small multiple of k, such as $n = 2k$, the setting we use in the remainder of the paper.

A more serious limitation regards the efficiency of encoding and decoding operations. As detailed in subsequent sections, the encoding and decoding processing times for standard Reed-Solomon erasure codes are prohibitive even for moderate values of k and n. The alternative we propose is to avoid this cost by using the much faster Tornado codes [11]. As always, there is a tradeoff associated with using one code in place of another. The main drawback of using Tornado codes is that the decoder requires slightly more than k of the transmitted packets to reconstruct the source data. This tradeoff is the main focus of our comparative simulation studies that we present in Section 6. But first, in Section 5, we provide an in-depth description of the way Tornado codes are constructed and their properties.

5 Tornado Codes

In this section, we describe in some detail the construction of a specific Tornado code and explain some of the general principles behind Tornado codes. We first outline how these codes differ from traditional Reed-Solomon erasure codes. Then we give a specific example of a Tornado code based on [11, 12] and compare its performance to a standard Reed-Solomon code. For the rest of the discussion, we will consider erasure codes that take a set of k source data packets and produce a set of ℓ redundant packets for a total of $n = k + \ell$ encoding packets all of a fixed length P.

5.1 Theory

We begin by providing intuition behind Reed-Solomon codes. We think of the ith source data packet as containing the value of a variable x_i, and we think of the jth redundant packet as containing the value of a variable y_j that is a linear combination of the x_i variables over an appropriate finite field. (For ease of description, we associate each variable with the data from a single packet, although in our simulations each packet may hold values for several variables.) For example, the third redundant packet might hold $y_3 = x_1 + x_2\alpha + \ldots + x_k\alpha^{k-1}$, where α is some primitive element of the field. Typically, the finite field multiplication operations are implemented using table lookup and the addition operations are implemented using exclusive-or. Each time a packet arrives, it is equivalent to receiving the value of one of these variables.

Reed-Solomon codes guarantee that successful receipt of any k distinct packets enables reconstruction of the source data. When e redundant packets and $k-e$ source data packets arrive, there is a system of e equations corresponding to the e redundant packets received. Substituting all values corresponding to the k received packets into these equations takes time proportional to $(k-e+1)e$. The remaining subsystem has e equations and e unknowns corresponding to the source data packets not received. With Reed-Solomon codes, this system has a special form that allows one to solve for the unknowns in time proportional to e^2 via a matrix inversion and matrix multiplication.

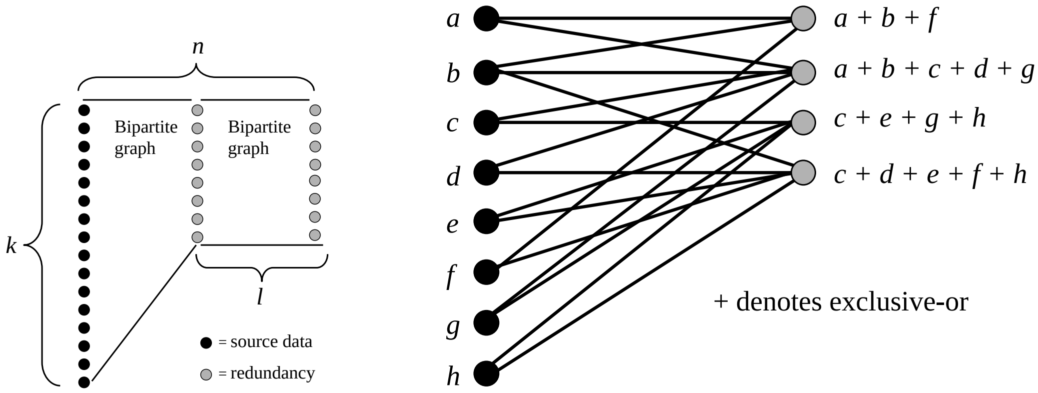

Figure 1: Structure of Tornado Codes

The large decoding time for Reed-Solomon codes arises from the *dense* system of linear equations used. Tornado codes are built using random equations that are *sparse*, i.e. the average number of variables per equation is small. This sparsity allows substantially more efficient encoding and decoding. The price we pay for much faster encoding and decoding is that k packets no longer suffice to reconstruct the source data; instead slightly more than k packets are needed. In fact, designing the proper structure for the system of equations so that the number of additional packets and the coding times are simultaneously small is a difficult challenge [11, 12].

For Tornado codes, the equations have the form $y_3 = x_1 \oplus x_4 \oplus x_7$, where $\oplus$ is bitwise exclusive-or. Tornado codes also use equations of the form $y_{53} = y_3 \oplus y_7 \oplus y_{13}$; that is, redundant packets may be derived from other redundant packets. The encoding time is dominated by the number of exclusive-or operations in the system of equations.

The decoding process for Tornado codes uses two basic operations. The first operation consists of replacing the received variables by their values in the equations in which they appear. The second operation is a simple *substitution rule*. The substitution rule can be applied to recover any missing variable that appears in an equation in which that variable is the unique missing variable. For example, consider again the equation $y_3 = x_1 \oplus x_4 \oplus x_7$. Suppose the redundant packet containing y_3 has been received, as well as the source data packets containing x_1 and x_7, but x_4 has not been received. Then we can use the above equation to solve for x_4, again using only exclusive-or operations. Using this substitution rule repeatedly, a single packet arrival may allow us to reconstitute several additional packets, as the effect of that arrival propagates. In practice, the number of possible substitution rule applications remains minimal until slightly more than k packets have arrived. Then often a single arrival generates a whirlwind of substitutions that allow recovery of all of the remaining source data packets. Hence the name Tornado codes.

The decoding may stop as soon as enough packets arrive so that the source data can be reconstructed. Note that Tornado codes use only exclusive-or operations and avoid both the field operations and the matrix inversion inherent in decoding Reed-Solomon codes. The total number of exclusive-or operations for decoding is at most the number used for encoding, and in general is less.

As we have stated, to reconstruct the source data using a Tornado code, it suffices to recover slightly more than k of the n packets. We say that the *decoding inefficiency* is $1+\epsilon$ if $(1+\epsilon)k$ encoding packets are required to reconstruct the source data. For Tornado codes the decoding inefficiency is not a fixed quantity but depends on the packet loss pattern and the random choices used to construct the code. This variance in decoding inefficiency is described in more detail in Section 5.3.

	Tornado	Reed-Solomon
Decoding inefficiency	$1+\epsilon$ required	1
Encoding times	$(k+\ell)\ln(1/\epsilon)P$	$k\ell P$
Decoding times	$(k+\ell)\ln(1/\epsilon)P$	$k\epsilon P$
Basic operation	XOR	Field operations

Table 1: Properties of Tornado vs. Reed-Solomon codes

The advantage of Tornado codes over standard codes is that they trade off a small increase in decoding inefficiency for a substantial decrease in encoding and decoding times. Recall Reed-Solomon codes have encoding times proportional to $k\ell P$ and decoding times proportional to $k\epsilon P$. As a result, Reed-Solomon codes can only be applied in practice when k and ℓ are relatively small. (Values used in [20, 23, 25, 24] have k and ℓ ranging from 8 to 256.) In contrast, there are families of Tornado codes that have encoding and decoding times that are proportional to $(k+\ell)\ln(1/\epsilon)P$ with decoding inefficiency $1+\epsilon$. And in practice, the encoding and decoding times of Tornado codes are orders of magnitude faster than Reed-Solomon codes for large values of k and ℓ. A summary comparing the properties of Tornado codes and standard Reed-Solomon codes is given in Table 1.

In the next section, we present an example of a fast Tornado code with decoding inefficiency $1+\epsilon \approx 1.054$ whose performance we compare directly with Reed-Solomon codes.

5.2 An Example

We now provide a specific example of a Tornado code. It is convenient to describe the association between the variables and the equations in terms of a levelled graph, as depicted in Figure 1. The nodes of the leftmost level of the graph correspond to the source data. Subsequent levels contain the redundant data.

Each redundant packet is the exclusive-or of the packets held in the neighboring nodes in the level to the left, as depicted on the right side of Figure 1. The number of exclusive-or operations required for both encoding and decoding is thus dominated by the number of edges in the entire graph.

We specify the code by specifying the random graphs to place between consecutive levels. The mathematics behind this code, which we call Tornado Z, is described in [11, 12] and will not be covered here. This code has 16,000 source data nodes and 16,000 redundant nodes. The code uses three levels; the number of nodes in the levels are 16,000, 8,000 and 8,000 respectively.

The graph between the first two levels is the union of two subgraphs, G_1 and G_2. The graph G_1 is based on a *truncated heavy tail* distribution. We say that a level has a truncated heavy tail distribution with parameter D when the fraction of nodes of degree i is $\frac{D+1}{Di(i-1)}$ for $i = 2, \ldots, D+1$. The graph G_1 connects the 16,000 source data nodes to 7,840 of the nodes at the second level. The node degrees on the left hand side are determined by the truncated heavy tail distribution, with $D = 200$. For example, this means that there are $\frac{(16,000)(201)}{(200)(2)(1)} = 8,040$ nodes of degree 2 on the left hand side. Each edge is attached to a node chosen uniformly at random from the 7,840 on the right hand side.[2] The distribution of node degrees on the right hand side is therefore Poisson.

In the second graph G_2, each of the 16,000 nodes on the left has degree 2. The nodes on the right are the remaining 160 nodes at the second level, and each of these nodes has degree 200. The edges of G_2 are generated by randomly permuting the 32,000 edge slots on the left and connecting them in that permuted order to the 160 nodes on the right. The graph G_2 helps prevent small cycles in G_1 from halting progress during decoding.

The second layer uses a graph with a specific distribution, designed using a linear programming tool discussed in [11, 12]. The linear program is used to find graphs that have low decoding inefficiency. In this graph, all of the 8,000 nodes on the left have degree 12. On the right hand side there are 4,093 nodes of degree 5; 3,097 nodes of degree 6; 122 nodes of degree 33; 472 nodes of degree 34; 1 node of degree 141; 27 nodes of degree 170; and 188 nodes of degree 171. The connections between the edge slots on the left and right are selected by permuting the edges slots on the left randomly and then connecting them to the edge slots on the right.

In total there are 222,516 edges in this graph, or approximately 14 edges per source data node. The sparseness of this graph allows for extremely fast encoding and decoding.

5.3 Performance

In practice, Tornado codes where values of k and ℓ are on the order of tens of thousands can be encoded and decoded in just a few seconds. In this section, we compare the efficiency of Tornado codes with standard codes that have been previously proposed for network applications [6, 20, 22, 23, 24, 25]. The erasure code listed in Tables 2 and 3 as *Cauchy* [4] is a standard implementations of Reed-Solomon erasure codes based on Cauchy matrices. (We note that the Cauchy implementation, available at [8], is faster for larger values of k than the implementation of Reed-Solomon codes based on Vandermonde matrices by Rizzo [22].) The Tornado Z codes were designed as described earlier in this section. The implementations were not carefully optimized, so their running times could be improved by constant factors. All experiments were benchmarked on a Sun 167 MHz UltraSPARC 1 with 64 megabytes of RAM running Solaris 2.5.1. All runs are with packet length $P = 1$KB. For all runs, a file consisting of k packets is encoded into $n = 2k$ packets, i.e., the stretch factor is 2.

Encoding Benchmarks		
	Reed-Solomon Codes	Tornado Codes
SIZE	Cauchy	Tornado Z
250 KB	4.6 seconds	0.11 seconds
500 KB	19 seconds	0.18 seconds
1 MB	93 seconds	0.29 seconds
2 MB	442 seconds	0.57 seconds
4 MB	1717 seconds	1.01 seconds
8 MB	6994 seconds	1.99 seconds
16M Bytes	30802 seconds	3.93 seconds

Table 2: Comparison of encoding times.

Decoding Benchmarks		
	Reed-Solomon Codes	Tornado Codes
SIZE	Cauchy	Tornado Z
250 KB	2.06 seconds	0.18 seconds
500 KB	8.4 seconds	0.24 seconds
1 MB	40.5 seconds	0.31 seconds
2 MB	199 seconds	0.44 seconds
4 MB	800 seconds	0.74 seconds
8 MB	3166 seconds	1.28 seconds
16 MB	13629 seconds	2.27 seconds

Table 3: Comparison of decoding times.

For the decoding of the Cauchy codes, we assume that $k/2$ original file packets and $k/2$ redundant packets were used to recover the original file. This assumption holds approximately when a carousel encoding with stretch factor 2 is used, so that roughly half the packets received are redundant packets.

Tornado Z has an average decoding inefficiency of 1.054, so on average $1.054 \cdot k/2$ original file packets and $1.054 \cdot k/2$ redundant packets were used to recover the original file. Our results demonstrate that Tornado codes can be encoded and decoded much faster than Reed-Solomon codes, even for relatively small files.

We note that there is a small variation in the decoding inefficiency for decoding Tornado codes depending on which particular set of encoding packets are received. To study this variation, we ran 10,000 trials using the Tornado Z code. In Figure 2, we show the percentage of trials for which the receiver could not reconstruct the source data for specific values of the decoding inefficiency. For example, using Tornado Z codes with each node representing one packet, a decoding inefficiency of 1.064 corresponds to receiving $17,024 = 1.064 \cdot 16,000$ packets. Over 90% of the clients could reconstruct the source data before receiving this many packets.

In our trials the average decoding inefficiency was 1.0536,

[2] Notice that this may yield some nodes of degree 0 on the right hand side; however, this happens with small probability, and such nodes can be removed. Also, there may be multiple edges between pairs of nodes. This does not affect the behavior of the algorithm dramatically, although the redistribution of such multiple edges improves performance marginally.

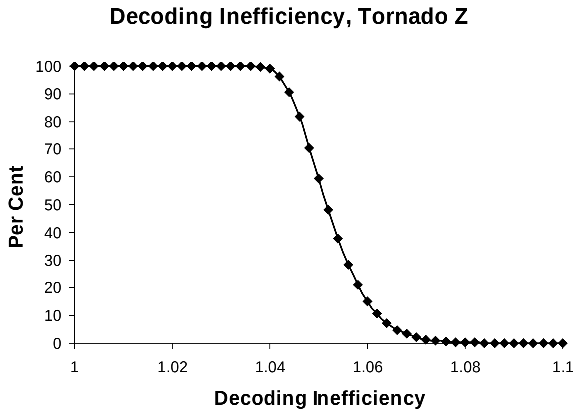

Figure 2: Decoding inefficiency variation over 10,000 trials of Tornado Z.

the maximum reception inefficiency was 1.10, and the standard deviation was 0.0073. For all 10,000 trials the same graph was used; this graph was *not* specially chosen, but was generated randomly as described in Section 5.2. Hence one might achieve better performance by testing various random graphs for performance before settling on one. Our tests suggest that the performance given in Figure 2 is representative.

6 Simulation Comparisons

From the previous section, it is clear that using Reed-Solomon erasure codes to encode over large files for bulk data distribution has prohibitive encoding and decoding overhead. But another approach, described in the introduction, is the method of interleaving suggested in [20, 22, 23, 24]. Interleaved codes are constructed as follows: suppose $K + L$ encoding packets are to be produced from K file packets. Partition the K file packets into blocks of length k, so that there are $B = K/k$ blocks in total. Stretch each block of k packets to an encoding block of $k + \ell$ packets using a standard erasure code by adding $\ell = kL/K$ redundant packets. Then, form the encoding of length $K + L$ by interleaving the encoding packets from each block, i.e., the encoding consists of sequences of B packets, each of which consist of exactly one packet from each block.

The choice of the value of the parameter k for interleaved codes is crucial. To optimize encoding and decoding speed of the interleaved codes, k should clearly be chosen to be as small as possible. But choosing k to be very small defeats the purpose of using encoding, since any redundant packet that arrives can only be used to reconstruct a source data packet from the same block. Moreover, redundant packets that arrive for data blocks that have already been reconstructed successfully do not benefit the sender.

To explain this in more detail, let us say that a block is *full* from the client viewpoint when at least k distinct transmitted packets associated with that block have been received. The entire file can only be decoded by the client when all blocks are full. (Note however that some of the decoding work can potentially be done in the background while packets arrive; the same also holds for Tornado codes.) The phenomenon that arises when k is relatively small is illustrated in Figure 3; while waiting for the last few blocks to

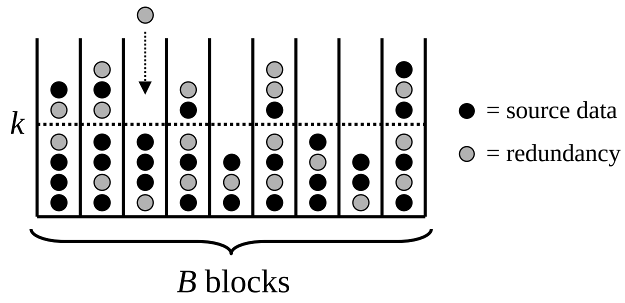

Figure 3: Waiting for the last blocks to fill...

fill, the receiver may receive many packets from blocks that have already been reconstructed successfully. These useless packets contribute directly to the decoding inefficiency. To summarize, the choice of the value of k for interleaved codes introduces a tradeoff between decoding speed and decoding inefficiency.

To compare various protocols, we compare the decoding inefficiency and decoding speed at each receiver. Recall that the decoding inefficiency is $1 + \epsilon$ if one must obtain $(1 + \epsilon)k$ distinct packets in order to decode the source data. For Tornado codes, there is some decoding inefficiency based on how the codes are constructed. For interleaved codes, decoding inefficiency arises because in practice one must obtain more than k packets to have enough packets to decode each block. We emphasize that for interleaved codes the decoding inefficiency is a random variable that depends on the loss rate, loss pattern, and the block size. The tradeoff between decoding inefficiency and coding time for interleaved codes motivates the following set of experiments.

- Suppose we choose k in the interleaved setting so that the decoding inefficiency is comparable to that of Tornado Z. How does the decoding time compare?
- Suppose we choose k in the interleaved setting so that the decoding time is comparable to that of Tornado Z. How does the decoding inefficiency compare?

In our initial simulations, we assume probabilistic loss patterns in which each transmission to each receiver is lost independently with a fixed probability p. We emphasize that using bursty loss models instead of this uniform loss model would not impact our results for Tornado code performance; only the overall loss rate is important. This is because when using Tornado codes we compute the entire encoding ahead of time and send out packets in a random order from the source end. Therefore, any loss pattern appears equivalent to a uniform loss pattern on the receiver end. Note that this randomization at the sender end may introduce latency, and therefore this Tornado code approach may not be appropriate for some applications such as real-time interactive video.

The choice of the uniform model does however impact the performance results of the interleaved codes, which (unless the same randomization of the transmission order is used) are highly dependent on the loss pattern. In particular, we would expect interleaved codes to have slightly better performance under bursty losses. We therefore also provide results from trace-driven simulations of the Internet to demonstrate the relatively small effect of burstiness on interleaved code performance.

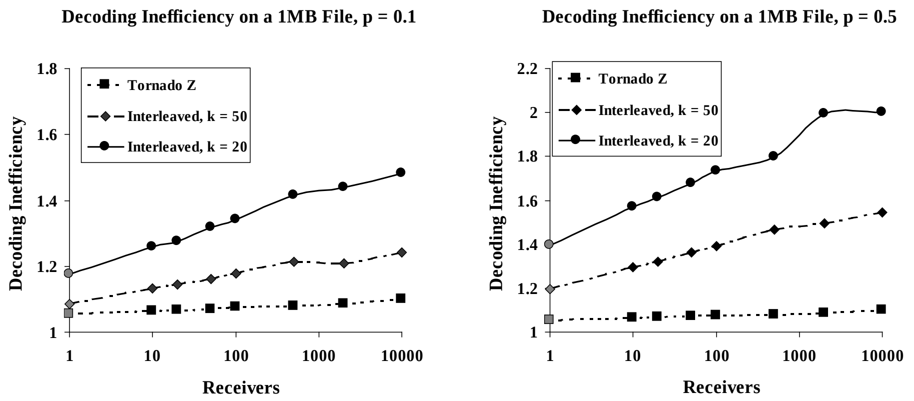

Figure 4: Comparison of decoding inefficiency for codes with comparable decoding times.

Speedup factor for Tornado Z					
	erasure probabilities				
SIZE	0.01	0.05	0.10	0.20	0.50
250 KB	1.37	2.05	5.55	11.1	11.1
500 KB	2.29	5.51	8.33	16.7	33.3
1 MB	4.12	10.3	17.1	25.8	51.6
2 MB	6.34	16.9	26.2	48.4	96.8
4 MB	7.87	22.3	34.6	62.7	115
8 MB	11.1	28.2	46.9	80	182
16 MB	14.2	34.9	56.4	100	212

Table 4: Speedup of Tornado Z codes over interleaved codes with comparable efficiency.

6.1 Equating Decoding Efficiency

Our first simulation compares the decoding time of Tornado Z with an interleaved code with decoding inefficiency comparable to those of Tornado Z. In Section 5, we determined experimentally that Tornado Z codes have the property that the decoding inefficiency is greater than 1.076 less than 1% of the time. In Table 4, we present the ratio between the running time of an interleaved code for which k is chosen so that this property is also realized and the running time of Tornado Z. Of course, this ratio changes as the loss probability and file size change.

We explain how the entries in Table 4 are derived. To compute the running time for interleaved codes, we first use simulations to determine for each loss probability value the maximum number of blocks the source data can be split into while still maintaining a decoding inefficiency less than 1.076 for less than 1% of the time. (For example, a two megabyte file consisting of 2000 one kilobyte packets can be split into at most eleven blocks while maintaining this property when packets are lost with probability 0.10.) We then calculate the decoding time per block, and multiply by the number of blocks to obtain the decoding time for the interleaved code. With a stretch factor of two, one half of all packets injected into the system are redundant encoding packets and the other half are source data packets. Therefore, in computing the decoding time per block, we assume that half the packets received are redundant encoding packets. Based on the data previously presented in the Cauchy codes column of Table 3, we approximate the decoding time for a block of k source data packets by $k^2/31250$ seconds. To compute the running time for Tornado Z, we simply use the decode times for Tornado Z as given earlier in Table 3.

As an example, suppose the encoding of a 16 MB file is transmitted over a 1 Mbit/second channel with a loss rate of 50%. It takes just over 4 minutes to receive enough packets to decode the file using either Tornado Z or an interleaved code (with the desired decoding inefficiency guarantee), but then the decoding time is almost 8 minutes for the interleaved code compared with just over 2 seconds for Tornado Z. Comparisons for encoding times yield similar results. We note that by using slightly slower Tornado codes with less decoding inefficiency, we would actually obtain even better speedup results at high loss rates. This is because interleaved codes would be harder pressed to match stronger decoding guarantees.

6.2 Equating Decoding Time

Our second set of simulations examines interleaved codes that have comparable decoding times to Tornado Z. Cauchy codes with block length $k = 20$ are roughly equivalent in speed to the Tornado Z code. We also compare with a block length $k = 50$, which is slower but still reasonable in practice.

Using these block sizes, we now study the maximum decoding inefficiency observed as we scale to a large number of receivers. The sender carousels through a two megabyte encoding of a one megabyte file, while receivers asynchronously attempt to download it. We simulate results for the case in which packets are lost independently and uniformly at random at each receiver at rates of 10% and 50%. The 10% loss rates are representative of congested Internet connections, while the 50% loss rates are near the upper limits of what a mobile receiver with poor connectivity might reasonably experience. The results we give can be interpolated to provide intuition for performance at intermediate rates of loss. For channels with very low loss rates, such as the 1% loss rates studied in [20], interleaved codes and Tornado have generally comparable performance.

Figure 4 shows for different numbers of receivers the worst case decoding efficiency experienced for any of the receivers averaged over 100 trials. In these figures, p refers to the probability a packet is lost at each receiver. Since the leftmost point in each subfigure is for the case of one receiver, this point is also just the average decoding inefficiency. The interesting feature of this figure is how the

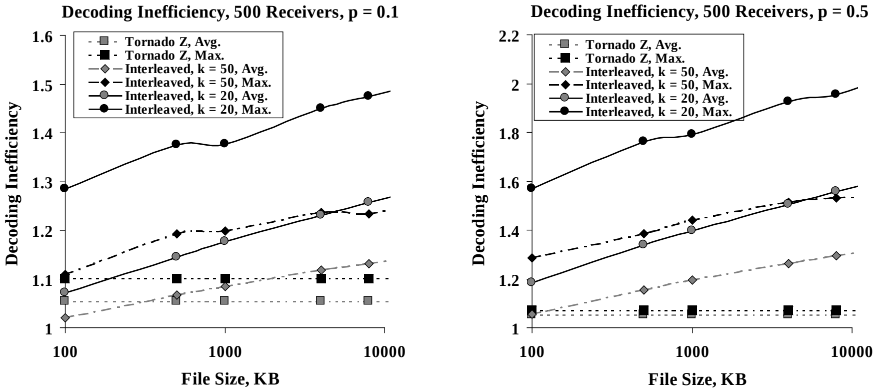

Figure 5: Comparison of decoding inefficiency as file size grows.

worst case decoding inefficiency grows with the number of receivers.

For packet loss rates of 10% and a block size of $k = 50$, the average inefficiency of interleaved codes is comparable to that of Tornado Z. But as packet loss rates increase, or if a smaller block size is used, the inefficiency of interleaved codes rises dramatically. Also, the inefficiency of the worst-case receiver does not scale with interleaved codes as the receiver size grows large. Tornado codes exhibit more robust scalability and better tolerance for high loss rates.

6.3 Scaling to Large Files

Our next experiments demonstrate that Tornado codes also scale better than an interleaved approach as the file size grows large. This is due to the fact that the number of packets a client must receive to reconstruct the source data when using interleaving grows super-linearly in the size of the source data. (This is the well-known "coupon collector's problem.") In contrast, the number of packets the receivers require to reconstruct the source data using Tornado codes grows linearly in the size of the source data, and in particular the decoding inefficiency does not increase as the file size increases.

The effect of this difference is easily seen in Figure 5. In this case both the average decoding inefficiency and the maximum decoding inefficiency grow with the length of the file when using the interleaving. This effect is completely avoided by using Tornado codes.

6.4 Trace-Driven Simulations

To study the impact of bursty loss patterns on the relative performance of Reed-Solomon and Tornado code approaches, we perform a similar comparison using publicly available MBone trace data collected by Yajnik, Kurose, and Towsley [26]. In these traces, between six and twenty clients from the US and abroad subscribed to MBone broadcasts each of roughly an hour in length and reported which packets they received. Clients experienced packet loss rates ranging from less than 1% to well over 20% over the course of these broadcasts.

To sample loss patterns from these traces, we simply chose a random starting point for each broadcast, and then used the trace data to generate packet loss patterns for each receiver in the broadcast beginning at that time. We then simulated downloading files of various lengths using interleaving and using Tornado codes with these loss patterns.

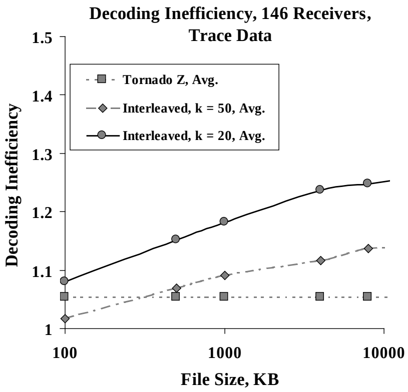

Figure 6: Comparison of decoding inefficiency on trace data.

Averaging over 146 loss patterns generated from 15 broadcasts, we plot the average decoding inefficiency for various file sizes in Figure 6.

The average loss rate over the randomly chosen trace segments we selected was just over 11%. In this trace data, there was considerable variance in the loss rate; some clients received virtually every packet, others experienced large burst losses over significant periods of time. While this trace data is limited in scope, the Tornado codes maintain superior decoding inefficiency in the presence of high burst losses present in this data set. In fact, the results appear very similar to that in Figure 5 when $p = 0.1$, suggesting that the bursty loss pattern has only a small effect.

7 Implementation of a Reliable Distribution Protocol using Tornado Codes

In this section, we describe an experimental system for distributing bulk data to a large number of heterogeneous receivers who may access the data asynchronously. Our implementation is designed for the Internet using a protocol built on top of IP Multicast. We have drawn on existing techniques to handle receiver heterogeneity and congestion control using layered multicast [15, 18, 25]. We emphasize that the purpose of developing this system is to demonstrate the feasibility of using Tornado codes in actual systems, and not to create a completely functional multicast protocol for deployment.

We expect that Tornado codes will also prove useful in other environments besides the Internet, such as satellite or wireless based systems. In these settings, different channel characteristics would suggest different approaches for congestion control and tolerating receiver heterogeneity. The Tornado code approach to bulk data distribution we apply, however, would remain essentially the same, even under varying end-to-end bandwidths and packet loss rates.

We first describe the design of our multicast protocol. The two main issues are the use of layered multicast and the approach the client uses to decode the message. Then we describe the experimental setup and performance results of our system.

7.1 Layering Across Multiple Multicast Groups

The congestion control approach we employ follows the lead of other authors who advocate *layered* multicast [15, 18, 25]. The main idea underlying this approach is to enable the source to transmit data across multiple multicast groups, thereby allowing the receivers to subscribe to an appropriate subset of these layers. Of course, practical considerations warrant keeping the number of multicast groups associated with a given source to a minimum. A receiver's subscription level is based on factors such as the width of its bottleneck link to the source and network congestion. The basic ideas common to the proposed layered schemes are:

- The server transmits data over multiple layers, where the layers are ordered by increasing transmission rate.
- The layers are *cumulative* in that a receiver subscribing to layer i also subscribes to all layers beneath it. We say that a receiver subscribes to *level* i when it subscribes to layers 0 through i.

For example, in our implementation, we use geometrically increasing transmission rates: $B_i = 2^{i-1}$ is the rate of the ith layer. Thus, a receiver at subscription level i would receive bandwidth proportional to $2B_i$, for $i \geq 1$. The protocol we use is based on the scheme described in recent work of Vicisano, Rizzo and Crowcroft [25] that proposes the following two novel ideas, summarized here briefly:

- Congestion control is achieved by the use of *synchronization points* (SP's) that are specially marked packets in the stream. A receiver can attempt to join a higher layer only immediately after an SP, and keeps track of the history of events only from the last SP. The rate at which SP's are sent in a stream is inversely proportional to the bandwidth: lower bandwidth receivers are given more frequent opportunities to move up to higher levels.
- Instead of explicit join attempts by clients, the server generates periodic *bursts* during which packets are sent at twice the normal rate on each layer. This has the effect of creating network congestion conditions similar to those that receivers would experience following an explicit join. Receivers use a packet loss event as an indication of congestion. So if a receiver witnesses no packet losses during the burst, it adds a layer at the next SP. Conversely, receivers drop to a lower subscription level whenever a packet loss occurs outside of a burst preceding a synchronization point.

Both the sending of SP's and burst periods are driven by the *sender*, with the receivers reacting appropriately. The attractive features of this approach are that receivers do not need to provide congestion control feedback to the source and receivers need not coordinate join attempts to prevent disruption to other receivers. These features are particularly important in the context of a digital fountain in which receiver-to-source and inter-receiver communication are undesirable. Moreover, the work of [25] demonstrates how to set transmission rates and the interarrival time between SP's so that the resulting congestion control policy is TCP-friendly, and shares bandwidth in a comparable way to point-to-point TCP connections. We refer the reader to [25] for further details.

7.2 Scheduling Packet Transmissions Across Multiple Multicast Groups

As described earlier, a receiver at level i subscribes to *all* layers 0 through i. Therefore, it is important to schedule packet transmissions carefully across the multiple layers, so as to minimize the number of duplicate packets that a client receives. The stretch factor c limits the number of distinct packets that can be transmitted, and therefore also has a strong effect on the number of duplicates a client receives, especially in the presence of high packet loss rates. Of course, using a large stretch factor provides more flexibility, but it slows decoding time and increases the space requirements for decoding.[3] For these reasons, we typically choose a stretch factor $c = 2$ as compared to $c = 8$ used in [23, 24], although using larger stretch factors with Tornado codes is certainly feasible. We find that this choice is suitable in practice because we use a packet transmission scheme that has the following property:

One Level Property: If a receiver remains at a fixed subscription level throughout the transmission and packet loss remains sufficiently low, it can reconstruct the source data before receiving any duplicate transmissions. Specifically, if the loss rate is below below $1 - \frac{1+\epsilon}{c}$, where $1+\epsilon$ is the reception inefficiency of the Tornado code, then in one cycle of ck encoding packets a receiver obtains the $(1+\epsilon)k$ packets necessary to decode.

Recently, Bhattacharyya et al. show that a general transmission scheme exists that realizes the one level property for any arbitrary set of layered transmission rates [3]. For example, Table 5 demonstrates a simple sending pattern for the rate organization previously described with 4 layers, 4 source data packets, and a stretch factor of 2.

Our sending pattern satisfies the One Level Property. In fact, the sender transmits a permutation of the entire encoding both to each multicast layer and to each cumulative subscription level before repeating a packet. Receivers that change their subscription level over time, however, do not witness this ideal behavior. While we show in Section 7.4 that the reception inefficiency remains low even when receiver subscription levels change frequently, optimizing properties of the schedule further for this scenario remains an open question.

7.3 Reconstruction at the Client

As detailed in the previous subsection, the client is responsible for observing SPs and modifying its subscription level as congestion warrants. The other activity that the client must perform is the reconstruction of the source data. There are two ways to implement the client decoding protocol. The

[3] The memory required for decoding Tornado codes is proportional to the length of the encoding, not to the size of the source data.

Layer	Bandwidth per Round	Packets sent during							
		Rd 1	Rd 2	Rd 3	Rd 4	Rd 5	Rd 6	Rd 7	Rd 8
3	4	0-3	4-7	0-3	4-7	0-3	4-7	0-3	4-7
2	2	4-5	0-1	6-7	2-3	4-5	0-1	6-7	2-3
1	1	6	2	4	0	7	3	5	1
0	1	7	3	5	1	6	2	4	0

Table 5: Packet transmission scheme for 4 layers

first is an incremental approach in which the client performs preliminary decoding operations after each packet arrives. This approach leads to some redundant computation: reconstructed source data may later arrive intact. Moreover, there may be substantial overhead in processing individual packets immediately on arrival. A second, patient approach that reduces these effects is to wait until a fixed number of packets arrive from which it is likely that the source can be reconstructed, based on statistical observations. If the decoding cannot be completed at this time, then additional packets may be processed individually or in small groups. While the incremental approach has the benefit of enabling some decoding computation to be overlapped with packet reception, we found the patient approach to be simpler to implement in practice, with little loss of decoding speed. In our final implementation we wait until $1.055k$ packets arrive, attempt to decode, and then process additional packets individually as needed until decoding is successful.

7.4 Experimental Setup and Results

Now we turn to measurements of the efficiency of our experimental system. First, we clarify the two sources of inefficiency. Recall that the *decoding inefficiency*, $1+\epsilon=\eta_c$, captures the inefficiency due specifically to the use of Tornado codes. It is defined as

$$\eta_c = \frac{\text{\# of distinct packets received prior to reconstruction}}{\text{\# of source data packets}}.$$

There is, however, another possible source of inefficiency: a receiver could obtain duplicate packets. The *distinctness inefficiency*, η_d, captures the loss in efficiency caused by receiving duplicate packets. This can occur either by cycling through the carousel under exceedingly high loss rates or by changing the receiver subscription layer as described in Section 7.2. It is defined as

$$\eta_d = \frac{\text{Total \# of packets received}}{\text{\# of distinct packets received}}.$$

Combining these two effects yields the *reception inefficiency*, η. It is defined as

$$\eta = \frac{\text{Total \# of packets received prior to reconstruction}}{\text{\# of source data packets}}.$$

It is clear that $\eta = \eta_c\eta_d$.

The experimental results measure our prototype implementation. Besides testing the layered protocol we have described, we also test a single layer protocol. That is, we also measure the reception inefficiency when the server transmits the file on a single multicast group at a fixed rate. These results allow us to focus on the efficiency of the packet transmission scheme independent of the layering scheme for congestion control. The server runs two threads: a UDP unicast thread that provides various control information such as multicast group information and file length to the client and a multicast transmission thread. The clients for both protocols connect to the server's known UDP port for control information and on receipt of the information, subscribe to the appropriate multicast groups.

Our test source data consisted of a Quicktime movie (a clip available from `www.nfl.com`) with size slightly over two megabytes. The encoding algorithm used a stretch factor of $c=2$ to produce 8264 packets of size 500 bytes. The packets were additionally tagged with 12 bytes of information (packet index, serial number and group number) to give a final packet size of 512 bytes. The server and clients were on three different subnets, located at Berkeley, CMU and Cornell. There were 16 hops on the path from Berkeley to CMU, and the bottleneck bandwidth (obtained by using *mtrace* and *pathchar* [9]) was 8 Mb/s with an RTT of 60 ms. There were 17 hops on the path from Berkeley to Cornell, and the bottleneck bandwidth was 9.3 Mb/s with an RTT of 87 ms. Base layer bandwidth was set at rates ranging from 64 Kb/sec to 512 Kb/sec. We ran experiments with the server both at Berkeley and at CMU and with the clients located at the other two subnets. Locating the server at CMU tended to generate higher packet loss rates for the same transmission bandwidth. The machines used at CMU and Berkeley were 167 MHz UltraSPARC-1's running Solaris 2.5.1. The client at Cornell ran on a 60 MHz Sparc. When running the layered protocol, we used 4 layers.

In our initial experiments, in some cases we witnessed loss rates over the course of the transmission of over 20% – rates that are admittedly far higher than the congestion control techniques of [25] were intended to handle. To generate even higher loss rates that might arise in other environments, such as mobile wireless networks, the base layer rate was set artificially high, causing a router along the path to drop packets persistently.

The data from the two sets of experiments are shown in Figure 7. As seen from the graphs for the single layered case, for packet losses of less than 50% , the distinctness inefficiency is almost always 1. This is to be expected because of the One Level Property.[4] Thus, for low loss rates, the reception inefficiency is effectively the decoding inefficiency, which in our example was roughly 1.07 on average. (This decoding inefficiency is slightly higher than for Tornado Z because a different code was used in these experiments, and because we wait until at least $1.055k$ packets arrive before trying to decode.) We further observe that the transmission scheme is robust even under severe loss rates - at nearly a 70% loss rate, the reception inefficiency is generally below 1.4. Of course, if one had reason to suspect such excessively high

[4] Note that it is possible to have a *cumulative* loss rate that is less than 50% but in which losses initially are higher than 50% in the first cycle so that the client receives some duplicates. This is precisely what happened for the outlying point at 35% packet loss in the single layer distinctness inefficiency graph.

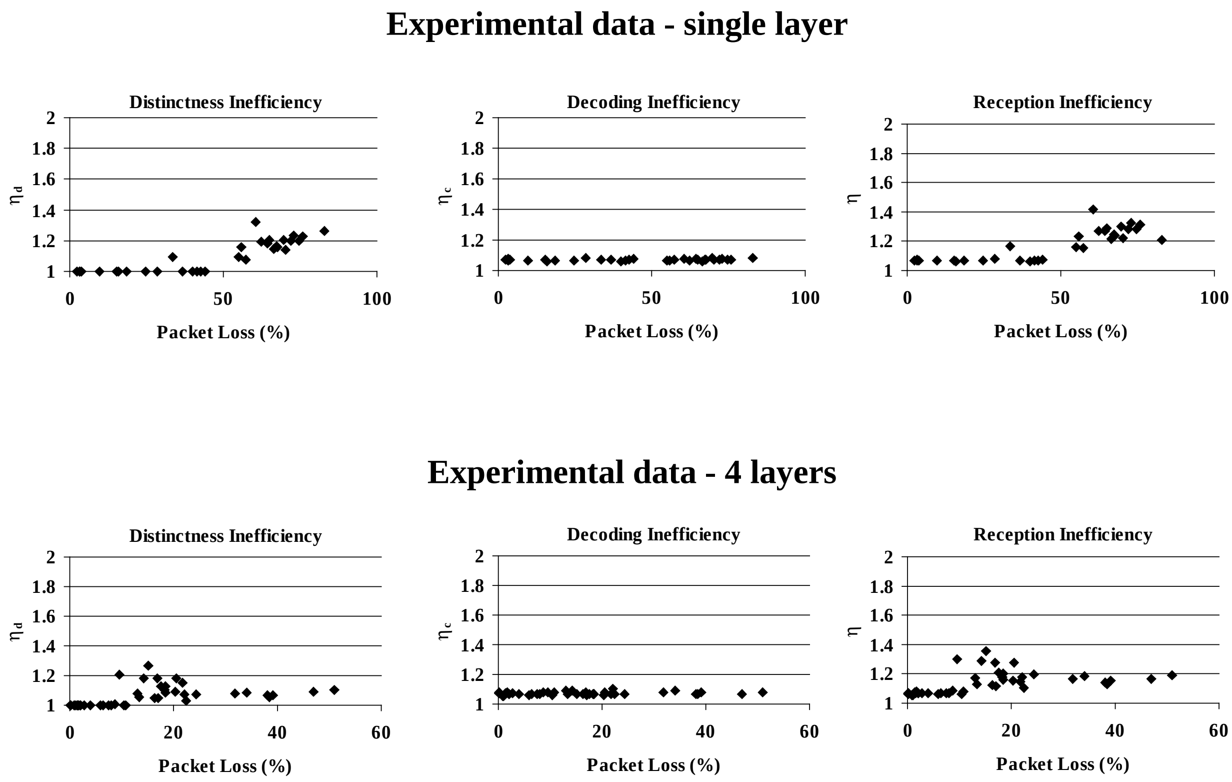

Figure 7: Experimental Results of the Prototype

loss rates ahead of time, one could choose a larger stretch factor, at the expense of proportionally higher encoding and decoding times.

Figure 7 also shows experimental data for the multilayered case. We observe that the use of multiple layers for congestion control increases the distinctness inefficiency. This is natural as switching among subscription levels can cause the client to receive packets that had already been obtained at other subscription levels. For high loss rates, the distinctness inefficiency remained low because receivers generally subscribed only to the base layer. An interesting direction we intend to pursue further is to study how the reception efficiency varies with the rates of change in receiver subscription level.

8 Conclusion

The introduction of Tornado codes yields significant new possibilities for the design of reliable multicast protocols. To explore these possibilities, we formalized the notion of an ideal digital fountain and explained how Tornado codes can yield a much closer approximation to a digital fountain than previous systems based on standard Reed-Solomon erasure codes. Our prototype multicast data distribution system demonstrates that simple protocols using Tornado codes are effective in practice. It would be useful to test a similar system with a large number of users to fully demonstrate the effectiveness of our approach.

Given that we can closely approximate a digital fountain with Tornado codes, we conclude with other possible applications for such an encoding scheme. One application is dispersity routing of data from endpoint to endpoint in a packet-routing network. With packets generated by a digital fountain, the source can inject packets along multiple paths in the network. Those packets that experience congestion are delayed, but the destination can recover the data once a sufficient number of packets arrive, irrespective of the paths they took. This application dates back to the seminal works on dispersity routing by Maxemchuk [13, 14] and information dispersal by Rabin [21]. Both suggested using standard erasure codes. We expect Tornado codes will lead to improved practical dispersity routing schemes.

Another application for which the Tornado code approximation might be useful arises in the context of mirrored data. Currently, to minimize response time, clients search for a lightly loaded mirror site on an uncongested path. If the sources use ideal digital fountains to transmit the data, clients can access multiple sources simultaneously, and aggregate *all* the packets they receive to recover the data efficiently. The problem with a Tornado code solution is that if the stretch factor is small, one receives duplicate packets frequently; if the stretch factor is large, the space and time requirements for decoding become prohibitive. We are currently studying how parameters may be set appropriately to yield a viable solution.

References

[1] S. Acharya, M. Franklin, and S. Zdonik, "Dissemination Based Data Delivery Using Broadcast Disks," *IEEE Personal Communications*, December 1995, pp. 50-60.

[2] A. Bestavros. "AIDA-based real-time fault-tolerant broadcast disks." In *Procedings of the 16th IEEE Real-Time System Symposium,* 1996.

[3] S. Bhattacharyya, J. F. Kurose, D. Towsley, and R. Nagarajan, "Efficient Rate-Controlled Bulk Data Transfer using Multiple Multicast Groups", In *Proc. of INFOCOM '98*, San Francisco, April 1998.

[4] J. Blömer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman, "An XOR-Based Erasure-Resilient Coding Scheme," *ICSI Technical Report No. TR-95-048*, August 1995.

[5] S. Floyd, V. Jacobson, C. G. Liu, S. McCanne, and L. Zhang, "A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing." In *ACM SIGCOMM '95*, pp. 342-356, August 1995.

[6] J. Gemmell, "ECSRM – Erasure Correcting Scalable Reliable Multicast," *Microsoft Research Technical Report MS-TR-97-20*, June 1997.

[7] C. Huitema, "The Case for Packet Level FEC." In *Proc. of IFIP 5th Int'l Workshop on Protocols for High Speed Networks,* Sophia Antipolis, France, October 1996.

[8] Cauchy-based Reed-Solomon codes. Available at `http://www.icsi.berkeley.edu/~luby`.

[9] V. Jacobson, "pathchar", available at `http://www-nrg.ee.lbl.gov/pathchar`.

[10] J. C. Lin and S. Paul, "RMTP: A Reliable Multicast Transport Protocol." In *IEEE INFOCOM '96*, pp. 1414-1424, March 1996.

[11] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann, "Practical Loss-Resilient Codes." In *Proceedings of the* 29^{th} *ACM Symposium on Theory of Computing*, 1997.

[12] M. Luby, M. Mitzenmacher, and A. Shokrollahi, "Analysis of Random Processes via And-Or Tree Evaluation." In *Proceedings of the* 9^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1998.

[13] N. F. Maxemchuk, **Dispersity Routing in Store and Forward Networks**. Ph. D. thesis, University of Pennsylvania, May 1975.

[14] N. F. Maxemchuk, "Dispersity Routing." Proceedings of ICC '75, San Francisco, CA, pp. 41-10 – 41-13, 1975.

[15] S. McCanne, V. Jacobson, and M. Vetterli, "Receiver-driven Layered Multicast." In *Proc. of ACM SIGCOMM '96*, pp. 117-130, 1996.

[16] C. K. Miller, "Reliable Multicast Protocols: A Practical View." In *Proc. of the 22nd Annual Conference on Local Computer Networks (LCN '97)*, 1997.

[17] J. Nonnenmacher and E. W. Biersack, "Reliable Multicast: Where to Use Forward Error Correction." In *Proc. of IFIP 5th Int'l Workshop on Protocols for High Speed Networks,* pp. 134-148, Sophia Antipolis, France, October 1996. Chapman and Hall.

[18] J. Nonnenmacher and E.W. Biersack, "Asynchronous Multicast Push: AMP." In *Proc. of International Conference on Computer Communications*, Cannes, France, November 1997.

[19] J. Nonnenmacher, M. Lacher, M. Jung, G. Carl, and E.W. Biersack, "How Bad is Reliable Multicast Without Local Recovery?" In *Proc. of INFOCOM '98*, San Francisco, April 1998.

[20] J. Nonnenmacher, E. W. Biersack, and D. Towsley, "Parity-Based Loss Recovery for Reliable Multicast Transmission." In *Proc. of ACM SIGCOMM '97*, 1997.

[21] M. O. Rabin, "Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance." In *Journal of the ACM*, Volume 38, pp. 335-348, 1989.

[22] L. Rizzo, "Effective Erasure Codes for Reliable Computer Communication Protocols." In *Computer Communication Review*, April 1997.

[23] L. Rizzo and L. Vicisano, "A Reliable Multicast data Distribution Protocol Based on Software FEC Techniques." In *Proc. of HPCS '97*, Greece, June 1997.

[24] E. Schooler and J. Gemmell, "Using multicast FEC to solve the midnight madness problem," *Microsoft Research Technical Report MS-TR-97-25*, September 1997.

[25] L. Vicisano, L. Rizzo, and J. Crowcroft. "TCP-like congestion control for layered multicast data transfer." In *Proc. of INFOCOM '98*, San Francisco, April 1998.

[26] M. Yajnik, J. Kurose, and D. Towsley, "Packet Loss Correlation in the MBone Multicast Network." In *Proceedings of IEEE Global Internet '96,* London, November 1996.

[27] R. Yavatkar, J. Griffoen and M. Sudan, "A Reliable Dissemination Protocol for Interactive Collaborative Applications." In *Proceedings of ACM Multimedia '95,* San Francisco, 1995, pp. 333-344.