Distributed Computing Column 49
Coding for Distributed Storage

Idit Keidar
Dept. of Electrical Engineering, Technion
Haifa, 32000, Israel
idish@ee.technion.ac.il

Storage systems nowadays are increasingly distributed. While disk arrays, which are a form of a tightly coupled distributed system, have been in use for over two decades, the trend today is to move towards networked, widely dispersed distributed storage comprised of loosely coupled nodes. Coding plays an important role in both contexts, as it can help provide fault-tolerance without an excessive storage overhead. Indeed, minimizing storage redundancy for a given level of fault-tolerance was the first requirement considered in the design of codes for distributed storage systems. Over the years, numerous other considerations have come to light, driving the coding community to develop solutions that cater the various needs of distributed storage systems. The current shift towards networked storage has raised the need for yet additional properties from codes, which are the subject of much ongoing research.

This column deals with advances in coding theory that are (or might be) applicable to distributed storage. It begins with a primer by Yuval Cassuto, describing the different considerations in distributed storage, as well as codes designed to address them. Yuval lists a range of requirements from codes for distributed storage. The list begins with properties required in disk arrays, such as low redundancy and low encode/decode complexity, and continues to considerations that arise in wide-area distributed storage such as degraded reads and efficient rebuild. The latter is the focus of our second contribution, by Anwitaman Datta and Frédérique Oggier. Their article gives an overview of codes that aim to achieve better repairability in networked distributed storage systems. In particular, Anwitaman and Frédérique consider the rebuild cost in the face of concurrent failures.

Many thanks to Yuval, Anwitaman, and Frédérique for their contributions!

Call for contributions: I welcome suggestions for material to include in this column, including news, reviews, open problems, tutorials and surveys, either exposing the community to new and interesting topics, or providing new insight on well-studied topics by organizing them in new ways.

What Can Coding Theory Do for Storage Systems?

Yuval Cassuto
Dept. of Electrical Engineering, Technion
Haifa, 32000, Israel
ycassuto@ee.technion.ac.il

Abstract

Storage systems and coding theory are two very different but highly interdependent engineering disciplines. The development of storage systems has impacted research in coding theory, and coding-theoretic contributions changed the way storage systems work. The presentation aims to give a short summary of the different problems solved by coding theory at the service of storage systems.

1 Introduction

Designing a working and performant storage system is a challenging engineering feat. Such a system is built on tremendous amount of knowledge and knowhow, with endless optimizations to fit the real-world conditions. One decision that is made very early in the design is how the system will address the issue of node failures. On the one hand, the processes that cause failures are complex and non-deterministic. On the other hand, the fault tolerance of the system needs to be clearly and unequivocally conveyed to the customer. The bridge between these different environments is provided by *codes*, which are combinatorial objects with well defined and provable properties and behaviors under certain conditions. The most common code used for fault tolerance is the repetition code, more commonly known as *replication* in storage-system terminology. When a data unit is replicated n times, it is clear that any $n-1$ or less missing units can be tolerated.

More generally, a storage system implements a code with n *coding units* spread across n nodes, and r of the n coding units are *redundant*. The size of the coding unit is a parameter that is determined by both the code properties (e.g., encoding/decoding complexity) and by the system characteristics (e.g., the typical access granularity to the system). In most practical storage systems the code has to be *systematic*, i.e., is composed of two types of coding units: *data units* and *parity units*. Parity units are calculated from data units using some arithmetic operations.

Since the code is deeply engrained in the system and its operation, it needs to satisfy different requirements so as to not interfere with the normal operation of the system, or cast a burden on its

performance. In this short presentation, we discuss a few of these code requirements, and briefly mention the coding-theoretic techniques used to achieve them. We focus here on *low redundancy*, *low encoding/decoding complexity*, *low update complexity*, *degraded-read efficiency*, and *rebuild efficiency*. Each is discussed in a separate section. The sections are given in a roughly chronological order, according to the times when the topic was put in the spotlight of coding theory research. By no means do we claim that the storyline or examples described here are the best representatives of the subject matter. Additionally, the sheer amount of high quality publications in these areas allows the inclusion of but a tiny sample of works we find most convenient for the presentation.

2 Low Redundancy

The principal goodness criterion of a storage system is the *storage efficiency*, defined as the amount of logical storage available to the customer, divided by the amount of physical storage used by the system. Since in most cases the storage capacity is the main feature of the system, and the physical storage media is the dominant component in bill-of-materials costs, it is clear why a high storage efficiency is a valuable property. An equivalent objective to maximizing the storage efficiency is to minimize the *storage redundancy*, where the latter is defined as the amount of physical storage that depends on information stored elsewhere in the system, and thus cannot be used by the customer to store arbitrary data. The main reason to use redundancy in storage systems is to avoid the loss of precious customer data when failures are incident upon system components. So implicit to any redundant storage-system design is the fact that the cost of losing data is significantly higher than the cost of the hardware that carries the data.

It is a fortunate coincidence that a large body of work in the field of information theory was in place to assist in the quest for low-redundancy reliable storage systems. While primarily motivated by digital communications [23] applications, the field of information theory provided storage systems with well-developed constructive and analytic tools to address the reliability vs. redundancy tradeoff. For example, the concept of *erasures*, used by (erasure-coded) storage systems to describe full-node failures, was introduced by Peter Elias as early as in 1954 [14]. In fact, the erasure channel is considered in information theory as the simplest non-trivial noisy channel, achieving great success in obtaining constructive coding results and analytical understanding.

The greatest contribution of classical coding theory to storage systems is the family of Reed-Solomon (RS) codes [21]. The key appeal of RS codes is that they achieve optimality with respect to erasure correction – known as the MDS[1] property – and they do so for all combinations of number of nodes, number of correctable erasures, and coding unit size (not smaller than log the number of nodes). In fact, the lower bound on the coding unit size (as a function of the number of nodes), limiting in other applications, is a non-issue for storage systems that anyhow use unit sizes greater than anything necessary for RS code existence. To deploy an RS code in a storage system, each coding unit of size m bits is regarded as an element from $\mathrm{GF}(2^m)$, a Galois field with 2^m elements. Correspondingly, the encoding, update, and erasure-decoding operations are implemented using finite-field arithmetic (additions, multiplications, reciprocations, exponentiations) over $\mathrm{GF}(2^m)$. It is important to note that multiplication of two $\mathrm{GF}(2^m)$ elements requires multiplying a pair of polynomials with m binary coefficients, which requires $O(m^2)$ bit operations in a straightforward implementation, or $O(m \log m \log\log m)$ operations using much trickier spectral techniques. As a result, RS codes over large coding units in general require complex hardware implementations

[1] MDS=Maximum Distance Separability

and non-trivial designs. Despite this inherent difficulty, RS codes are very dominant in storage system implementations, a lot thanks to their rich structure that attracted a massive amount of research toward their efficient realization. One flavor of RS codes that was found most apt for implementation is the Cauchy-matrix RS codes [6].

3 Low Encoding/Decoding Complexity

On the ground of RS codes' non-trivial implementation formed a new branch of coding theory, one that aims to replace the complex arithmetic by simple eXclusive OR (XOR) operations. The objects of study in that new branch are called *array codes* [3]. As their name implies, array codes are defined over two-dimensional arrays carrying $b \times n$ coding units (data+parity). Moving from one-dimensional coding theory to two-dimensions allows combining simple XOR operations on small coding units, with a full-column erasure model[2] corresponding to a full-node failure in the storage system. Array codes also enjoy simpler, geometrically specified design, compared to the algebraically specified RS codes. Thus a hardware/software implementor can observe the coding XOR operations directly, without the overhead of an intermediary layer of finite-field arithmetic. The encoding of array codes can be specified pictorially by a *parity-group diagram*, which gives the parity constraints the array bits need to satisfy. For example, each shape in the diagram of Figure 1 shows a parity constraint among the coding units that carry it. This particular example depicts a simple horizontal parity, but any two-dimensional parity groups can be specified in a similar way. It is clear why a code that has a simple geometric specification is advantageous for implementation.

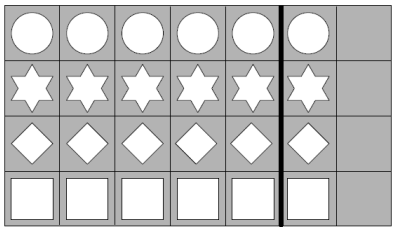


Figure 1: Geometric specification of the horizontal parity. Each coding unit on the rightmost column is the XOR of all the units to its left that carry the same shape.

The fact that array codes use simple XOR operations does not mean that they do not enjoy a rich algebraic structure. In fact, many of the ideas used for array-code constructions originated from a paper that developed an algebraic framework for array codes. In 1993, Blaum and Roth [4] showed that MDS array codes for any number of erasures can be obtained by simple geometric specification of parity groups. Specifically, taking the r parity groups as diagonals with slopes $\{0, \ldots, r-1\}$ yields an r-erasure MDS code when the array dimensions are $(p-1) \times (n \leq p)$, for some prime p. This construction and the algebraic framework that supports it (RS-like check matrices over a polynomial ring) were the foundation upon which many practically successful diagonal-based array codes were constructed [1, 2, 12, 18].

An example of a diagonal parity is given in Figure 2.

Figure 1 and Figure 2 together specify the encoding rules of two parity columns of a $r = 2$ array code.

[2]also called *phased burst erasures* in the literature.

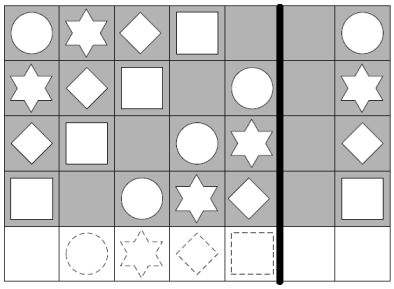


Figure 2: A parity-group specification of a slope 1 diagonal parity. The diagonals wrap around from the top back to the bottom, skipping an imaginary row mandated by the algebraic framework of [4].

4 Low Update Complexity

One aspect of storage systems not captured by traditional coding theory is the *update* problem. The operation model assumed by traditional coding theory is one with an encoder emitting codewords to a channel, and a decoder that corrects the garbled codewords. In a storage system, a "codeword" may be as large as the entire content of data+parity in the system. Therefore, selective update, and not full encoding is the operation whose efficiency counts. The *update complexity* of a code is defined as the average total number of updated coding units (data+parity) needed for a single data-unit update. The importance of the update complexity is that it multiplies the time and wear penalty of writing to the storage system in normal operation, even if no failures occur! Thus a good array code must take care to update only a few parity units for each update of a data unit. A fundamental and easy to prove lower limit on the update complexity now follows.

Proposition 1. *A code that corrects r erasures must have update complexity at least $r + 1$.*

It is clear that if a data update results in r or less total code updates, then if the columns of these r update locations are all erased, it is not possible to recover the data.

But when calculating the update complexity of popular RAID-6 array codes such as EVENODD [1] and RDP [12], we find that they do not quite meet this fundamental bound. Both codes correct $r = 2$ erasures with an update complexity close to $4 = r + 2$. More precisely, their (slightly different) update complexities are $4 - o(1)$, where $o(1)$ tends to 0 as the array dimensions go to infinity. This suboptimality costs 33% in writing time and wear. For general r erasures, the update complexity of the array-code construction of [2] is $2r - o(1)$, reflecting a factor 2 gap from optimality.

It turns out [5] that to obtain an update complexity of exactly $r + 1$, one needs to give up an important property of the code: having dedicated parity columns in the array. This result motivated the exploration of MDS array codes with optimal update complexity, whose parity bits are spread across all array columns, and not confined to r parity columns as in EVENODD and RDP.

4.1 Optimal update-complexity MDS array codes

The prospects of deploying in storage systems array codes that have both optimal redundancy (MDS) and optimal update complexity have motivated significant coding-theory research in this direction. A precursory example of the existence of such codes was found in [27], where $r = 2$ codes

were constructed for any $(p-1)/2 \times (p-1)$ array with p a prime. More such $r=2$ codes were added by [26] for $(p-1) \times 2(p-1)$ arrays with p a prime. Other codes for $r=3,4$ were obtained by [19]. Finally, for all these parameters, MDS optimal-update codes were constructed with the cyclic property [8], which reduces the implementation complexity of the codes.

The parity bits in all known optimal-update MDS codes are laid out in *rows* of the array. An example of a specification of such a $r=3$ code is given in Figure 3. This code has 6 information

a_0	a_1	a_2	a_3	a_4	a_5
$a_2+a_3+a_4$	$a_3+a_4+a_5$	$a_4+a_5+a_0$	$a_5+a_0+a_1$	$a_0+a_1+a_2$	$a_1+a_2+a_3$

Figure 3: Example of an $r=3$ cyclic MDS code with optimal update complexity.

bits $a_0, \ldots, a_5$, and 6 parity bits whose XOR operations are specified in the bottom row. The MDS property is seen in that all 6 information bits can be recovered from *any* set of $r=3$ columns. The optimal-update property is found in that each information bit appears in $r=3$ parity bits.

Moving from the column parity layout to the row layout of optimal-update codes introduces one advantage (+) and one disadvantage (−), as described in the following.

+ **Parity-write load balancing.** When a bulk of updates is applied to the storage system, the row layout balances the corresponding parity updates across the entire system. In contrast, the column layout refers all parity updates to the same columns, which requires applying external load balancing techniques such as *striping*.

− **Shortening difficulty.** In the dedicated parity columns case, a storage system with n' nodes could use any code with $n \geq n'$ columns, with the remaining $n-n'$ columns implicitly set to all zeros. In the row parity layout, every column has a parity component, and cannot be removed from the array simply by setting all its bits to zero.

5 Giving Up MDS for Efficient Degraded Read

So far in the presentation, all the surveyed codes belonged to the class of MDS codes. But in practice having strictly optimal redundancy may be secondary to more important system features. This observation was made in [17], for the particular feature of efficient *degraded reads*. Degraded reads are read operations that cannot be accomplished from their systematic locations in (yet to be rebuilt) failing nodes, and must be calculated from a combination of data units and parity units in non-failing nodes. For that purpose, [17] proposed the *Pyramid construction*, which is a transformation of MDS codes to non-MDS codes such that data units in fewer columns are required to perform a degraded read. This is done by taking a parity group of an MDS code and splitting it to two or more parities, such that each is calculated from only a subset of the array columns. Adding these "local" parities to the code allows reconstruction of data from a smaller subset of the columns in small failure events.

In the process of splitting parity groups in the Pyramid transformation, the erasure correction capabilities of the code are severely compromised. For example, a Pyramid code with $r=4$ parity columns, generated by splitting one parity group of a $r=3$ MDS code, can recover from only 1/2 of the 4-erasure combinations. In contrast, the algebraic construction of [9] offers the

same degraded-read capabilities while recovering from 7/8 of the 4-erasure combinations. It also shows that with sparser parity groups the code can have significantly lower decoding and update complexities, important enough features to motivate the minor loss of erasure correctability over MDS codes.

There are many storage-system architectures in the literature that use codes with sub-optimal redundancy to gain a more important system feature. This is a well motivated choice, which can be made more efficient with a proper coding-theoretic modeling followed by constructions that are optimal for some joint design parameters.

6 Efficient Rebuild

New code-design considerations are raised as erasure codes move from the comfort of a collocated array to a wide-area distributed system. The first of these considerations was the amount of information needed to be communicated in order to rebuild the content of a storage node after its failure. Codes that aim to minimize the rebuild communication cost are called *rebuilding codes* or *regenerating codes*. The problem of efficient rebuilding codes was introduced by [13], in which a detailed characterization of the tradeoff between the code's storage and communication efficiencies was developed. The characterization includes both constructions and fundamental information-theoretic limits. This work has motivated a large effort toward constructing efficient rebuilding codes, and in particular ones that enjoy many of the good properties mentioned in earlier sections (systematic, low redundancy, simple arithmetic, low update complexity). One of these constructions is the *Zigzag code* [25], which simultaneously achieves the MDS property, optimal update complexity, and optimal rebuilding. This impressive set of features comes with two caveats: arithmetic over fields of size at least 3, and an exponential number of coding units in each column. (In comparison, RS codes require a logarithmic number, and the array codes of Sections 3 and 4 require a linear number.) These two caveats are shown in [25] to be necessary to achieve the triple optimality.

A related code-design consideration is to minimize the *number of nodes* that participate in the rebuild. We note that this is a different consideration than minimizing the total rebuild communication, since the optimal schemes for the latter communicate (a little) information from *all* the non-failing nodes in the system. Codes rebuilding from a small set of nodes are related to a well studied concept in theoretical computer science called *locally decodable codes*. Characterization of the achievability and limits for such codes was contributed in [16].

Apart from the two code-design considerations mentioned in the preceding paragraphs, several others (such as self-repairing codes discussed later in this column[3]) were studied with similar success. It is expected that with the growing ubiquity and diversity of distributed storage systems, more such code-design considerations will meet rigorous coding-theoretic treatment. The resulting constructions hold a true potential to improve distributed storage systems in various ways.

7 Future Directions

As these lines are written, many more codes for distributed storage systems are constructed by different research groups around the world. So there is no real need to mark the directions for

[3]Datta and Oggier, An Overview of Codes Tailor-made for Better Repairability in Networked Distributed Storage Systems, ACM SIGACT News 44(1), March 2013.

the future, as it is happening anyhow. That said, it will be advantageous to mention here two important areas of coding theory that so far were left out of the main thread of distributed-storage codes. It is a matter of personal opinion that in the future they will take a more central role in protecting and distributing data in storage systems.

1. **LDPC codes.** Low Density Parity Check (LDPC) codes are the main pillar of modern coding theory [22]. In many applications they offer the state of the art performance. In storage systems however, they did not receive significant research attention, despite the commonality between them and array codes in using low-density matrices [15]. To connect between LDPC codes and storage systems, we recently proposed a new two-dimensional erasure channel model that combines node failures with random erasures within the failing nodes [10]. We showed that over this channel a new two-dimensional LDPC construction outperforms traditional algebraically constructed array codes. This new coding framework is not "storage system ready" yet (for example, the codes are not given in systematic form), but we believe that it has a high potential to become a practical alternative.

2. **Fountain codes.** The fountain coding framework [7] achieved great success in distributing data over lossy network links. This success is attributed to the construction of very low overhead codes with low complexity of decoding [20, 24]. When we use fountain codes for storage, we need to distribute coded packets to network nodes, such that some global data-recoverability properties are maintained. When nodes fail, rebuilding is done by redistributing code packets between nodes to retain these global properties. In order for the fountain code to support this use case, it must provide the nodes with a good idea of the current decodability *state* of their code packets, at a finer granularity than able/not able to fully decode. A small step toward this objective was taken in [11] by proposing a fountain code whose decoding state is given by the sizes of the connected components of a graph. As a result, the node that holds the coded packets can tell fellow nodes in the system which types of packets are best for it at the current instant.

8 Conclusion

We hope that this short presentation made the case that coding theory has been instrumental to the development of efficient storage systems. More importantly, we are certain that emerging distributed storage-system architectures will enjoy even richer coding-theoretic contributions.

References

[1] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures," *IEEE Transactions on Computers*, vol. 44, no. 2, pp. 192–202, 1995.

[2] M. Blaum, J. Bruck, and A. Vardy, "MDS array codes with independent parity symbols," *IEEE Transactions on Information Theory*, vol. 42, no. 2, pp. 529–542, 1996.

[3] M. Blaum, P. Farrell, and H. van Tilborg, "Array codes," *Handbook of Coding Theory, V.S. Pless and W.C. Huffman*, pp. 1855–1909, 1998.

[4] M. Blaum and R. Roth, "New array codes for multiple phased burst correction," *IEEE Transactions on Information Theory*, vol. 39, no. 1, pp. 66–77, 1993.

[5] ——, "On lowest density MDS codes," *IEEE Transactions on Information Theory*, vol. 45, no. 1, pp. 46–59, 1999.

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

[7] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A digital fountain approach to reliable distribution of bulk data," in *Proc. ACM SIGCOMM'98*, Vancouver BC,Canada, 1998, pp. 56–67.

[8] Y. Cassuto and J. Bruck, "Cyclic lowest-density MDS array codes," *IEEE Transactions on Information Theory*, vol. 55, no. 4, pp. 1721–1729, 2009.

[9] ——, "Low-complexity array codes for random and clustered 4-erasures," *IEEE Transactions on Information Theory*, vol. 58, no. 1, pp. 146–158, 2012.

[10] Y. Cassuto and M. A. Shokrollahi, "Array-code ensembles -or- two-dimensional LDPC codes," in *Proc. of the IEEE International Symposium on Info. Theory*, St. Petersburg Russia, 2011.

[11] ——, "On-line fountain codes for semi-random loss channels," in *Proc. IEEE Information Theory Workshop*, Paraty, Brazil, 2011.

[12] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," in *In Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, San-Francisco CA, 2004.

[13] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Transactions on Information Theory*, vol. 56, no. 9, pp. 4539–4551, 2010.

[14] P. Elias, "The binary erasure channel," *MIT Information Theory Lectures*, 1954.

[15] J. L. Fan, "Array codes as low-density parity check codes," in *Proc. of the Intl. Symp. on Turbo Codes*, 2000, pp. 543–546.

[16] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin, "On the locality of codeword symbols," *ECCC: TR11-100*, 2011, to appear in IEEE Trans. Info. Theory.

[17] C. Huang, M. Chen, and J. Li, "Pyramid codes: flexible schemes to trade space for access efficiency in reliable data storage systems," in *In Proceedings of the Sixth IEEE International Symposium on Network Computing and Applications*, Cambridge, MA USA, 2007.

[18] C. Huang and L. Xu, "Star: An efficient coding scheme for correcting triple storage node failures," in *In Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San-Francisco CA, 2005.

[19] E. Louidor and R. Roth, "Lowest-density MDS codes over extension alphabets," *IEEE Transactions on Information Theory*, vol. 52, no. 7, pp. 3186–3197, 2006.

[20] M. Luby, "LT codes," in *Proc. of the Annual IEEE Symposium on Foundations of Computer Science FOCS*, Vancouver BC, Canada, 2002, pp. 271–280.

[21] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *SIAM J. Appl. Math.*, vol. 8, pp. 300–304, 1960.

[22] T. Richardson and R. Urbanke, *Modern coding theory.* New York USA: Cambridge University Press, 2008.

[23] C. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 9, pp. 379–423, Oct. 1948.

[24] M. A. Shokrollahi, "Raptor codes," *IEEE Transactions on Information Theory*, vol. 52, no. 6, pp. 2551–2567, 2006.

[25] I. Tamo, Z. Wang, and J. Bruck, "Zigzag codes: MDS array codes with optimal rebuilding," *arXiv:1112.0371 [cs.IT]*, 2011, submitted to IEEE Trans. Info. Theory, 10/2011.

[26] L. Xu, V. Bohossian, J. Bruck, and D. Wagner, "Low-density MDS codes and factors of complete graphs," *IEEE Transactions on Information Theory*, vol. 45, no. 6, pp. 1817–1826, 1999.

[27] G. Zaitsev, V. Zinov'ev, and N. Semakov, "Minimum-check-density codes for correcting bytes of errors, erasures, or defects," *Problems Inform. Transm.*, vol. 19, pp. 197–204, 1981.

An Overview of Codes Tailor-made for Better Repairability in Networked Distributed Storage Systems

Anwitaman Datta Frédérique Oggier
Nanyang Technological University, Singapore
{anwitaman, frederique}@ntu.edu.sg

Abstract

The increasing amount of digital data generated by today's society asks for better storage solutions. This survey looks at a new generation of coding techniques designed specifically for the maintenance needs of networked distributed storage systems (NDSS), trying to reach the best compromise among storage space efficiency, fault-tolerance, and maintenance overheads. Four families of codes, namely, pyramid, hierarchical, regenerating and locally repairable codes such as self-repairing codes, along with a heuristic of cross-object coding to improve repairability in NDSS are presented at a high level. The code descriptions are accompanied with simple examples emphasizing the main ideas behind each of these code families. We discuss their pros and cons before concluding with a brief and preliminary comparison. This survey deliberately excludes technical details and does not contain an exhaustive list of code constructions. Instead, it provides an overview of the major novel code families in a manner easily accessible to a broad audience, by presenting the big picture of advances in coding techniques for maintenance of NDSS.

Keywords: coding techniques, networked distributed storage systems, hierarchical codes, pyramid codes, regenerating codes, locally repairable codes, self-repairing codes, cross-object coding.

1 Introduction

We live in an age of data deluge. A study sponsored by the information storage company *EMC* estimated that the world's data is more than doubling every two years, reaching 1.8 zettabytes (1 ZB = 10^{21} Bytes) of data to be stored in 2011.[1] This includes various digital data continuously being generated by individuals as well as business and government organizations, who all need scalable solutions to store data reliably and securely.

Storage technology has been evolving fast in the last quarter of a century to meet the numerous challenges posed in storing an increasing amount of data and catering to diverse applications with

[1] http://www.emc.com/about/news/press/2011/20110628-01.htm

different workload characteristics. In 1988, RAID (Redundant Arrays of Inexpensive Disks) was proposed [23], which combines multiple storage disks (typically from two to seven) to realize a single logical storage unit. Data is stored redundantly, using replication, parity, or more recently erasure codes. Such redundancy makes a RAID logical unit significantly more reliable than the individual constituent disks. Besides meeting cost effective reliable storage, RAID systems provide good throughput by leveraging parallel I/O at the different disks, and, more recently, geographic distributions of the constituent disks to achieve resilience against local events (such as a fire) that could cause correlated failures.

While RAID has evolved and stayed an integral part of storage solutions to date, new classes of storage technology have emerged, where multiple logical storage units (simply referred to as 'storage nodes') are assembled together to scale out the storage capacity of a system. The massive volume of data involved means that it would be extremely expensive, if not impossible, to build single pieces of hardware with enough storage as well as I/O capabilities. By the term 'networked', we refer to these storage systems that pool resources from multiple interconnected storage nodes, which in turn may or not use RAID. The data is distributed across these interconnected storage units and hence the name 'networked distributed storage systems' (NDSS). It is worth emphasizing at this juncture that though the term 'RAID' is now also used in the literature for NDSS environments, for instance, HDFS-RAID [1] and 'distributed RAID'[2], but in this article we use the term RAID to signify traditional RAID systems where the storage nodes are collocated, and the number of parity blocks per data object is few, say one or two (RAID-1 to RAID-6). Unlike in traditional RAID systems where the storage disks are collocated, all data objects are stored in the same set of storage disks, and these disks share an exclusive communication bus within a stand-alone unit, in NDSS, a shared interconnect is used across the storage nodes, and different objects may be stored across arbitrarily different (possibly intersecting) subsets of storage nodes, and thus there is competition and interference in the usage of the network resources.

NDSS come in many flavors such as data centers and peer-to-peer (P2P) storage/backup systems. While data centers comprise thousands of compute and storage nodes, individual clusters such as that of Google File System (GFS) [9] are formed out of hundreds up to thousands of nodes. P2P systems like Wuala,[3] in contrast, formed swarms of tens to hundreds of nodes for individual files or directories, but would distribute such swarms arbitrarily out of hundreds of thousands of peers.

While P2P systems are geographically distributed and connected through an arbitrary topology, data center interconnects have well defined topologies and are either collocated or distributed across a few geographic regions. Furthermore, individual P2P nodes may frequently go offline and come back online (temporary churn), creating unreliable and heterogeneous connectivity. On the contrary, data centers use dedicated resources with relatively infrequent temporary outages.

Despite these differences, NDSS share several common characteristics. While I/O of individual nodes continues to be a potential bottleneck, available bandwidth, both at the network's edges and within the interconnect becomes a critical shared resource. Also, given the system scale, failure of a significant subset of the constituent nodes, as well as other network components, is the norm rather than the exception. To enable a highly available overall service, it is thus essential to tolerate both short-term outages of some nodes and to provide resilience against permanent failures of individual components. Fault-tolerance is achieved using redundancy, while long-term resilience relies on

[2]http://www.disi.unige.it/project/draid/distributedraid.html

[3]The current deployment of Wuala (`www.wuala.com`) no longer uses a hybrid peer-to-peer architecture.

replenishment of lost redundancy over time.

A common practice to realize redundancy is to keep three copies of an object to be stored (called 3-way replication): when one copy is lost, the second copy is used to regenerate the first one, and hopefully, not both the remaining copies are lost before the repair is completed. There is of course a price to pay: redundancy naturally reduces the efficiency, or alternatively put, increases the overheads of the storage infrastructure. The cost for such an infrastructure should be estimated not only in terms of the hardware, but also of real estate and maintenance of a data center. A US Environmental Protection Agency report of 2007[4] indicates that the US used 61 billion kilowatt-hours of power for data centers and servers in 2006. That is 1.5 percent of the US electricity use, and it cost the companies that paid those bills more than $4.5 billion.

There are different ways to reduce these expenses, starting from the physical media, which has witnessed a continuous shrinking of physical space and cost per unit of data, as well as reductions in terms of cooling needs. This article focuses on a different aspect, that of the trade-off between fault-tolerance and efficiency in storage space utilization via coding techniques, or more precisely erasure codes.

An erasure code $EC(n, k)$ transforms a sequence of k symbols into a longer sequence of $n > k$ symbols. Adding extra $n - k$ symbols helps in recovering the original data in case some of the n symbols are lost. An $EC(n, k)$ induces a n/k overhead. Erasure codes were designed for data transmitted over a noisy channel, where coding is used to append redundancy to the transmitted signal to help the receiver recover the intended message, even when some symbols are erasured/corrupted by noise (see Figure 1). Codes offering the best trade-off between redundancy and fault-tolerance, called maximum distance separable (MDS) codes, tolerate $n - k$ erasures, that is, no matter which group of $n - k$ symbols are lost, the original data can be recovered. The simplest examples are the repetition code $EC(n, 1)$ (given $k = 1$ symbol, repeat it n times), which is the same as replication, and the parity check code $EC(k + 1, k)$ (compute one extra symbol which is the sum of the first k symbols). The celebrated Reed-Solomon codes [26] are another instance of such codes: consider a sequence of k symbols as a degree $k - 1$ polynomial, which is evaluated in n symbols. Conversely, given n symbols, or in fact at least (any) k symbols, it is possible to interpolate them to recover the polynomial and decode the data. Think of a line in the plane. Given any $k = 2$ or more points, the line is completely determined, while with only one point, the line is lost.

This same storage overhead/fault tolerance trade-off has also long been studied in the context of RAID storage units. While RAID 1 uses replication, subsequent RAID systems integrate parity bits, and Reed-Solomon codes can be found in RAID 6. Notable examples of new codes designed to suit the peculiarities of RAID systems include weaver codes [12], array codes [29] as well as other heuristics [11]. Optimizing the codes for the nuances of RAID systems, such as physical proximity of storage devices leading to clustered failures are natural aspects [3] gaining traction. Note that even though we do not detail here those codes optimized for traditional RAID systems, they may nonetheless provide some benefits in the context of NDSS, and vice-versa.

A similar evolution has been observed in the world of NDSS, and a wide-spectrum of NDSS have started to adopt erasure codes: for example, the new version of Google's file system, Microsoft's Windows Azure Storage [2] as well as other storage solution companies such as CleverSafe[5] and Wuala. This has happened due to a combination of several factors, including years of implementation refinements, ubiquity of significantly powerful but cheap hardware, as well as the sheer scale

[4] http://arstechnica.com/old/content/2007/08/epa-power-usage-in-data-centers-could-double-by-2011.ars

[5] `http://www.cleversafe.com/`

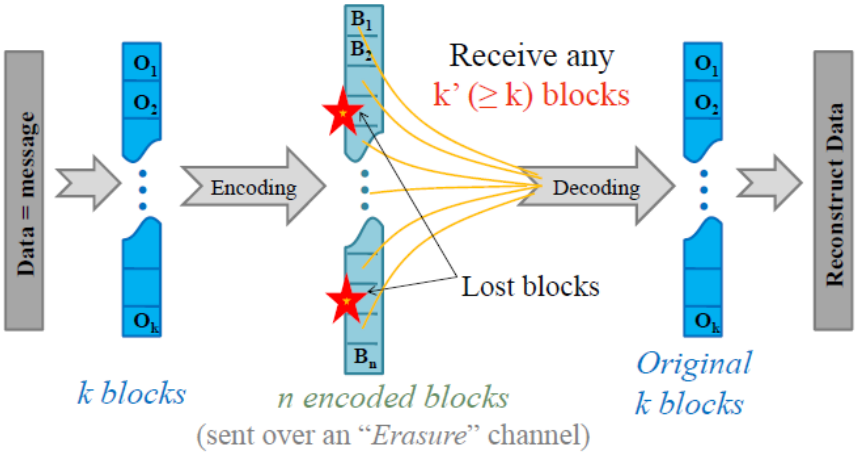


Figure 1: Coding for erasure channels: a message of k symbols is encoded into n fragments before transmission over an erasure channel. As long as at least $k' \geq k$ symbols arrive at destination, the receiver can decode the message.

of the data to be stored.

We will next elaborate how erasure codes are used in NDSS, and while MDS codes are optimal in terms of fault-tolerance and storage overhead tradeoffs, why there is a renewed interest in the coding theory community to design new codes that take into account maintenance of NDSS explicitly.

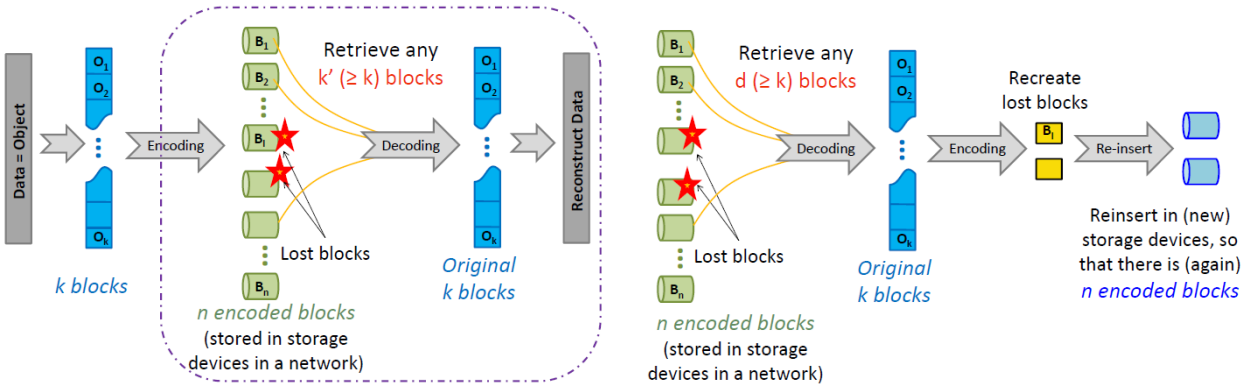
2 Networked Distributed Storage Systems

In an NDSS, if one object is stored using an erasure code and each 'encoded symbol' is stored at a different node, then the object stays available as long as the number of node failures does not exceed the code recovery capability.

Now let individual storage nodes fail according to an i.i.d. random process with the failure probability being f. The expected number of independent node failures is binomially distributed, hence the probability of losing an object with an $EC(n,k)$ MDS erasure code is $\sum_{j=1}^{k} \binom{n}{n-k+j} f^{n-k+j}(1-f)^{k-j}$. In contrast, it is f^r with r-way replication. For example, if the probability of failure of individual nodes is $f = 0.1$, then for the same storage overhead of 3, corresponding to $r = 3$ for replication and to an $EC(9,3)$ erasure code, the probabilities of losing an object are 10^{-3} and $\sim 3 \cdot 10^{-6}$ respectively. Such resilience analysis illustrates the high fault-tolerance that erasure codes provide. Using erasure codes, however, means that a larger number of storage nodes are involved in storing individual data objects.

There is however a fundamental difference between a communication channel, where erasures occur once during transmission, and an NDSS, where faults accumulate over time, threatening data availability in the long run.

Traditionally, erasure codes were not designed to reconstruct subsets of arbitrary encoded blocks efficiently. When a data block encoded by an MDS erasure code is lost and has to be recreated, one would typically first need data equivalent in amount to recreate the whole object in one place (either by storing a full copy of the data, or else by downloading an adequate number of encoded



(a) Data retrieval: as long as $k' \geq k$ nodes are alive, the object can be retrieved.

(b) Node repair: one node has to recover the object, re-encode it, and then distribute the lost blocks to the new nodes.

Figure 2: Erasure coding for NDSS: the object to be stored is cut into k, then encoded into n fragments, given to different storage nodes. Reconstruction of the data is shown on the left, while repair after node failures is illustrated on the right.

blocks), even in order to recreate a single encoded block, as illustrated in Figure 2.

In recent years, the coding theory community has thus focused on designing codes which better suit NDSS nuances, particularly with respect to replenishing lost redundancy efficiently. The focus of such works has been on (i) bandwidth, which is typically a scarce resource in NDSS, (ii) the number of storage nodes involved in a repair process, (iii) the number of disk accesses (I/O) at the nodes facilitating a repair, and (iv) the repair time, since delay in the repair process may leave the system vulnerable to further faults. Note that these aspects are often interrelated. There are numerous other aspects, such as data placement, meta-information management to coordinate the network, as well as interferences among multiple objects contending for resources, to name a few prominent ones, which all together determine an actual system's performance. The novel codes we describe next are yet to go through a comprehensive benchmarking across this wide spectrum of metrics. Instead, we hope to make these early and mostly theoretical results accessible to practitioners, in order to accelerate the process of such further investigations.

Thus the rest of this article assumes a network of N nodes, storing one object of size k, encoded into n symbols, also referred to as encoded blocks or fragments, each of them being stored at distinct n nodes out of the N choices. When a node storing no symbol corresponding to the object being repaired participates in the repair process by downloading data from nodes owning data (also called *live nodes*), it is termed a *newcomer*. Typical values of n and k depend on the environments considered: for data centers, the number of temporary failures is relatively low, thus small (n, k) values such as $(9, 6)$ or $(13, 10)$ (with respective overheads of 1.5 and 1.3) are generally fine [1]. In P2P systems such as Wuala, larger parameters like $(517, 100)$ are desirable to guarantee availability since nodes frequently go temporarily offline. When discussing the repair properties of a code, it is also important to distinguish which repair strategy is best suited: in P2P systems, a lazy approach (where several failures are tolerated before triggering repair) can avoid unnecessary repairs since nodes may be temporarily offline. Data centers might instead opt for immediate repairs. Yet,

proactive repairs can lead to cascading failures[6]. Thus in all cases, ability to repair multiple faults simultaneously is essential.

In summary, codes designed to optimize the maintenance process should take into account different code parameters, repair strategies, the ability to replenish single as well as multiple lost fragments, and repair time. Recent coding works aimed in particular at:

(i) Minimize the absolute amount of data transfer needed to recreate one lost encoded block at a time when storage nodes fail. *Regenerating codes* [6] form a new family of codes achieving the minimum possible repair bandwidth (per repair) given an amount of storage per node, where the optimal storage-bandwidth trade-off is determined using a network coding inspired analysis, assuming that each new-coming node contacts $d \geq k$ arbitrary live nodes for each repair. Regenerating codes, like MDS erasure codes, allow data retrievability from any arbitrary set of k nodes. Collaborative regenerating codes [27, 16] are a generalization allowing simultaneous repair of multiple faults.

(ii) Minimize the number of nodes to be contacted for recreating one encoded block, referred to as fan-in. Reduction in the number of nodes needed for one repair typically increases the number of ways repair may be carried out, thus avoiding bottlenecks caused by stragglers. It also makes multiple parallel repairs possible, all in turn translating into faster system recovery. To the best of our knowledge, *self-repairing codes* [17] were the first instances of $EC(n, k)$ code families achieving a repair fan-in of 2 for up to $\frac{n-1}{2}$ simultaneous and arbitrary failures. Since then, such codes have become a popular topic of study under the nomenclature of 'locally repairable codes' - the name being reminiscent of a relatively well established theoretical computer science topic of locally decodable codes. Other specific instances of locally repairable code families such as [18, 10, 25], as well as study of the fundamental trade-offs and achievability of such codes [13] have commenced in the last years.

Local repairability come at a price, since either nodes store the minimum possible amount of data, in which case the MDS property has to be sacrificed (if one encoded symbol can be repaired from other two, any set of k nodes including these 3 nodes will not be adequate to reconstruct the data), or the amount of data stored in each node has to be increased. A *resilience analysis* of self-repairing codes [17] has shown that object retrieval is little impaired by it, and in fact, the MDS property might not be as critical for NDSS as it is for communication, since NDSS have the option of repairing data.

There are other codes which fall somewhere 'in between' these extremes. Prominent among these are hierarchical and pyramid codes which we summarize first before taking a closer look at regenerating and locally repairable codes.

3 Hierarchical and Pyramid codes

Consider an object comprising eight data blocks $\mathbf{o}_1, \ldots, \mathbf{o}_8$. Create three encoded fragments $\mathbf{o}_1$, $\mathbf{o}_2$ and $\mathbf{o}_1 + \mathbf{o}_2$ using the first two blocks, and repeat the same process for blocks $\mathbf{o}_{2j+1}$ and $\mathbf{o}_{2j+2}$ (for $j = 1...3$). One can then build another layer of encoded blocks, namely $\mathbf{o}_1 + \mathbf{o}_2 + \mathbf{o}_3 + \mathbf{o}_4$ and $\mathbf{o}_5 + \mathbf{o}_6 + \mathbf{o}_7 + \mathbf{o}_8$. The fragment $\mathbf{o}_1 + \mathbf{o}_2$ may be viewed as providing *local redundancy*, while $\mathbf{o}_1 + \mathbf{o}_2 + \mathbf{o}_3 + \mathbf{o}_4$ achieves *global redundancy*. The same idea can be iterated to build a hierarchy (Figure 3), where the next level global redundant fragment is $\mathbf{o}_1 + \mathbf{o}_2 + \mathbf{o}_3 + \mathbf{o}_4 + \mathbf{o}_5 + \mathbf{o}_6 + \mathbf{o}_7 + \mathbf{o}_8$.

[6]For example `http://storagemojo.com/2011/04/29/amazons-ebs-outage/`

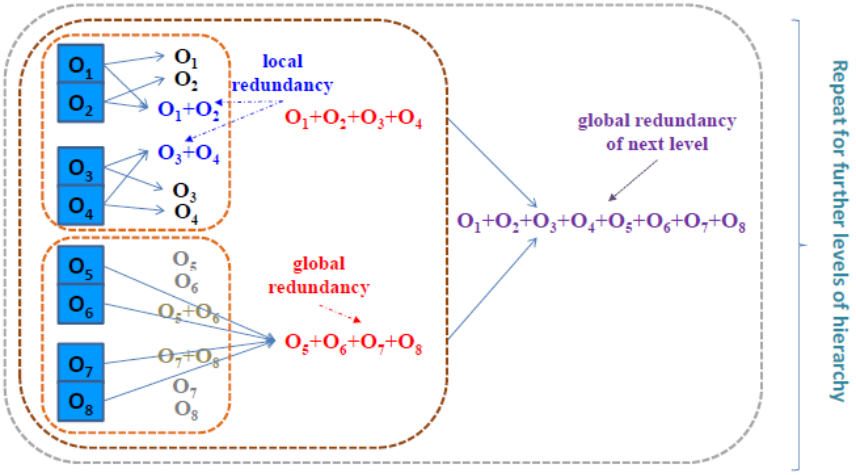


Figure 3: Hierarchical codes.

Consequently, when some of the encoded fragments are lost, localized repair is attempted, and global redundancy is used only if necessary. For instance, if the node storing $\mathbf{o}_1$ is lost, then nodes storing $\mathbf{o}_2$ and $\mathbf{o}_1 + \mathbf{o}_2$ are adequate for repair. However, if nodes storing $\mathbf{o}_1$ and $\mathbf{o}_1 + \mathbf{o}_2$ are both lost, one may first reconstruct $\mathbf{o}_1 + \mathbf{o}_2$ by retrieving $\mathbf{o}_1 + \mathbf{o}_2 + \mathbf{o}_3 + \mathbf{o}_4$ and $\mathbf{o}_3 + \mathbf{o}_4$, and then rebuild $\mathbf{o}_1$.

This basic idea can be extended to realize more complex schemes, where (any standard) erasure coding technique is used in a *bottom-up* manner to create local and global redundancy at a level, and the process is iterated. That is the essential idea behind *Hierarchical codes* [7]. For the same example, one may also note that if both $\mathbf{o}_1$ and $\mathbf{o}_2$ are lost, then repair is no longer possible. This illustrates that the different encoded pieces have unequal importance. Because of such assymmetry, the resilience of such codes have only been studied with simulations in [7].

In contrast, *Pyramid codes* [14] were designed in a top-down manner, but aiming again to have local and global redundancy to provide better fault-tolerance and improve read performance by trading storage space efficiency for access efficiency. Such local redundancy can naturally be harnessed for efficient repairs as well. A new version of Pyramid codes, where the coefficients used in the encoding have been numerically optimized, namely Locally Reconstructable Codes [15] has more recently been proposed and is being used in the Azure [2] system.

We use an example to illustrate the design of a simple Pyramid code. Take an $EC(11, 8)$ MDS code, say a Reed-Solomon code with generator matrix G, of the form

$$[x_1, \ldots, x_{11}] = [\mathbf{o}_1, \ldots, \mathbf{o}_8, c_1, c_2, c_3].$$

A Pyramid code can be built from this base code, by retaining the pieces $\mathbf{o}_1, \ldots, \mathbf{o}_8$, and two of the other pieces (without loss of generality, lets say, c_2, c_3).

Additionally, split the data blocks into two groups $\mathbf{o}_1, \ldots, \mathbf{o}_4$ and $\mathbf{o}_5, \ldots, \mathbf{o}_8$, and compute some more redundancy coefficients for each of the two groups, which is done by picking a first symbol $c_{1,1}$ corresponding to c_1 by setting $\mathbf{o}_5 = \ldots = \mathbf{o}_8 = 0$ and $c_{1,2}$ corresponding to c_1 with $\mathbf{o}_1 = \ldots = \mathbf{o}_4 = 0$.

This results in an $EC(12, 8)$, whose codewords look like

$$[\mathbf{o}_1, \ldots, \mathbf{o}_8, c_{1,1}, c_{1,2}, c_2, c_3]$$

where $c_{1,1} + c_{1,2}$ is equal to the original code's c_1:

$$c_{1,1} + c_{1,2} = c_1.$$

For both Hierarchical and Pyramid codes, at each hierarchy level, there is some 'local redundancy' which can repair lost blocks without accessing blocks outside the subgroup, while if there are too many errors within a subgroup, then the 'global redundancy' at that level will be used. One moves further up the pyramid until repair is eventually completed. Use of local redundancy means that a small number of nodes is contacted, which translates into a smaller bandwidth footprint. Furthermore, if multiple isolated (in the hierarchy) failures occur, they can be repaired independently and in parallel.

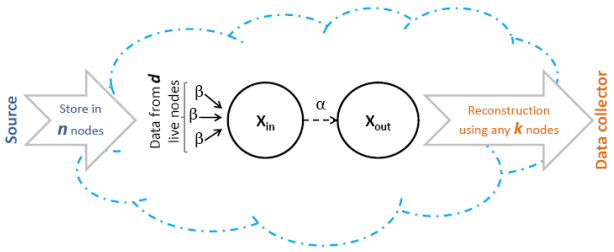
In contrast to Hierarchical codes, where analysis of the resilience has not been carried out, Pyramid codes' top-down approach allows to discern distinct failure regimes under which data is recoverable, and regimes when data is not recoverable. For instance, in the example above, as long as there are three or fewer failures, the object is always reconstructable. Likewise, if there are five or more failures, then the data cannot be reconstructed. However, there is also an intermediate region, in this simple case, it being the scenario of four arbitrary failures, in which, for certain combinations of failures, data cannot be reconstructed, while for others, it can be.

Coinciding with these works, researchers from the network coding community started studying the fundamental limits and trade-offs of bandwidth usage for regeneration of a lost encoded block vis-a-vis the storage overhead (subject to the MDS constraint) culminating in a new family of codes, broadly known as *regenerating codes*, discussed next.

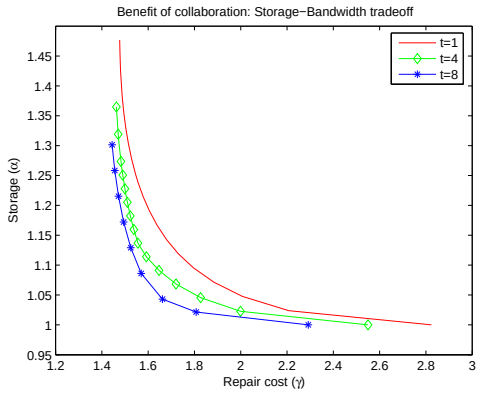
4 Regenerating codes

The repair of lost redundancy in a storage system can be abstracted as an *information flow graph* [6].

First, the data object is encoded using an $EC(n, k)$ MDS code and the encoded blocks are stored across n storage nodes. Each storage node is assumed to store an amount α of data (meaning that the size of an encoded block is at most α, since only one object is stored). When one node fails, new nodes contact $d \geq k$ live nodes and download β amount of data from each contacted node in order to perform the repair. If several failures occur, the model [6] assumes that repairs are taken care of one at a time. Information flows from the data owner to the data collector as follows (see Figure 4(a) for an illustration): (1) The original placement of the data distributed over n nodes is modeled as directed edges of weight α from the sources (data owners) to the original storage nodes. (2) A storage node is denoted by X and modeled as two logical nodes X_{in} and X_{out}, which are connected with a directed edge $X_{in} \rightarrow X_{out}$ with weight α representing the storage capacity of the node. The data flows from the data owner to X_{in}, then from X_{in} to X_{out}. (3) The regeneration process consists of directed edges of weight β from d contacted live nodes to the X_{in} of the newcomer. (4) Finally, the reconstruction/access of the whole object is abstracted with edges of weight α to represent the destination (data collector) downloading data from arbitrary k live storage nodes.



(a) Information flow graph for regenerating codes: each storage node is modeled as two virtual nodes, X_{in} which collects β amount of information from arbitrary d live nodes, while storing a maximum of α amount of information, and X_{out}, which is accessed by any data collector contacting the storage node. A max-flow min-cut analysis yields the feasible values for storage capacity α and repair bandwidth $\gamma = d\beta$ in terms of the number d of nodes contacted and code parameters n, k, where $d \geq k$.



(b) Trade-off curve for the amount of storage space α used per node, and the amount of bandwidth γ needed to regenerate a lost node. If multiple repairs t are carried out simultaneously, and the t new nodes at which lost redundancy is being created collaborate among themselves, then better trade-offs can be realized, as can be observed from the plot (done using $k = 32$, $d = 48$, n can be any integer bigger than $d + t$).

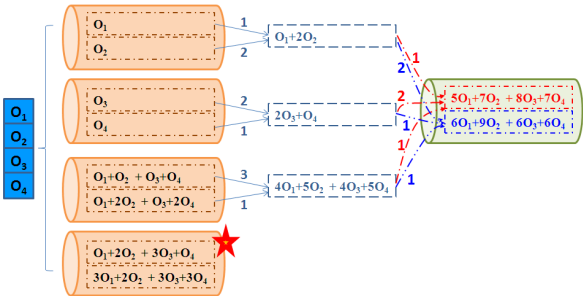
Figure 4: The underlying network coding theory inspiring regenerating codes.

Then, the maximum information that can flow from the source to the destination is determined by the max-flow over a min-cut of this graph. For the original object to be reconstructible at the destination, this flow needs to be at least as large as the size of the original object.

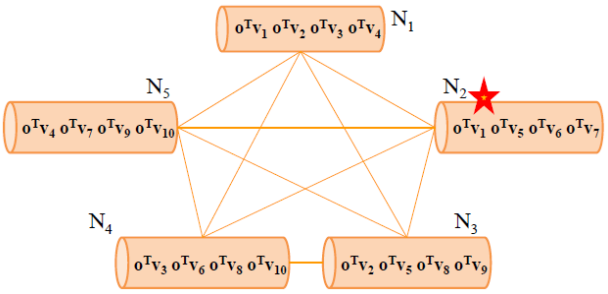
Any code that enables the information flow to be actually equal to the object size is called a regenerating code (RGC) [6]. Now, given k and n, the natural question is, *what are the minimal storage capacity α and bandwidth $\gamma = d\beta$ needed for repairing an object of a given size*? This can be formulated as a linear non-convex optimization problem: minimize the total download bandwidth $d\beta$, subject to the constraint that the information flow equals the object size. The optimal solution is a piecewise linear function, which describes a trade-off between the storage capacity α and the bandwidth β as shown in Figure 4(b) [$t = 1$], and has two distinguished boundary points: the minimal storage repair (MSR) point (when α is equal to the object size divided by k), and the minimal bandwidth repair (MBR) point.

The trade-off analysis only determines what can best be achieved, but in itself does not provide any specific code construction. Several codes have since been proposed, most of which operate either at the MSR or MBR points of the trade-off curve, e.g., [24].

The specific codes need to satisfy the constraints determined by the max-flow min-cut analysis, however there is no constraint or need to regenerate precisely the same (bitwise) data as was lost (see Figure 5 (a)). When the regenerated data is in fact not the same as that lost, but nevertheless provides equivalent redundancy, it is called *functional regeneration*, while if it is bitwise identical to what was lost, then it is called *exact regeneration*, as illustrated in Figure 5 (b). Note that the proof of storage-bandwidth trade-off determined by the min-cut bound does not depend on the type of repair (functional/exact).



(a) An example of functional repair for $k = 2$ and $n = 4$, adapted from [6]: an object is cut into 4 pieces $\mathbf{o}_1, \ldots, \mathbf{o}_4$, and two linear combinations of them are stored at each node. When the 4th node fails, a new node downloads linear combinations of the two pieces at each node (the number on each edge describes what is the factor that multiplies the encoded fragment), from which it computes two new pieces of data, different from those lost, but any $k = 2$ of the 4 nodes permit object retrieval.



(b) An example of exact repair from [24]: an object $\mathbf{o}$ is encoded by taking its inner product with 10 vectors $\mathbf{v}_1, \ldots, \mathbf{v}_{10}$, to obtain $\mathbf{o}^T\mathbf{v}_i$, $i = 1, \ldots, 10$, as encoded fragments. They are distributed to the 5 nodes $N_1, \ldots, N_5$ as shown. Say, node N_2 fails. A newcomer can regenerate by contacting every node left, and download one encoded piece from each of them, namely $\mathbf{o}^T\mathbf{v}_1$ from N_1, $\mathbf{o}^T\mathbf{v}_5$ from N_3, $\mathbf{o}^T\mathbf{v}_6$ from N_4 and $\mathbf{o}^T\mathbf{v}_7$ from N_5.

Figure 5: Regenerating codes: functional versus exact repair.

The original model [6] has since been generalized [16, 27] to show that in case of multiple faults, the new nodes carrying out regenerations can collaborate among themselves to perform several repairs in parallel, which was in turn shown to reduce the overall bandwidth needed per regeneration (Figure 4(b) $t > 1$ representing the number of failures/new collaborating nodes). Instances of codes for this setting, referred to as collaborative regenerating codes (CRGC) are rarer than classical regenerating codes, and up to now, only a few code constructions are known [27, 28].

(Collaborative) regenerating codes stem from a precise information theoretical characterization. However, they also suffer from algorithmic and system design complexity inherited from network coding, which is larger than even traditional erasure codes, apart from the added computational overheads. The value of fan-in d for regeneration has also practical implications. With a high fan-in d even a small number of slow or overloaded nodes can thwart the repairs.

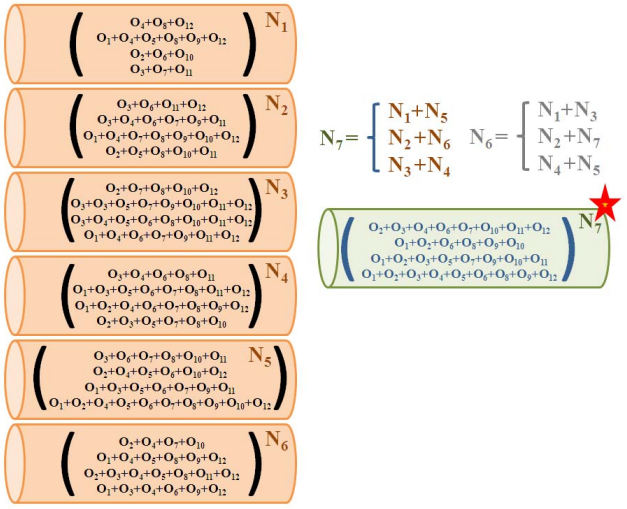
5 Locally repairable codes

The codes proposed in the context of network coding aim at reducing the repair bandwidth, and can be seen as the combination of an MDS code and a network code. Hierarchical and Pyramid codes instead tried to reduce the repair degree or fan-in (i.e., the number of nodes needed to be contacted to repair) by using "erasure codes on top of erasure codes". We next present some recent families of locally repairable codes (LRC) [17, 19, 18, 25], which minimize the repair fan-in d, trying to achieve $d << k$ such as $d = 2$ or 3. Forcing the repair degree to be small has advantages in terms of repair time and bandwidth, however, it might affect other code parameters (such as its rate, or storage overhead). We will next elaborate a few specific instances of locally repairable codes.

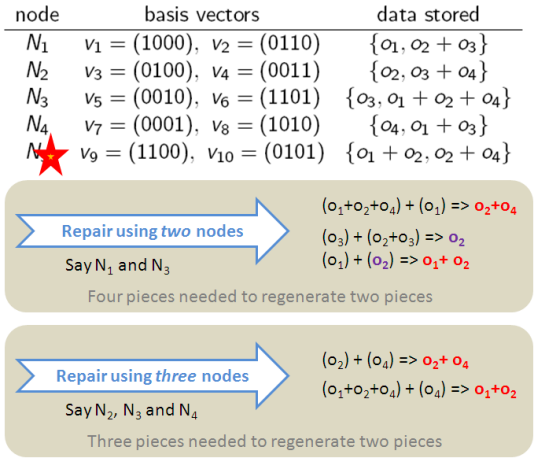
The term "locally repairable" is inspired by [10], where the repair degree d of a node is called the "locality d" of a codeword coordinate, and is reminiscent of *locally decodable* and *locally correctable*

codes, which are well established topics of study in theoretical computer science. Self-repairing codes (SRC) [17, 19] were to our knowledge the first (n, k) codes designed to achieve $d = 2$ per repair for up to $\frac{n-1}{2}$ simultaneous failures. Other families of locally repairable codes based on projective geometric construction (Projective Self-repairing Codes) [18] and puncturing of Reed-Mueller codes [25] have been very recently proposed. Some instances of these latter codes can achieve a repair degree of either 2 or 3.

With $d = 2$ resources of at most two live nodes may get saturated due to a repair. Thus simultaneous repairs can be carried out in parallel, which in turn provides fast recovery from multiple faults. For example, in Figure 6(a) if the 7th node fails, it can be reconstructed in 3 different ways, by contacting either N_1, N_5, or N_2, N_6, or N_3, N_4. If both the 6th and 7th node fail each of them can still be reconstructed in two different ways. One newcomer can contact first N_1 and then N_5 to repair N_7, while another newcomer can in parallel contact first N_3 then N_1 to repair N_6.



(a) An example of self-repairing codes from [17]: the object **o** has length 12, and encoding is done by taking linear combinations of the 12 pieces as shown, which are then stored at 7 nodes.



(b) An example of self-repairing codes from [18]: The object **o** is split into four pieces, and *xor*-ed combinations of these pieces are generated. Two such pieces are stored at each node, over a group of five nodes, so that contacting any two nodes is adequate to reconstruct the original object. Furthermore, *systematic* pieces are available in the system, which can be downloaded and just appended together to reconstruct the original data.

Figure 6: Self-repairing codes.

Figure 6(b) shows another example illustrating how the fan-in can be varied to achieve different repair bandwidths while using SRC. If a node, say N_5, fails, then the lost data can be reconstructed by contacting a subset of live nodes. Two different strategies with different fan-ins $d = 2$ and $d = 3$ and correspondingly different total bandwidth usage have been shown to demonstrate some of the flexibilities of the regeneration process.

Notice that the optimal storage-bandwidth trade-off of regenerating codes does not apply here,

since the constraint $d > k$ is relaxed. Thus better trade-off points in terms of total bandwidth usage for a repair can also be achieved (not illustrated here, see [18] for details).

Recall that if a node can be repaired with $d < k$ other nodes then there exist dependencies among them. The data object can be recovered only out of k independent encoded pieces, and hence when the k nodes include $d+1$ nodes with mutual dependency, then the data cannot be recovered from them. LRCs however allow recovery of the whole object using many specific combinations of k encoded fragments. From the closed form and numerical analyses of [17] and [18], respectively, one can observe that while there is some deterioration of the static resilience[7] with respect to MDS codes of equivalent storage overhead, the degradation is rather marginal. This can alternatively be interpreted as that for a specific desired value of fault-tolerance, the storage overhead for using LRC is negligibly higher than MDS codes. An immediate caveat emptor that is needed at this juncture is that, the rates of the known instances of locally repairable codes in general, and self-repairing codes in particular, are pretty low, and much higher rates are desirable for practical usage. The static resilience of such relatively higher rate locally repairable codes, if and when such codes are invented, will need to be revisited to determine their utility. Such trade-offs are yet to be fully understood, though some early works have recently been carried out [10, 13].

6 Cross-Object Coding

All the coding techniques we have seen so far address the repairability problem at the granularity of isolated objects that are stored using erasure coding. However, a simple heuristic of superimposing two codes, one over individual objects, and another across encoded pieces from multiple objects [4] as shown in Figure 7, can provide good repairability properties as well.

Consider m objects $O_1, \ldots, O_m$ to be stored. For $j = 1, \ldots, m$, object O_j is erasure encoded into n encoded pieces $e_{j1}, \ldots, e_{jn}$, to be stored in mn distinct storage nodes. Additionally, *parity groups* formed by m encoded pieces (with one encoded piece chosen from each of the m objects) can be created, together with a parity piece (or xor), where w.l.o.g, a parity group is of the form $e_{1l}, \ldots, e_{ml}$ for $l = 1, \ldots, n$, and the parity piece p_l is $p_l = e_{1l} + \ldots + e_{ml}$. The parity pieces are then stored in additional n distinct storage nodes. Such an additional redundancy is akin to RAID-4.

This code design, called *Redundantly grouped coding* is similar to a two-dimensional product code [8] in that the coding is done both horizontally and vertically. In the context of RAID systems, similar strategy has also been applied to create intra-disk redundancy [5]. The design objectives here are somewhat different, namely: (i) the horizontal layer of coding primarily achieves fault-tolerance by using an (n, k) erasure coding of individual objects, while (ii) the vertical single parity check code mainly enables cheap repairs (by choosing a suitable m) by creating RAID-4 like parity of the erasure encoded pieces from different objects.

The number of objects m that are cross-coded indeed determines the fan-in for repairing isolated failures independently of the code parameters n and k. If $m < k$, it can be shown that the probability that more than one failure occurs per column is small, and thus repair using the parity bit is often enough - resulting in cheaper repairs, while relatively infrequently repairs may have to be performed using the (n, k) code. The choice of m determines trade-offs between repairability, fault-tolerance and storage overheads which have been formally analyzed in [4]. Somewhat surprisingly, the analysis demonstrates that for many practical parameter choices, this cross-object

[7] Static resilience is a metric to quantify a storage system's ability to tolerate failures based on its original configuration, and assuming that no repairs to compensate for failures are carried out.

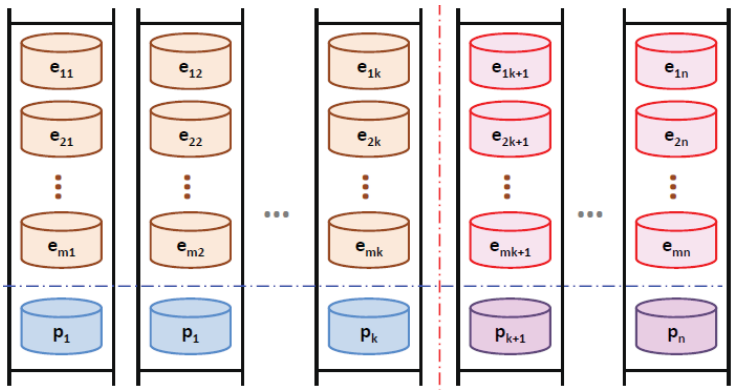


Figure 7: Redundantly grouped coding: a horizontal layer of coding is performed on each object using an (n, k) code, while a parity bit is computed vertically across m objects, where m is a design parameter.

coding achieves better repairability while retaining equivalent fault-tolerance as maximum distance separable erasure codes incurring equivalent storage overhead.

Such a strategy also leads to other practical concerns as well as opportunities, such as the issues of object deletion or updates, which need further rigorous investigation before considering them as a practical option.

7 Preliminary comparison of the codes

The coding techniques presented in this paper have so far undergone only partial evaluation and benchmarking, and more rigorous evaluation of even the stand-alone approaches is ongoing work for most. Thus, it is somewhat premature to provide results from any comparative study, though some preliminary works on the same have also recently been carried out [22] taking into consideration realistic settings where multiple objects are collocated in a common pool of storage nodes, and multiple storage nodes may potentially fail simultaneously, all creating interferences between the different repair operations competing for the limited and shared network resources. Instead, we give one example of a theoretical result by considering the repair bandwidth per repair in the presence of multiple failures for some of these codes, and we provide an overview of what a system designer may expect from all these codes in Table-1. We further enumerate several other metrics that need to be studied to better understand their applicability.

One would not allow in practice failures to accumulate indefinitely, and instead a regeneration process will have to be carried out. If this regeneration is triggered when precisely x out of the n storage nodes are still available, then the total bandwidth cost to regenerate each of the $n-x$ failed nodes is depicted in Figure 8. Note that delayed repair where multiple failures are accumulated may be a design choice, as in P2P systems with frequent temporary outages, or an inevitable effect of correlated failures where multiple faults accumulate before the system can respond.

For locally repairable codes such as SRC the repairs can be done in sequence or in parallel, denoted γ_{seq} and γ_{prl} respectively in the figure. This is compared with MDS erasure codes (γ_{eclazy}) when the repairs are done in sequence, as well as with RGC codes at MSR point (γ_{MSRGC}) for a

Code Family	Main Design Objective	MDS	Fan-in d	Simultaneous Repairs	Bandwidth per Repair
EC/RS	noisy channels	yes	k	$\leq n-k$	$1+\frac{k-1}{t}$
RGC [6]	min. repair bandwidth	yes	$\geq k$	1	$\frac{d}{d-k+1}$
CRGC [27, 16]	min. repair bandwidth	yes	$\geq k$	t	$\frac{d+t-1}{d-k+1}$
SRC [17]	min. fan-in	no	2	$\leq \frac{n-1}{2}$	2
Pyramid [14]	localize repairs (probabilistically)	no	depends	depends	depends
Hierarchical [7]	localize repair (probabilistically)	no	depends	depends	depends
Cross-object coding [4]	constant repair fan-in (probabilistically)	no	depends: m or k	depends	depends: m or k

Table 1: Code design overview: We specify 'depends' to some of the metrics, to signify that the corresponding value depends on the specific fault pattern and possible code parameters. A case in point being general Pyramid or Hierarchical codes. They have several parameters, the details of which we have not delved into in this high level survey. But, one can already note from the simple Hierarchical code example discussed in this paper that parallel repairs may be possible sometimes (for instance when $\mathbf{o}_1$ and $\mathbf{o}_4$ fail simultaneously), while it may have to be done in a serialized manner (for example, if $\mathbf{o}_1$ and $\mathbf{o}_1+\mathbf{o}_2$ fail simultaneously), while it may be impossible in other scenarios (such as when $\mathbf{o}_1$ and $\mathbf{o}_2$ fail simultaneously). The other aspects of repair likewise may vary, depending on failure pattern as well as code parameters.

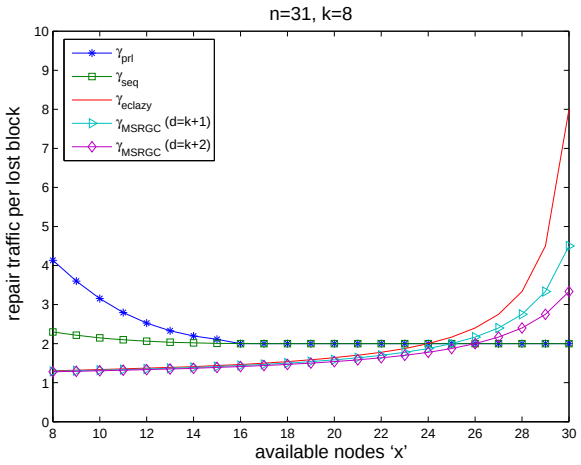


Figure 8: Comparison among traditional erasure codes, regenerating codes and self-repairing codes (derived theoretically in [17]): Average traffic normalized with B/k per lost block for various choices of x (B is the size of the stored object) for (n=31,k=8) encoding schemes. For parallel repairs using erasure codes the traffic is $k=8$ (not shown). The SRC code parameters are denoted as SRC(n,k).

few choices of d. The bandwidth need has been normalized with the size of one encoded fragment. We notice that for up to a certain point, self-repairing codes have the least (and a constant of 2) bandwidth need for repairs even when they are carried out in parallel.

For larger number of faults, the absolute bandwidth usage for traditional erasure codes and regenerating codes is lower than that of self-repairing codes. However given that erasure codes and regenerating codes need to contact k and $d \geq k$ nodes respectively, some preliminary empirical studies have shown the regeneration process for such codes to be slow [21] which can in turn make the system vulnerable. In contrast, because of an extremely small fan-in $d = 2$, self-repairing codes can support fast and parallel repairs [17] while dealing with a much larger number of simultaneous faults. Comparison with some other codes such as hierarchical and pyramid codes has been excluded here due to the lack of necessary analytical results, as well as the fact that the different encoded pieces have assymetrical importance, and thus, just the number of failures does not adequately capture the system state for such codes.

Given that repair processes run continuously or as and when deemed necessary, the static resilience is not the most relevant metric of interest for storage system designers. Often, another metric, namely *mean time to data loss* (MTTDL) is used to characterize the reliability of a system. MTTDL is determined by taking into account the cumulative effect of the failures along with that of the repair processes. For the novel codes discussed in this manuscript, such study of MTTDL is yet to be carried out in the literature. However, a qualitative remark worth emphasizing is that, precisely because of the better repair characteristics such as fast repairs, some of these codes are likely to improve MTTDL significantly. Whether the gains outweigh the drawbacks, such as the lack of MDS property (and consequent poorer static resilience), is another open issue.

8 Concluding remarks

There is a long tradition of using codes for storage systems. This includes traditional erasure codes as well as turbo and low density parity check codes (LDPC) coming from communication theory, rateless (digital fountain and tornado) codes originally designed for content distribution centric applications, or locally decodable codes emerging from the theoretical computer science community to cite a few. The long believed mantra in applying codes for storage has been '*the storage device is the erasure channel*'.

Such a simplification ignores the maintenance process in NDSS for long term reliability. This realization has led to a renewed interest in designing codes tailor-made for NDSS. This article surveys the major families of novel codes which emphasize primarily better repairability. There are many other system aspects which influence the overall performance of these codes, that are yet to be benchmarked. This high level survey is aimed at exposing the recent theoretical advances providing a single and easy point of entry to the topic. Those interested in further mathematical details depicting the construction of these codes may refer to a longer and a more rigorous survey [20] in addition to the respective individual papers.

Acknowledgement

A. Datta's work was supported by MoE Tier-1 Grant RG29/09. F. Oggier's work was supported by the Singapore National Research Foundation under Research Grant NRF-CRP2-2007-03.

References

[1] Apache.org. HadoopFS-RAID. http://wiki.apache.org/hadoop/HDFS-RAID, 2012.

[2] B. Calder, et al., "Windows Azure Storage: a highly available cloud storage service with strong consistency" Twenty-Third ACM Symposium on Operating Systems Principles, SOSP 2011.

[3] Y. Cassuto, J. Bruck, "Low-Complexity Array Codes for Random and Clustered 4-Erasures", IEEE Transactions on Information Theory, 01/2012.

[4] A. Datta and F. Oggier, "Redundantly Grouped Cross-object Coding for Repairable Storage", Asia-Pacific Workshop on Systems, APSys 2012.

[5] A. Dholakia, E. Eleftheriou, X-Y. Hu, I. Iliadis, J. Menon, K.K. Rao, "A new intra-disk redundancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors", ACM Transactions on Storage, 2008.

[6] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright and K. Ramchandran, "Network Coding for Distributed Storage Systems" IEEE Transactions on Information Theory, Vol. 56, Issue 9, Sept. 2010.

[7] A. Duminuco, E. Biersack, "Hierarchical Codes: How to Make Erasure Codes Attractive for Peer-to-Peer Storage Systems" , Eighth International Conference on In Peer-to-Peer Computing, P2P 2008.

[8] P. Elias, "Error-free coding", Transactions on Information Theory, vol. 4, no. 4, September 1954.

[9] S. Ghemawat, H. Gobioff, S-T. Leung, "The Google file system", ACM symposium on Operating systems principles, SOSP 2003.

[10] P. Gopalan, C. Huang, H. Simitci, S. Yekhanin, "On the locality of codewords symbols", Electronic Colloquium on Computational Complexity (ECCC), vol. 18, 2011.

[11] K. M. Greenan, X. Li, J. J. Wylie, "Flat XOR-based erasure codes in storage systems: constructions, efficient recovery, and tradeoffs". IEEE conference on Massive Data Storage, 2010.

[12] J. L. Hafner, "WEAVER codes: highly fault tolerant erasure codes for storage systems", 4th conference on USENIX Conference on File and Storage Technologies, FAST 2005.

[13] H. D. L. Hollmann, "Storage codes - coding rate and repair locality", International Conference on Computing, Networking and Communications, ICNC 2013.

[14] C. Huang, M. Chen, and J. Li, "Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems", Sixth IEEE International Symposium on Network Computing and Applications, NCA 2007.

[15] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Lin, S. Yekhanin, "Erasure Coding in Windows Azure Storage", USENIX conference on Annual Technical Conference, USENIX ATC 2012.

[16] A.-M. Kermarrec, N. Le Scouarnec, G. Straub, "Repairing Multiple Failures with Coordinated and Adaptive Regenerating Codes", The 2011 International Symposium on Network Coding, NetCod 2011.

[17] F. Oggier, A. Datta, "Self-repairing Homomorphic Codes for Distributed Storage Systems", The 30th IEEE International Conference on Computer Communications, INFOCOM 2011. Extended version at `http://arxiv.org/abs/1107.3129`

[18] F. Oggier, A. Datta, "Self-Repairing Codes for Distributed Storage – A Projective Geometric Construction", IEEE Information Theory Workshop, ITW 2011.

[19] F. Oggier, A. Datta, "Homomorphic Self-Repairing Codes for Agile Maintenance of Distributed Storage Systems", `http://arxiv.org/abs/1107.3129`

[20] F. Oggier, A. Datta, "Coding Techniques for Repairability in Networked Distributed Storage Systems", `http://sands.sce.ntu.edu.sg/CodingForNetworkedStorage/pdf/longsurvey.pdf`, September 2012.

[21] L. Pamies-Juarez, E. Biersack, "Cost Analysis of Redundancy Schemes for Distributed Storage Systems", arXiv:1103.2662, 2011.

[22] L. Pamies-Juarez, F. Oggier, A. Datta, "An Empirical Study of the Repair Performance of Novel Coding Schemes for Networked Distributed Storage Systems", arXiv:1206.2187, 2012.

[23] D. A. Patterson, G. Gibson, R. H. Katz "A case for redundant arrays of inexpensive disks (RAID)" ACM SIGMOD International Conference on Management of Data, 1988.

[24] K. V. Rashmi, N. B. Shah, P. Vijay Kumar, K. Ramchandran, "Explicit Construction of Optimal Exact Regenerating Codes for Distributed Storage", Allerton 2009.

[25] A. S. Rawat, S.Vishwanath, "On Locality in Distributed Storage Systems", IEEE Information Theory Workshop, ITW 2012.

[26] I. S. Reed, G. Solomon, "Polynomial Codes Over Certain Finite Fields", Journal of the Society for Industrial and Appl. Mathematics, no 2, vol 8, SIAM, 1960.

[27] K. W. Shum, "Cooperative Regenerating Codes for Distributed Storage Systems", IEEE International Conference on Communications, ICC 2011.

[28] K. W. Shum, Y. Hu, "Cooperative Regenerating Codes", arXiv:1207.6762, 2012.

[29] I. Tamo, Z. Wang, J. Bruck, "MDS Array Codes with Optimal Rebuilding", arXiv:1103.3737, 2011.