Efficient Reliability in Volunteer Storage Systems with Random Linear Coding

Ádám Visegrádi, Péter Kacsuk

Computer and Automation Research Institute, Hungarian Academy of Sciences, Hungary
{visegradi.adam,kacsuk.peter}@sztaki.mta.hu

Abstract. Volunteer systems pose difficult challenges for data storage. Because of the extremely low reliability of volunteer nodes, these systems require so high redundancy that replication is infeasible. Erasure coding has been proposed to cope with this problem as it needs much less redundancy to achieve the same reliability. Its downside is that the reparation of the system creates high overhead, as fully decoding the original data is required to generate new coded data.
Random linear coding has been proposed to be used as a data storage method, as it provides a better redundancy/reliability ratio, and less control overhead. We propose that it also helps in the reparation of the system, as decoding is not required; instead, coded data can be generated from already existing coded data. However, it may be possible that this iterative reparation leads to degradation of data over time; even more so, if sparse coding is used to increase compute efficiency.
This paper examines the effects of random linear coding and the iterative reparation of the system. It shows the reliability that can be achieved with random linear coding in a highly volatile distributed system. We conclude that random linear coding can achieve high reliability even in highly volatile systems.

1 Introduction

Volunteer and community compute systems use the donated compute capacity of people or organisations. This scheme provides immense computational power for extremely low cost. These volunteers can also contribute storage space to the system; however, this poses difficult architectural challenges.

The main problem is that the nodes of the system are highly unreliable, and that no policy can be enforced on these nodes (cf. grids or clusters). This is not a concern for computation, as any task can be restarted on another node if fails. In data storage however, this unreliability necessitates high redundancy and, therefore, high storage overhead.

More reliable systems, like Hadoop[9] and MapReduce[10] have proven that using cheap commodity hardware is a feasible alternative to expensive RAID storages, but even these systems may require high redundancy overhead to store data reliably. The BOINC[7] volunteer compute middleware is used widely to

support compute intensive applications for the fraction of the cost of owning a cluster, and it would have ample resources for data intensive applications.[8] However, BOINC does not support data intensive applications well. Although it has support for raw block storage[6], and a data archival solution is being developed[5], it yet lacks a working distributed storage. In any case, the distributed storage system will require redundancy, which decreases the effective storage capacity.

1.1 Achieving Redundancy

The basic way to introduce redundancy is to *replicate* the blocks of the data. Although it is the simplest solution, this approach has major drawbacks. First, the raw storage space required to store some data is the (integer) multiple of the size of the data; and this factor of redundancy has to be very high if the nodes' reliability is low. Second, as failing nodes remove replicas of a block of the data from the system, that block may become rare, impairing locality.

Erasure coding (EC) is an alternative to replication that alleviates both problems of replication, at the cost of CPU time. Erasure coding algorithms—e.g. Reed–Solomon[18] or Fountain codes[17]—create $n > k$ coded blocks from the original data in a way that *any* $k' >= k$ coded blocks will be sufficient for reconstruction. That is, they are block codes with a coding rate (n, k). Because *any* k' block is sufficient for reconstruction, a much lower factor of redundancy is sufficient than in case of replication[21].

Furthermore, in erasure coding, the factor of redundancy does not have to be an integer. Although this seems to be a trifle, but, for instance, the difference between a theoretically required redundancy—given a set of QoS parameters—of 2.1 (erasure coding) and $\lceil 2.1 \rceil = 3$ (replication) can be substantial when there are peta-bytes of data.

The problem with erasure coding is the design complexity of the system, and the overhead required for its reparation[19]. In both cases (replication and EC), when blocks go missing, they have to be complemented. In case of replication, only rare blocks have to be further replicated. In case of erasure coding, to complement the missing blocks, the whole data has to be reconstructed, so new coded blocks can be generated from it[19][12]. Dimakis et al. propose regenerating codes[11] to remedy this problem. Regenerating codes use network coding[4] to communicate encoded packets, which enables the reparation of redundancy without reconstructing the original data. However, if the data is very dispersed and each node stores only one piece of a data object, this approach may not provide many benefits. This can happen in large volunteer systems, which are of particular interest to us. Furthermore, if a deterministic erasure coding scheme is used, then each block will have its own identity, and when a block goes missing, *that* particular block must be complemented. This requires each individual block to be identifiable, which imposes managment overhead on the system.

A promising approach is to use *random linear coding* (RLC) to store the data. Linear coding treats the blocks of data as vectors—and the data itself as a matrix—over a finite field $\mathbb{F}(2^w)$ $(w \geq 1)$. Coding is performed by creating

linear combinations of the original blocks, while decoding is done by solving the corresponding linear system. Linear coding has been well studied in the area of networking, as an alternative to routing[15, 13]; and random network coding has been proposed as a simple solution for finding suitable coefficients for linear combinations[14].

Random linear coding can be considered to be a *rateless* erasure coding method, as $n > k$ randomly encoded packets can be generated from the original file for any n (rateless), of which any $k' >= k$ will be sufficient for reconstruction (erasure coding). Furthermore, it is stochastically optimal, and it converges to optimal with increasing field size[14]; that is, $P[k' = k] \longrightarrow 1(q \longrightarrow \infty)$.

As a rateless erasure coding, RLC could solve the problems of replication; it would even perform better in terms of redundancy than traditional erasure coding schemes [3]. Also, the problem of reparation in a RLC system would become quite straight-forward: as stored blocks are random linear combinations (r.l.c.) of the original data blocks, a r.l.c. of the *coded* blocks will *also* be a r.l.c. of the *original* blocks. That is, reparation can be done by randomly selecting existing coded blocks from the system, and creating r.l.c.-s of them—we call this *iterative reparation*. With iterative reparation no reconstruction is needed to generate coded data. These properties make RLC a great candidate for coding data in distributed storage scenarios.

A drawback of RLC is that it's CPU intensive. The more blocks we cut the data into (as we increase n), the more reliable the RLC scheme becomes—but increasing n also increases decoding time (linearly). Intuitively, coding ,,trades" redundancy requirements for CPU requirements. Although this may be problematic in HPC, we have shown[2] that RLC can provide reasonable throughput with the right parameter set.

Furthermore, storage media have considerable delay, particularly when seeking. Therefore, it is but reasonable to perform coding and decoding solely in working memory, which limits the file size. The solution for this is to cut data into *segments* that are coded and encoded independently. This enables us to store arbitrarily large files using RLC.

While a given RLC scheme provides a predictable throughput, segmenting enables us to control the initial delay (decoding the first segment) when decoding successive segments in a stream. Seeking in the stream is also possible, but with the time cost of the initial delay.

In this paper, *file* and *data* refer to such a segment.

1.2 Reliability of a System Using Random Linear Coding

The most interesting question about RLC is what reliability/redundancy ratio can it achieve when the nodes of the underlying system are extremely unreliable; e.g. when 50-90% of the blocks fail between maintenance events.

This question is even more interesting when *sparse coding* is used. In sparse coding, the coefficients in the coding matrix will be set to 0 with a given probability. This alleviates the CPU overhead of RLC (as multiplications with 0 can be omitted), but decreases reliability (greater probability of singular coefficient

matrices). Therefore, we hypothesize that the reliability of the system would decrease when the iterative reparation is used with sparse coding because of gradual information loss.

These questions has been addressed in network transfer scenarios[20, 16]. In this paper we present our simulational results showing how RLC performs under specific conditions we consider to represent volunteer storage scenarios.

2 Simulation Framework

2.1 Storage Scheme

We have conceived the following model of a RLC distributed storage system. The system itself is considered as a bag of coded blocks. When storing a file, it is loaded into memory, cut into N blocks, and R coded blocks are generated from it. The coded blocks are generated using linear combinations; each linear combination is a result of multiplying the set of original blocks with a randomly chosen vector of N elements. As sparse coding is used, each coefficient is set to 0 with probability $(1 - A)$; that is, about $A \cdot N$ coefficients will be non-zero.

The system has to be maintained, which means that the coded blocks are *replenished* from time to time. Between replenishing blocks, each block may fail with probability F. That is, our system is a simple iteration of replenishing blocks, where about $100 \cdot F\%$ percent of the blocks fail between iterations. Replenishing blocks takes place only when the number of blocks fall under a certain threshold T, and when it does, so many blocks are generated that the total number of blocks becomes again R. The special case is when $T = R$, that is, when a certain level of redundancy is maintained.

The iteration described is shown in Figure 1; the catalog of parameters is shown in Table 1.

$N \in \{4, 8, ..., 128\}$	The number of blocks the file is cut into.
$A \in \{0.1, 0.2, ..., 1\}$	Probability of a coefficient is selected randomly from $\mathbb{F}\left(2^{16}\right)$. *Otherwise*, it is set to 0.
$F \in \{0, 0.1, ..., 0.9\}$	Probability of a block failing between maintenance events.
$T \in \left\{N, \frac{3}{2}N, 2N, 3N, 4N\right\}$	Threshold for replenishing blocks.
$R \in \left\{N, \frac{3}{2}N, 2N, 3N, 4N\right\}; R > T$	Target redundancy.

Table 1: Parameters of the simulation

Reconstruction and replenishment requires at least N coded blocks to be gathered, as any less than that would produce an under-determined rectangular matrix, which cannot be inverted. On the other hand, RLC is stochastically optimal; that is, with high probability, N coded blocks will be sufficient. Depending on the parameters, it is more or less likely to gather N blocks whose coefficient

matrix is singular. In this case, it is not necessary to gather another N blocks. If there is a single vector that is a linear combination of the others, it can be exchanged with a new, randomly selected block. However, finding the offending vector(s) is computationally demanding and is not necessary. The simplest solution is to drop either a randomly selected block, or the one where the Gaussian elimination has failed, and then complement it with one randomly selected from the system. After this change, the probability of the matrix being still singular is even lower.

Therefore, in the unfortunate case when a singular matrix is found, instead of trying every possible combination of vectors (which is infeasible in a distributed system), we try dropping a block from the gathered set and complement it with a random one from the system, never using a block twice. We repeat this process until either we have found a non-singular matrix or we have tried all blocks with no avail. This means an $\omega(N)$–$O(R)$ communication cost, but the upper bound may further be restricted in a particular implementation if necessary. It should also be noted that this overhead is very low, as these operations—that is, the matrix inversion—require transferring only the coefficient vectors, but not the corresponding data blocks.

This strategy is shown in Figure 2.

```
   Input: file
   Output: replenish_failed | ∅
 1 file_blocks := load_file(file);
 2 block_set := replenish(file_blocks, R);
 3 while not replenish_failed do
 4 |   block_set =
   |   remove_randomly(block_set, F);
 5 |   if length(block_set) < T then
 6 |   |   block_set =
   |   |   replenish(block_set, R);
 7 |   end
 8 end
```

Fig. 1: System maintenance cycle

```
   Input: block_set
   Output: working_set | failure
 1 candidates := block_set;
 2 for i = 1 to N do
 3 |   move_random(
 4 |       candidates → working_set);
 5 end
 6 while singular(working_set)
 7 |       and length(candidates) ≠ 0 do
 8 |   remove_random(working_set);
 9 |   move_random(
10 |       candidates → working_set);
11 end
12 if singular(working_set)
13 |   and length(candidates) = 0 then
14 |   failure := true;
15 end
```

Fig. 2: Strategy for gathering a usable block set

2.2 Measurements

We have used our RLC library[1] to implement a simulation framework. This library can perform finite field operations over $\mathbb{F}\left(2^8\right)$ and $\mathbb{F}\left(2^{16}\right)$ using discrete

logarithm tables. We have used the finite field $\mathbb{F}\left(2^{16}\right)$ as it is faster and produces a more reliable RLC scheme than $\mathbb{F}\left(2^{8}\right)$[2].

In our measurements, we have used one small file of 256 bytes (i.e. 128 finite field elements) with random content, which we cut into $N \in \{4, 8, ..., 128\}$ blocks. The reason to use a small file was that the actual size of the file does not affect its reliability whatsoever; it only affects the de/coding time.

We have performed two kinds of measurements. First, we have measured the reliability of RLC as a function of sparsity (A), without a time dimension (single step). And second, we have measured the reliability of a simulated system over time.

In the single step scenario, the following restrictions were applied to the model. As there is no time dimension, redundancy in the system does not fluctuate; therefore, the redundancy threshold (T) has no meaning, only the target redundancy (R). In this case, lines [3..8] in Figure 1 are not executed; instead, after the generating initial blocks, the data is immediately attempted to be reconstructed using the strategy shown in Figure 2. Also, the failure ratio (F) has no meaning in this case either.

In the simulational scenario, F is introduced in the system to model block failures. The case where $F = 0$ is not evaluated, as it would be identical to the single step scenario. When $F > 0$, the redundancy threshold *must* be greater than the number of blocks: if $N = T$ and $F > 0$, the system is bound to fail before the first maintenance event. Therefore, the $N = T$ case is not evaluated either.

In both cases, the goal was to measure the reliability of the system. Defining reliability as the general ability to reconstruct the original file is not feasible for either of these measurement—neither is in a real world scenario. Denoting the set of coded blocks available in the system with $BlkSet$, this general definition can be formulated as follows: $\exists S \subset BlkSet : |S| = N \wedge \det(\text{coeff\_matrix}(S)) \neq 0$; that is, there is *a* way to reconstruct the data. However, the size of $BlkSet$ can be huge, it is between T and R at any time, and trying all possible combinations ($\omega\binom{T}{N}$ and $O\binom{R}{N}$) is not feasible. Thus, we define the feasibility of a system as the ability to reconstruct the file using the strategy shown in Figure 2. We measure reliability as the fraction of test cases in which a) the file could be reconstructed from coded data – single step scenario; and b) the file could be reconstructed after several maintenance iterations – simulation scenario. Thus, we define **reliability** as the estimated probability of a file being reconstructable in the system using the aforementioned algorithm. In these measurements, the total number of iterations were limited to 100, and a simulation for each parameter set was repeated 50 times.

As it is possible that this strategy needs to use more than N blocks to reconstruct the data, we also measured **wasted communication**, which we define as the number of *extra* blocks needed to reconstruct the data: total_blocks_needed$-N$.

3 Simulation Results

In this section we present the results of our measurements. Most of the figures presented show the reliability of a system, based on a two-dimensional parameter set, while the reliability (as defined in 2.2) is a value between 0 and 1. These figures present this function in the following way: the two axes of the diagrams pertain to the two dimensions of the parameter space, while the value of reliability is represented as a shade of gray. The black regions (reliability=0) mean that for that parameter set, all experiments have failed, the data could not be reconstructed; while white regions (reliability=1) mean that, in those cases, RLC have not failed at all. As we will show, gray areas are surprisingly narrow, and therefore, we believe that this form of presentation can convey all essential information to the reader.

3.1 Single Step Scenario

This scenario is intended to measure the reliability of RLC itself, without a time dimension. In this scenario, F and T are meaningless. The file is cut into N blocks, R blocks are generated, from which the file is attempted to be reconstructed.

The results of these measurements are shown in Figure 3. The three charts show the results with redundancy factors 1 (no redundancy), 1.5 and 2 respectively. The black area shows the parameter sets where the file could not be reconstructed at all (infeasible parameters). The white area shows the experiments where the file could always be reconstructed (reliable parameters). What immediately meets the eye is that there is little or no gradient between reliable and unfeasible parameter sets (feasible but unreliable parameters). This implicates that if a parameter set is feasible, it is very likely to be reliable too.

Another observation is that for higher values of N, even extremely sparse, $A = 0.1$ coding is feasible. This means that each coefficient is set to 0 with 90% probability, which can theoretically decrease coding time to 1/10th of the non-sparse case.

As stated before, it is possible, that reconstructing a file requires the transfer of more than N blocks. Figure 4 shows the wasted communication cost (as defined in 2.2). Using parameters outside the dashed line (lower left corner), in no experiment were we able to reconstruct the file (infeasible parameters). As these parameters are infeasible, wasted communication is undefined. Inside the dashed line, the lightness of a point represents the average number of extra blocks transferred to reconstruct the file (in this figure, darker is better). Again, the transient area is very narrow, which means that either a parameter set is not feasible (no reconstruction is possible), or it likely generates negligible wasted communication.

3.2 Simulational Scenario

In this scenario we measured the reliability of the system over time, with maintenance events in each interval. We did 100 iterations, and recorded the instant

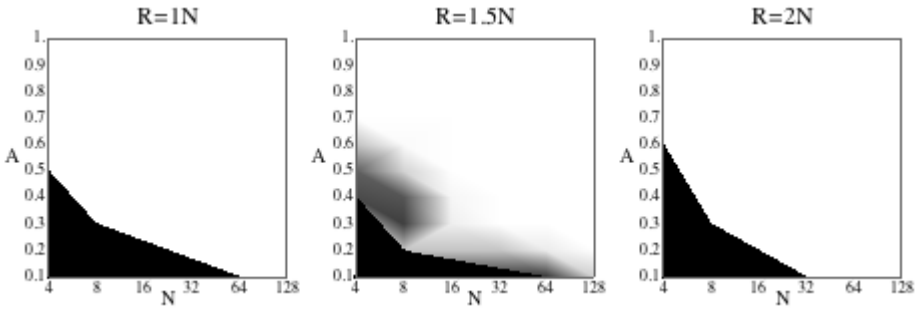

Fig. 3: Reliability as a function of sparsity (A) and the number of original blocks (N).

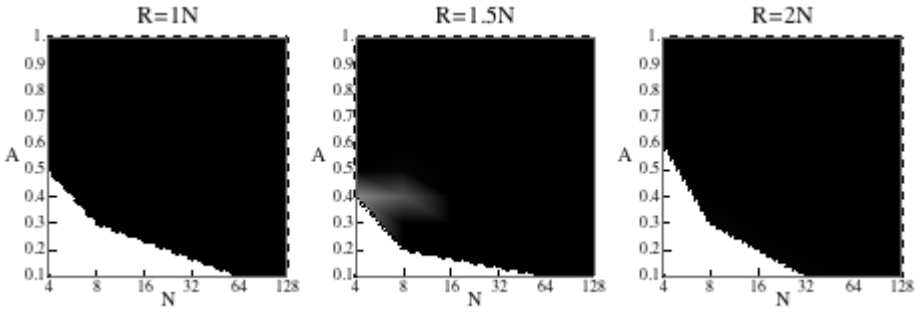

Fig. 4: Average number of wasted blocks as a function of sparsity (A) and the number of original blocks (N).

when the file became unavailable. We considered an experiment successful, if the data did not become unavailable in this time.

For experiments that were *not* successful, we have measured for how long the file was available; that is, we recorded the iteration in which the experiment has failed. In Figure 5, we show how many experiments have failed in that iteration overall. Each line corresponds to a specific value of F; the inner figure is a magnification of the outer one.

In the outer figure, we can see that almost all failed experiment have failed at the beginning. Of the failed experiments, 68% has failed in the first iteration, and more than 90% has failed in the first 15 iteration.

In the inner, magnified figure, we can see that for $F = 0.9$ and 0.8, failure is imminent at the beginning, while for lower values, failure is 1) less likely as time goes on, and 2) the actual value of F does not affect failure much (lines converge together).

In Figure 6 the reliability of the system is shown as a function of N and F, for different factors of redundancy and for two factors of sparsity. In these cases, we used the same values for maintenance threshold and target redundancy ($R = T$). The first row shows the results for $A = 1$, that is, when sparse coding

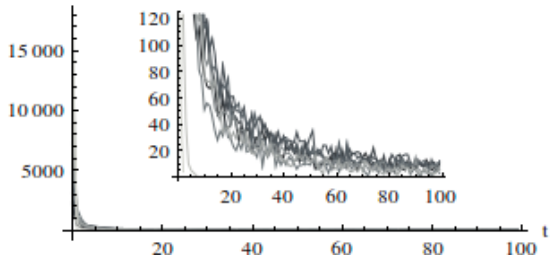

Fig. 5: Number of failed experiments in each iteration (t), for specific values of F.

is not used. The second row corresponds to $A = 0.5$. From left to right in a row, redundancy increases.

It can be seen that the two rows look very much alike, the only considerable difference being at the "corners" of the reliable sets (white area). This matches the observation of the single step case, that RLC can be very reliable even with low values for A—especially when N is high.

In terms of N, reliability increases with N up to a point, where it plateaus. This peak is reached later if A is lower. This means that to achieve reliability, there is a lower bound on N depending on the reliability of the system (F) and the value we choose for A.

It is also clear—and expected—that using higher factors of redundancy increases the reliability of the system, in that higher values of F can be tolerated (see the increasing plateaus of the white areas). We would like to note here that, for example, Hadoop uses a replication factor of 3 in cluster environments, while our measurements show that, with the same level of redundancy, a failure rate of 50% ($F = 0.5$) can be tolerated by RLC even with sparse coding. Using three-fold *replication*, losing 50% of blocks would very likely make the data unrecoverable.

Also note that although F depends on the properties of the particular system, it is not entirely out of our control. By increasing the frequency of maintenance events, we can decrease the value of F, and therefore we can trade off maintenance network cost for redundancy; that is, network overhead for storage overhead.

We have also examined a scenario where the reparation threshold T was lower than the target redundancy R ($3N$ and $4N$ respectively). We have found that this system behaves the same way as when R and T were both chosen to be $4N$: it plateaus at the same point ($N = 32$), and at the same level ($F = 0.6$). The difference is so subtle, that we have omitted a figures about this case, as the reader could not distinguish them from the right side charts in Figure 6: the average reliability over all cases is slightly lower when $T < R$, the difference is 10^{-3} when $A = 1$ and $3 \cdot 10^{-2}$ when $A = 0.5$. Graphically, this means that gray points would be unnoticeably darker, but the "white area" would be essentially the same. Note that the similar case is the one with the higher redundancy; that

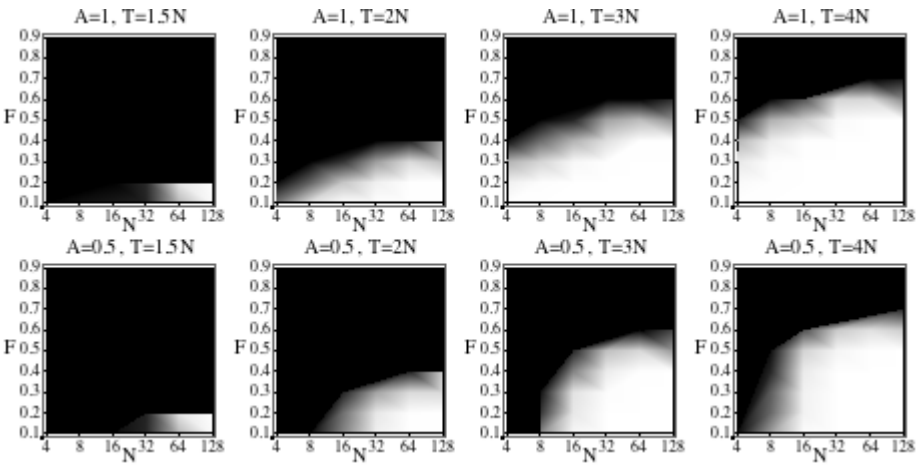

Fig. 6: Reliability as a function of N and F, for specific values of T and A. First row: $A = 1$, second row: $A = 0.5$.

is, although we allow the system to degrade to a block count of T, it essentially works like if it was repaired in every maintenance event.

4 Conclusion

In this paper, our goal was to determine the reliability of a distributed storage system employing random linear coding. With random linear coding we are trying to address the extreme challenges posed by volunteer storage systems, which stem from the nodes of a volunteer system being unreliable.

We have experimented with a model which, we believe, captures the main properties of a volunteer storage reasonably well; and used parameters that emulate the unreliability of such a system.

We have shown that random linear coding performs surprisingly well under such conditions, and can tolerate huge loss of data even when redundancy is relatively low. We have also shown that it achieves this reliability with virtually no wasted bandwidth. Furthermore, while using sparse coding is good solution for random linear coding being CPU intensive, its effects on the reliability are negligible.

Based on these results we conclude that random linear coding is a suitable solution for volunteer storage systems; therefore, we feel confident implementing such a storage system in the future on top of the BOINC middleware to support data intensive applications on volunteer computing platforms.

Bibliography

[1] Random Network Coding Library. https://github.com/avisegradi/rnc-lib. Accessed: 2013-09-10.

[2] *Efficient Random Network Coding for Distributed Storage Systems*, Ádám Visegrádi and Péter Kacsuk. Euro-Par 2013, MHPC Workshop [in press].

[3] S. Acedanski, S. Deb, M. Médard, and R. Koetter. How good is random linear coding based distributed networked storage. In *Workshop on Network Coding, Theory and Applications*, 2005.

[4] R. Ahlswede, N. Cai, S. Li, and R. Yeung. Network information flow. *Information Theory, IEEE Transactions on*, 46(4):1204–1216, 2000.

[5] D. Anderson. BOINC volunteer data archival. https://boinc.berkeley.edu/trac/wiki/VolunteerDataArchival. Accessed: January 2014.

[6] D. Anderson. BOINC volunteer storage. https://boinc.berkeley.edu/trac/wiki/VolunteerStorage. Accessed: January 2014.

[7] D. Anderson. BOINC: a system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4 – 10, Nov. 2004.

[8] D. P. Anderson and G. Fedak. The computational and storage potential of volunteer computing. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 73–80. IEEE, 2006.

[9] D. Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11:21, 2007.

[10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *Information Theory, IEEE Transactions on*, 56(9):4539–4551, 2010.

[12] A. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99(3):476–489, 2011.

[13] C. Fragouli, J.-Y. Le Boudec, and J. Widmer. Network coding: an instant primer. *ACM SIGCOMM Computer Communication Review*, 36(1):63–68, 2006.

[14] T. Ho, M. Médard, R. Koetter, D. Karger, M. Effros, J. Shi, and B. Leong. A random linear network coding approach to multicast. *Information Theory, IEEE Transactions on*, 52(10):4413–4430, 2006.

[15] S.-Y. Li, R. W. Yeung, and N. Cai. Linear network coding. *Information Theory, IEEE Transactions on*, 49(2):371–381, 2003.

[16] G. Ma, Y. Xu, M. Lin, and Y. Xuan. A content distribution system based on sparse linear network coding. *NetCod'07*, 2007.

[17] D. MacKay. Fountain codes. In *Communications, IEEE Proceedings-*, volume 152, page 1062–1068, 2005.

[18] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial & Applied Mathematics*, 8(2):300–304, 1960.

[19] R. Rodrigues and B. Liskov. High availability in DHTs: erasure coding vs. replication. *Peer-to-Peer Systems IV*, page 226–239, 2005.

[20] M. Wang and B. Li. How practical is network coding? In *Quality of Service, 2006. IWQoS 2006. 14th IEEE International Workshop on*, page 274–278, 2006.

[21] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. *Peer-to-Peer Systems*, page 328–337, 2002.