An Overview of Codes Tailor-made for Networked Distributed Data Storage

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

***Abstract*—The continuously 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 needs of distributed networked storage systems, trying to reach the best compromise among storage space efficiency, fault tolerance, and maintenance overheads. Four families of codes tailor-made for distributed settings, namely - pyramid, hierarchical, regenerating and self-repairing codes - are presented at a high level, emphasizing the main ideas behind each of these codes, and discussing their pros and cons, before concluding with a quantitative comparison among them. This survey deliberately excluded technical details for the codes, nor does it provide an exhaustive summary of the numerous works. Instead, it provides an overview of the major code families in a manner easily accessible to a broad audience, by presenting the big picture of advances in coding techniques for distributed storage solutions.**

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

I. Introduction

Similarly to the role granaries played in the march of agricultural civilizations of yore, massive data stores form a cornerstone in today's knowledge based societies, apart being a key enabler of the cloud paradigm. The volume of data to be stored is huge, and increasing rapidly. As of June 2011, a study sponsored by the information storage company EMC estimates that the world's data is more than doubling every 2 years, growing faster than Moore's Law, and reaching 1.8 zettabytes of data to be stored in 2011.[1] This involves various kinds of digital data continuously being generated by individuals as well as business and government organizations, who all need scalable solutions to store such data reliably and securely over time. Quoting the EMC study, next decade will see enterprises managing 50× more data, and handling 75× more files.

A very basic yet critical concern in storage systems is ensuring that the data is not irrevocably lost due to device failures. Redundancy is key to ensure fault tolerance, which can be achieved using replication, i.e., storing multiple copies of the same data. For instance, a common practice is to keep at least three copies of the data (called three way replication), the rationale being that if, or rather, when, one copy of the data is lost, the second copy is used to regenerate the first one, and hopefully, not both the two other copies are lost before 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.

Imagine if one could come up with a way to store the same amount of data, reliably, but using only half the needed storage space. That would imply a reduction of the storage infrastructure by half. The cost for such infrastructure should be estimated not only in terms of the raw cents per gigabyte metric, but also in terms of the physical space needed - the corresponding cost of real estate, as well as construction, operational and maintenance costs of the data center. The numbers can in fact be staggering, as we move towards a world witnessing data centers which equal medium sized cities in terms of power consumption. A US Environmental Protection Agency report of 2007[2] indicates that the US used 61 billion kilowatt-hours of power for data centers and servers in 2006. That is 1.5 percent of all US electricity use, and it cost the companies that paid those bills more than $4.5 billion. Such observations have led to a wide-scale realization of the energy and environmental implications of large-scale data centers, and consequent efforts to build 'carbon neutral data centers'.

There are different ways one could imagine reducing the storage space, starting from the physical media, which has witnessed a continuous shrinking of physical space per unit of data. This article questions storage space at a different layer, that of the trade-of between fault tolerance, and the level of efficiency in storage space utilization in large scale distributed data storage systems.

The tension between more fault tolerance versus less storage overhead is well felt through the evolution of RAID (redundant array of inexpensive/independent disks). RAID is a technology that enables storage virtualization by distributing data across multiple disk drives that act as a logical unit, which has now been used for over two decades. The first RAID system, RAID 0, offered good performance and additional storage by exactly storing data across multiple drives, so that the disk head can read more data in a single move, a technique referred to as striping. Its main drawback was that it had no fault tolerance, and a single disk failure could cause the loss of the entire stored data. This was remedied in RAID 1, where replication (mirroring) was introduced, but at the cost of storage overhead. The subsequent RAID 2,3,4 and 5 exchanged replication against coding, by opting for parity bits, possibly the simplest coding technique, also known as Hamming parity codes: a parity bit is just the sum of all bits across one drive. This allowed to retrieve the stored data, even

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

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

in the event of a failure. While the original RAID was designed with stand alone computer systems in mind, dRAID -d stands for distributed- has since been widely adopted in networked storage systems, including in storage area networks. The need for more storage capacity has however pushed for more storage capacity per disk, more disks per system, and consequently the use of cheaper (less reliable) disks, resulting in the demand for more fault tolerance and thus in a new quest for better coding techniques for (d)RAID, ideally as ”simple” as parity codes, but at the same time capable of tolerating more failures. A notable example of codes proposed to suit the peculiarites of (d)RAID systems is the family of weaver codes [3], which can deal with up to 4 faults.

Finding codes offering the best trade-off between amount of redundancy and fault tolerance is no new question to a coding theorist. In communication theory, data is sent over a noisy channel, and coding is used to append redundancy to the transmitted signal, to help the receiver recover the intended message, even when alterated by noise, possibly erasures. The length of the data, or information symbols, is usually denoted by k, and $n - k$ symbols of redundancy are computed to obtain a codeword of length n. The amount of information k that a code can carry and its correction capability are linked, and this dependency is characterized by the so-called Singleton bound. The best $EC(n, k)$ codes, meeting the bound, are called maximum distance separable (MDS) codes. The celebrated Reed-Solomon codes [9] are an instance of such codes. MDS erasure codes, which are capable of recovering erased information symbols, can handle up to $n - k$ erasures (see Figure 1).

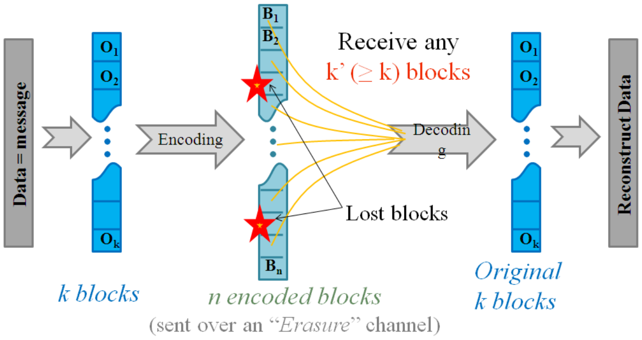


Fig. 1. Coding for erasure channels: a message of k symbols is encoded into a codeword of length n, before being sent over an erasure channel. As long as not more than $n - k$ symbols are lost (erased), the receiver can decode the message.

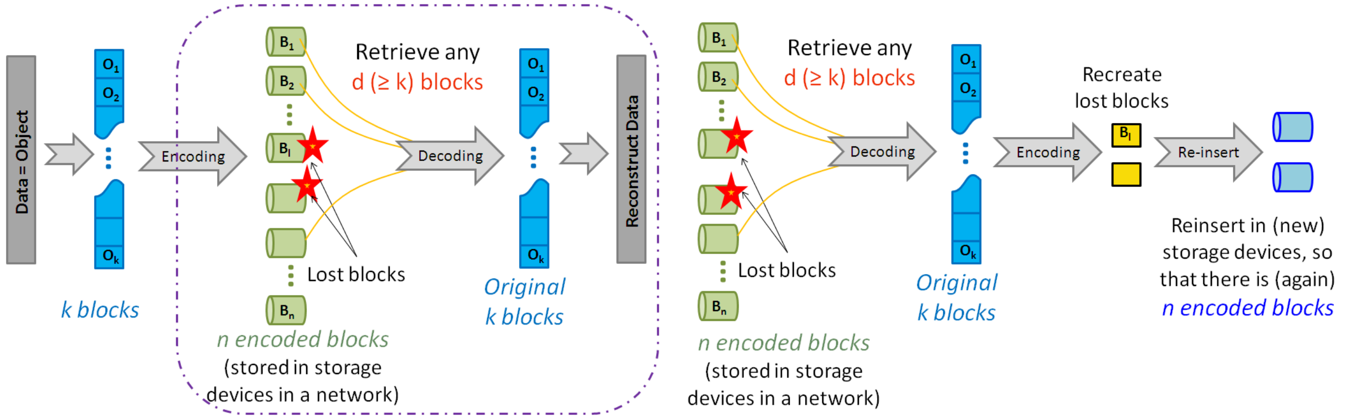
Now armed with MDS codes, it seems at first clear how to upgrade RAID: replace the Hamming code by a more powerful MDS code, thus immediately guaranteeing the best possible fault tolerance at the lowest cost. Things are however not that straightforward: because erasure codes have been designed and optimized to deal with lossy communication channels, their sole objective is to recover a transmitted message at the receiver. There is no notion of reliability over time, placement of the data, or read/write operations to optimize. Furthermore, the complexity of the mathematical operations involved, for example in Reed-Solomon codes, necessitates dedicated hardware, making these codes not suitable for RAID systems.

There is one example of storage application where MDS codes have been successfully used though, providing orders of magnitude more efficiency/reliability: that of the CD/DVD. When there is a scratch on a CD, the information stored in those portions are lost (erasures), but depending on the degree of such losses, the disc can still be read, thanks to the incorporation of Reed-Solomon codes. That is the exception that proves the rule.

II. Networked Distributed Storage Systems

MDS codes have also been considered in the context of networked distributed storage settings. If some storage nodes fail, or become inaccessible, e.g., due to communication faults or power outage, the data should remain available. By considering a node failure as an erasure, the data can be encoded with an erasure code, which will then provide fault tolerance, in that the stored data can be retrieved even in the presence of faulty nodes, as long as the number of failures does not exceed the capability of the code. Such fault tolerance is achieved using a much smaller overhead than three way replication. Alternatively put, for same storage overhead, a much higher level of fault-tolerance can be achieved. For instance, if the probability of individual storage devices to fail is 0.1, then with the use of a $(9, 3)$ MDS erasure code, the probability of losing the data is roughly $3*10^{-6}$, but if three way replication scheme is used instead, the probability of losing the data will be as high as 10^{-3}.

There is however a fundamental difference between a CD or a communication channel on the one hand, and a networked storage system on the other hand. Unlike in the CD, where a scratch cannot be removed, or in a communication channel where the receiver just has to deal with what it receives, in a networked storage system, the failed storage nodes can be replaced with new ones, and in fact this maintenance process is essential for durability and availability of the data in the long run. Specifically, the loss of redundancy needs to be replenished, by recreating new redundancy at other existing or new storage devices in the system. Such ability to replace failed storage devices, and regenerate lost redundancy enables the deployment of a reliable storage system comprising many storage nodes, and thus scaling to extremely large volumes of data - it also makes regeneration of lost data imperative. As already mentioned above, MDS erasure codes were not designed to address reliability over time, in particular they were not made to address maintenance/repair issues. When a data block encoded by an MDS erasure code is lost and needs to be recreated, one would 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 blocks), even in order to recreate a single encoded block, as illustrated on Figure 2.



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

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

Fig. 2. Erasure coding for distributed networked storage: 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 of node failures is illustrated on the right.

The particularly good behavior exhibited by MDS erasure codes with respect to storage overhead vs. fault-tolerance trade-off has remained appealing over time, encouraged by the CD success story, and on-going efforts have been witnessed, to retain their benefits while mitigating their drawbacks, such as lack of repair mechanism and complexity. Companies such as CleverSafe[3] and Wuala[4] claim to use Reed-Solomon like codes. Within the academic circle, erasure codes have been intensively studied in the peer-to-peer storage systems related research, and have regained interest within the coding community, which has started to take on the challenges offered by storage systems.

These challenges can tentatively be summarized as follows: design erasure codes which carry out repair efficiently, in terms of (i) bandwidth, which is a very scarce resource in data-center networks, (ii) computation and energy costs, but also (iii) repair time, since delay in the repair process may leave the system vulnerable to further faults leading to irretrievable loss of data. This is besides spelling out many of the system issues, such as placement of the data, number of disk accesses, management of the meta-information to coordinate the network, etc.

With respect to the repair process, there are arguably two extreme points possible in the design-space of codes tailor-made for distributed networked storage:

(i) Minimize the absolute amount of data transfer needed to recreate the lost data from one node. *Regenerating codes* [1] 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 at least k live nodes for each repair. Regenerating codes, like MDS erasure codes, allow data retrievability from any arbitrary set of k nodes. Collaborative regenerating codes [10], [5] are a generalization allowing to deal with multiple faults. Actual explicit constructions of regenerating codes are currently sparse (see [11] for an example of non-collaborative codes at the two extremes of the trade-off curve).

(ii) Minimize the number of nodes to be contacted for repairing one node failure. Recently proposed *self-repairing codes* [7] achieve this optimal, by allowing one repair while contacting only two nodes, as long as less than half of the nodes have failed. This is also true for multiple faults, assuming a proper repair scheduling. This is at the price of sacrificing the MDS property, though a *static resilience analysis* of self-repairing codes has shown that object retrieval is little impaired by the loss of the MDS property. It might be worth noting that other codes, such as weaver codes [3], or some instances of pyramid codes [4] which will be discussed below have also traded the MDS property for better repair. As will be emphasized again later on, being able to repair by contacting less than k nodes and the MDS property are mutually exclusive, and code and system designers need to choose one or the other.

There are several other codes which fall somewhere 'in between' these extremes, and have been tailor-made with networked distributed storage applications in mind. Prominent among these are hierarchical and pyramid codes, that we will summarize first, before taking closer look at regenerating and self-repairing codes.

III. Pyramid and Hierarchical Codes

In order to alleviate the drawbacks of erasure codes with respect to applications to storage systems, iterated erasure code constructions have been proposed (*pyramid codes* [4] and independently, *hierarchical codes* [2]), where subgroups of the code itself are essentially basic erasure codes. We note here that the specific instances discussed in [4] and [2] are slightly different, however, since the underlying principle is

[3] http://www.cleversafe.com/

[4] http://www.wuala.com/

analogous, we present them as such. Figure 3 illustrates the basic ideas, which we summarize next.

Consider first a data object of size $s \cdot k'$, partitioned into s subgroups, each comprising k' of the unencoded blocks. An $EC(n', k')$ code is used to generate $n' - k'$ *local redundancy* blocks within each such subgroup. A further r' *global redundancy* blocks are generated using all the $s \cdot k'$ unencoded blocks, and some coding scheme. Thus, a code group is formed, which effectively is a code mapping $s \cdot k'$ unencoded blocks into $s \cdot n' + r'$ encoded blocks.

Note that the local redundancy blocks can be used to repair any losses within a subgroup - thus requiring to access a number of blocks (much) less than the equivalent to recreate the whole object, while the global redundancy blocks provide further protection if there are 'too many' failures within a single subgroup.

If the data object is instead of size $g \cdot s \cdot k'$, then the basic code group working on objects of size $s \cdot k'$ as described above can be repeated, to build another level of the hierarchy - and further 'global redundant' blocks can be created using all the original $g \cdot s \cdot k'$ blocks. This process can in fact be iterated for a multi-hierarchical or pyramid code, as the names suggest. At each level of such a hierarchy, 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 - if even that is not adequate to carry out the repairs, then one will need to move further up the pyramid, and so on.

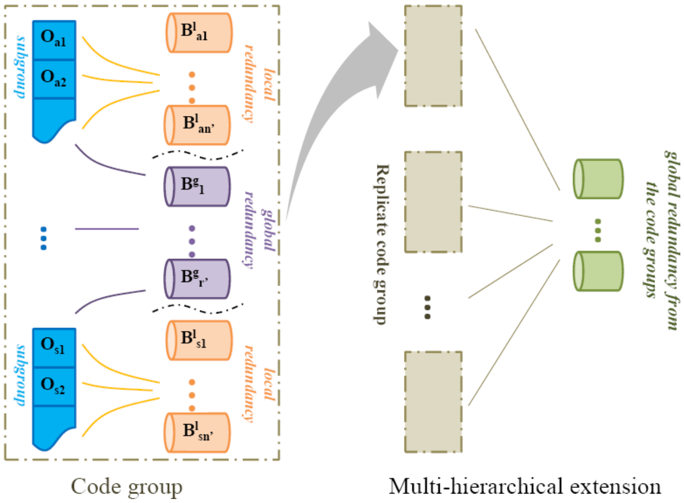


Fig. 3. Pyramid codes

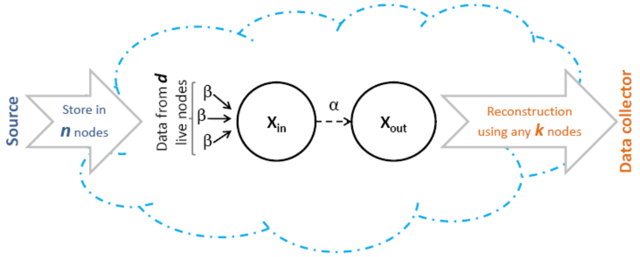
With respect to two design criteria that we chose to investigate, namely repair bandwidth and number of nodes contacted for repair, it is clear that an advantage of both pyramid and hierarchical codes is the reduction in the number of nodes that need to be contacted (and consequently, a reduction in bandwidth usage) to recreate a lost encoded block, though no optimality can be claimed, neither w.r.to the nodes contacted nor bandwidth used. The (possibly) biggest disadvantage is the lack of symmetry in the structure, meaning that some encoded blocks are more or less important than others, which makes it difficult if not impossible to do a thorough resilience analysis (affecting the understanding of the code effectiveness). Furthermore, more complicated algorithms (either for reconstruction or repair) and system design to utilize the codes in practice are thus required. Note that contrarily to hierarchical codes, which were intended to improve repair in distributed networked storage systems, pyramid codes were meant to improve read performance, by trading storage space efficiency for access efficiency (in particular, some instances of pyramid codes are not MDS).

Coinciding these works, researchers from the network coding community started to study 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. This initiative has culminated in a new family of codes, broadly known as regenerating codes, which we discuss next.

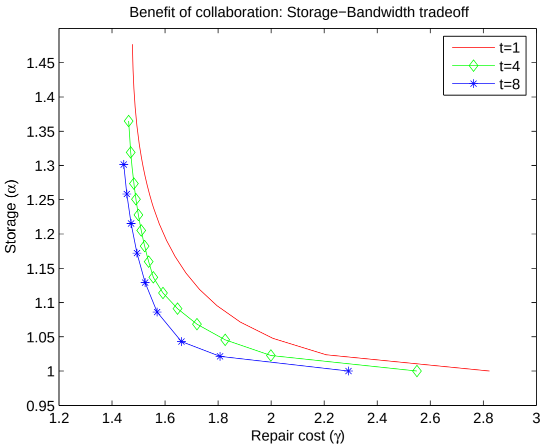
IV. Regenerating codes

The repair of lost redundancy in a storage system can be abstracted as an *information flow graph* [1] as follows. Initially, the data object to be stored is processed and divided in smaller encoded blocks to be stored across n storage nodes, such that obtaining any k of these n encoded blocks would suffice to recreate the original data (this is essentially the MDS property). When some storage nodes fail, the corresponding stored data is lost. New nodes, at which some new information will then need to be regenerated, will contact the existing live nodes to do so. Over time, many of the original nodes may become unavailable, and these existing live nodes themselves may be nodes which had joined subsequently and had carried out regeneration themselves. Thus, the regeneration process essentially needs to ensure two things: (i) independently of the sequence of failures, the regeneration is done in a manner, which allows further future regenerations while always guaranteeing that (ii) any of the k live nodes possess enough information among them to permit reconstruction of the original data object.

Consider that each storage node stores α amount of data. Consider also that when a node fails, the new node contacts $d > k$ live nodes and downloads β amount of data from each contacted node. We will call d the fan-in for regeneration. Each storage node X^i can be modeled as two logical nodes X^i_{in} and X^i_{out}, which are connected with a directed edge $X^i_{in} \rightarrow X^i_{out}$ with a weight α representing the storage capacity of the node. The regeneration process is modeled with directed edges of weight β from the X_{out} nodes of contacted live nodes to the X_{in} node of the new node at which the lost data is being regenerated. The original placement of the data distributed over n nodes can similarly be modeled as directed edges of weight α from the source (data owner) to the original storage nodes. Finally, the data reconstruction problem can be abstracted with edges of weight α to represent the destination (data accessor) downloading data from arbitrary k live storage nodes. Note that the edges from source, and to destination could also be modeled with edges with larger capacities (e.g., ∞), but it does not matter - since the maximum meaningful information that is transferred over such links correspond to the storage capacity. Figure 4(a) depicts an abstraction for the information flow graph, where each storage node is modeled as two virtual nodes, namely X_{in} which collects β amount of information from arbitrary d live nodes in the system, while storing a maximum of α amount of information at X_{out}, which is accessed by any data collector contacting the storage node. 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 data to be reconstructible at the destination, this flow needs to be at least as large as the size of the original object.



(a) Regeneration process for one failed node using network coding principles: A max-flow min-cut analysis yields the feasible values for storage capacity α and amount of information obtained from each live node contacted β in terms of the number of nodes contacted d and code parameters n, k, where $d \geq k$.



(b) Based on the max-flow min-cut analysis, one can determine the trade-off curve for the amount of storage space α used per node in the system, 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.

Fig. 4. The underlying network coding theory inspiring regenerating codes

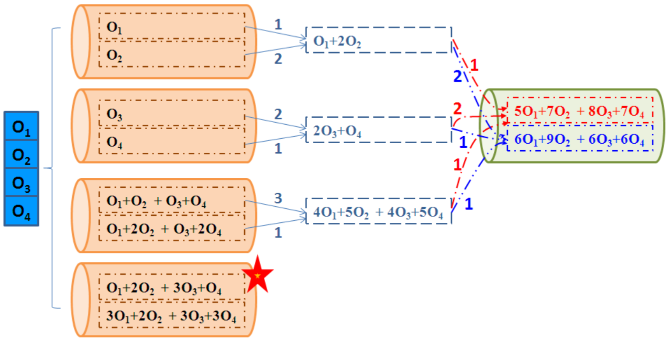
A code which enables the information flow to be actually equal to the object size has been called a regenerating code. Now, given a regenerating code, the natural question to be addressed is, *what are the minimal storage capacity α and bandwidth β needed for repair, as a function of the object size, and parameters k, n, and d?*. 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), 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.

Note that the trade-off analysis only determines what can best be achieved, but in itself does not provide any specific code achieving the same, though the fact that any k nodes must provide enough redundancy for object reconstruction in itself already implies that a regenerating code includes an underlying MDS code. Several codes have since been proposed - most of which operate either at the MSR or MBR points of the trade-off curve. We do not provide an exhaustive list, but summarize the two main classes of regenerating codes instead, each of them illustrated by an example.

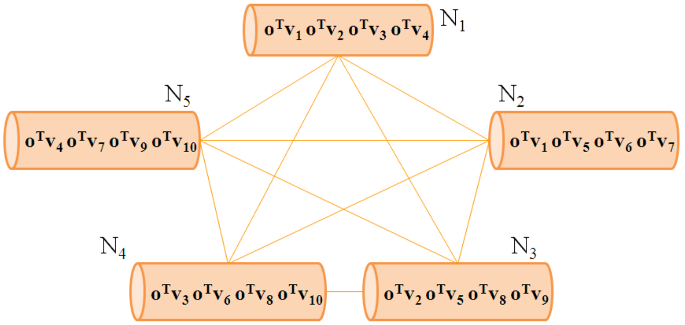
The max-flow min-cut argument determines the amount of data that needs to be transferred in order to regenerate data at a new node, where the regenerated data provides equivalent redundancy to what was provided by the lost data. There is however no constraint or need to regenerate precisely the same (bit wise) data as was lost (see Figure 5 (a)). This observation has been leveraged to propose what is known as *functional regeneration* - where the regenerated data is in fact not the same as that lost, but nevertheless provides equivalent redundancy. In contrast, if the regenerated data is bit wise identical to what was lost, then it is called *exact regeneration*. Exact regeneration is illustrated on Figure 5 (b).

The original analysis was based on the assumption that individual repairs are done independently, or in other words, that regenerating codes are designed to repair one failure. The analysis has since been generalized [5], [10] to show that in case of multiple faults, the new nodes carrying out regenerations can furthermore collaborate among themselves to further reduce the overall bandwidth needed per regeneration (Figure 4(b)). Instances of codes for this setting are even sparser than for classical regenerating codes, and up to now, the only code construction proposed incurs a repair bandwidth cost equal to that of erasure codes under lazy repair [10], but distributes the repair load. Optimal code constructions which meet the trade-off curve remains an open issue.

Regenerating codes promise the optimal usage of bandwidth per repair (subject to MDS property of the codes). Some of the proposed codes apply network coding on top of traditional erasure codes and benefit from the mature decoding algorithms for erasure codes. Furthermore, the notion of functional repair provides greater flexibility in code design choices. However, the information flow analysis itself does not suggest any code construction, and the subsequent codes proposed so far are rather restrictive - for example, some works only deal with a single fault, and need to contact all the other $n-1$ nodes. The



(a) An example of functional repair for $k = 2$ and $n = 4$, adapted from [1]: 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 is failing, 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 that lost, but any $k = 2$ of the 4 nodes permit object retrieval.


$o^T v_1\, o^T v_2\, o^T v_3\, o^T v_4$
N_1
N_5
$o^T v_4\, o^T v_7\, o^T v_9\, o^T v_{10}$
N_2
$o^T v_1\, o^T v_5\, o^T v_6\, o^T v_7$
N_4
$o^T v_3\, o^T v_6\, o^T v_8\, o^T v_{10}$
N_3
$o^T v_2\, o^T v_5\, o^T v_8\, o^T v_9$

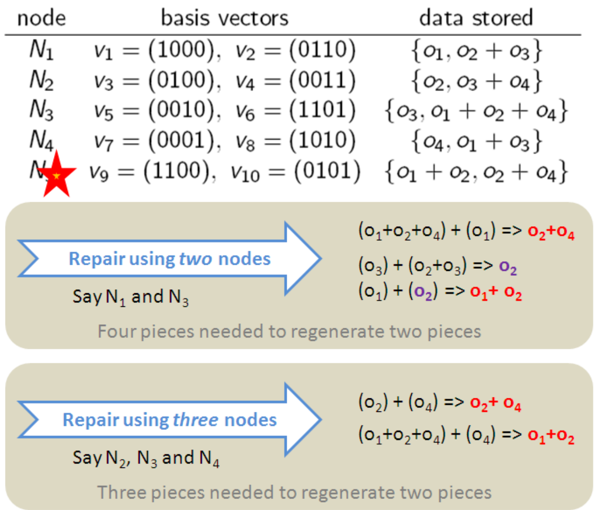
(b) An example of exact repair from [11]: an object $\mathbf{o}$ is encoded by taking its inner product with 10 vectors $\mathbf{v}_1, \ldots, \mathbf{v}_{10}$, to obtain $\mathbf{o}^T \mathbf{v}_i$, $i = 1, \ldots, 10$, as encoded fragments. They are distributed to the 5 nodes $N_1, \ldots, N_5$ as shown. Say, node N_2 fails. A newcomer can regenerate by contacting every node left (shown by the edges), 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.

Fig. 5. Regenerating codes: functional versus exact repair.

value of fan-in d for regeneration has practical implications. Lower fan-in is good, because there are then more alternative choices that can be made, particularly avoiding bottlenecks, and possibly accomplishing repairs in parallel. Furthermore, the algorithmic and system design complexity of using network coding is much larger than even traditional erasure codes, apart the added computational overheads.

V. Self-Repairing Codes

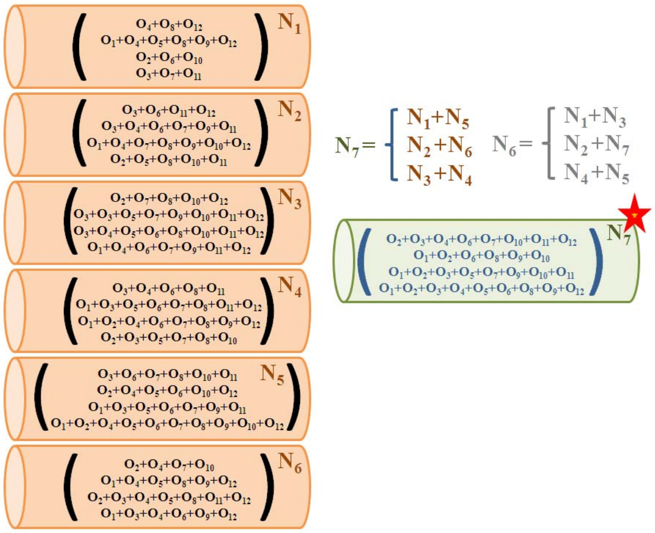
We mentioned that regenerating codes implicitly include an MDS code, and assume that the number of contacted live nodes d is bigger than k. We would like to emphasize that relaxing the constraint $d \geq k$ to allow $d < k$ necessarily implies the loss of the MDS property. Indeed, if a node can be repaired with $d < k$ other nodes, this means that there exists a linear dependency[5] among less than k fragments, so that if the data accessor were to contact k nodes including those $d + 1$ linearly dependent ones, he could not recover



node	basis vectors	data stored
N_1	$v_1 = (1000),\ v_2 = (0110)$	$\{o_1, o_2 + o_3\}$
N_2	$v_3 = (0100),\ v_4 = (0011)$	$\{o_2, o_3 + o_4\}$
N_3	$v_5 = (0010),\ v_6 = (1101)$	$\{o_3, o_1 + o_2 + o_4\}$
N_4	$v_7 = (0001),\ v_8 = (1010)$	$\{o_4, o_1 + o_3\}$
N_5	$v_9 = (1100),\ v_{10} = (0101)$	$\{o_1 + o_2, o_2 + o_4\}$

Repair using *two* nodes
Say N_1 and N_3
$(o_1+o_2+o_4) + (o_1) \Rightarrow o_2+o_4$
$(o_3) + (o_2+o_3) \Rightarrow o_2$
$(o_1) + (o_2) \Rightarrow o_1+ o_2$
Four pieces needed to regenerate two pieces
Repair using *three* nodes
Say N_2, N_3 and N_4
$(o_2) + (o_4) \Rightarrow o_2+ o_4$
$(o_1+o_2+o_4) + (o_4) \Rightarrow o_1+o_2$
Three pieces needed to regenerate two pieces

(a) An example of self-repairing codes from [8]: The object $\mathbf{o}$ is split into four pieces, and *xor*-ed combinations of these pieces are generated. The specific combinations (represented using the basis vectors) to be used is determined based on projective geometric principles. 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. If a node fails, say, N_5, 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.


N_1
$o_4+o_8+o_{12}$
$o_1+o_4+o_5+o_8+o_9+o_{12}$
$o_2+o_6+o_{10}$
$o_3+o_7+o_{11}$
N_2
$o_3+o_6+o_{11}+o_{12}$
$o_3+o_4+o_6+o_7+o_9+o_{11}$
$o_1+o_4+o_7+o_8+o_9+o_{10}+o_{12}$
$o_2+o_5+o_8+o_{10}+o_{11}$
N_3
$o_2+o_7+o_8+o_{10}+o_{12}$
$o_3+o_3+o_5+o_7+o_9+o_{10}+o_{11}+o_{12}$
$o_3+o_4+o_5+o_6+o_8+o_{10}+o_{11}+o_{12}$
$o_1+o_4+o_6+o_7+o_9+o_{11}+o_{12}$
N_4
$o_3+o_4+o_6+o_8+o_{11}$
$o_1+o_3+o_5+o_6+o_7+o_8+o_{11}+o_{12}$
$o_1+o_2+o_4+o_6+o_7+o_8+o_9+o_{12}$
$o_2+o_3+o_5+o_7+o_8+o_{10}$
N_5
$o_3+o_6+o_7+o_8+o_{10}+o_{11}$
$o_2+o_4+o_5+o_6+o_{10}+o_{12}$
$o_1+o_3+o_5+o_6+o_7+o_9+o_{11}$
$o_1+o_2+o_4+o_5+o_6+o_7+o_8+o_9+o_{10}+o_{12}$
N_6
$o_2+o_4+o_7+o_{10}$
$o_1+o_4+o_5+o_8+o_9+o_{12}$
$o_2+o_3+o_4+o_5+o_8+o_{11}+o_{12}$
$o_1+o_3+o_4+o_6+o_9+o_{12}$
$N_7 = \{ N_1+N_5,\ N_2+N_6,\ N_3+N_4$
$N_6 = \{ N_1+N_3,\ N_2+N_7,\ N_4+N_5$
N_7
$o_2+o_3+o_4+o_6+o_7+o_{10}+o_{11}+o_{12}$
$o_1+o_2+o_6+o_8+o_9+o_{10}$
$o_1+o_2+o_3+o_5+o_7+o_9+o_{10}+o_{11}$
$o_1+o_2+o_3+o_4+o_5+o_6+o_8+o_9+o_{12}$

(b) An example of self-repairing codes from [7]: the object $\mathbf{o}$ has length 12, and encoding is done by taking linear combinations of the 12 pieces as shown, which are then stored at 7 nodes. 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, so that for example, a first newcomer can contact first N_1 then N_5 to repair N_7, while the second comer can in parallel contact first N_3 then N_1 to repair N_6.

Fig. 6. Self-repairing codes

[5] Only linear codes are considered.

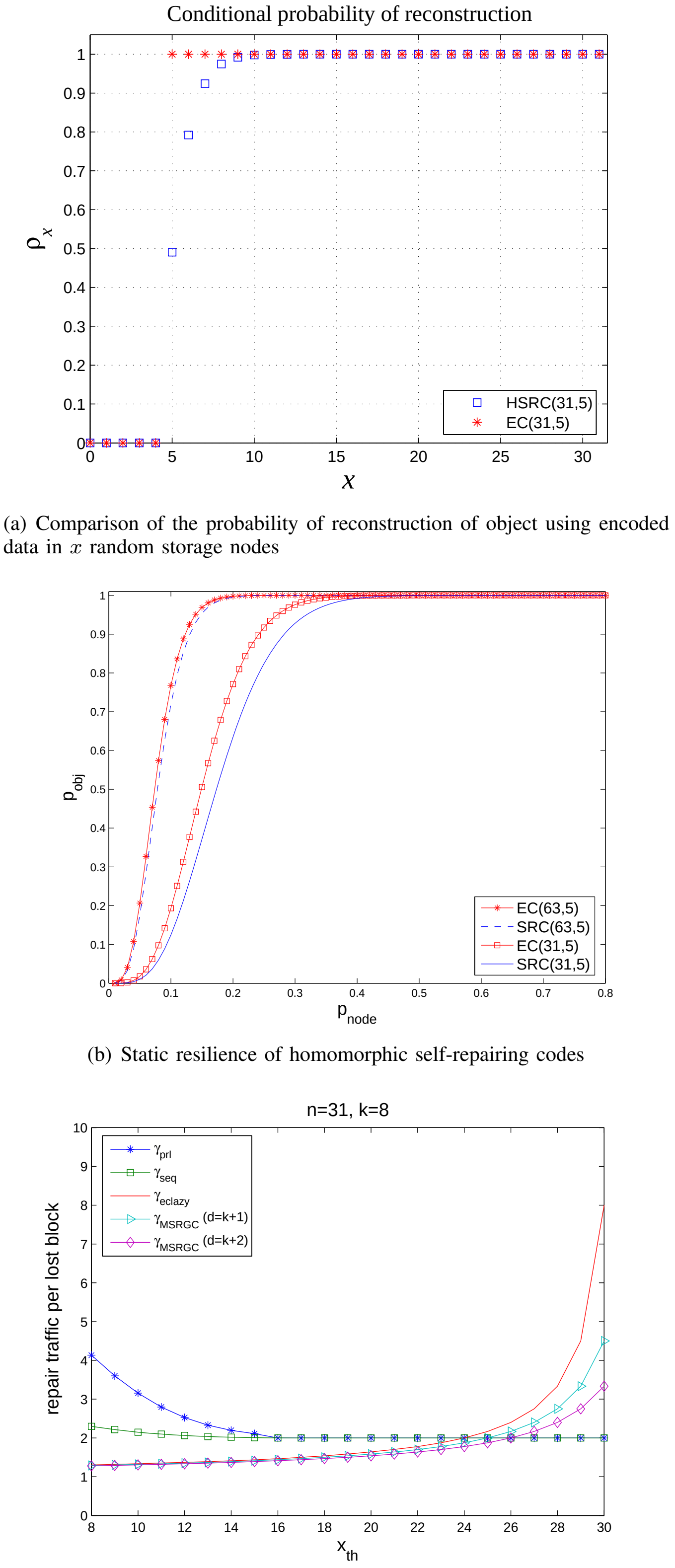
the encoded object. To the contrary of regenerating codes, self-repairing codes [7] work in a regime where $d << k$, in particular d can take most of the time the minimal value of $d = 2$. We note that $d = 1$ is possible only when using replication/repetition code. As just explained, self-repairing codes are not MDS codes, given that they achieve regeneration using $d < k$. To evaluate the loss of the MDS property, a *static resilience* analysis is provided, which for distributed systems, refers to the probability that an object, once stored in the system, will continue to stay available without any further maintenance, even when some individual member nodes of the distributed system become unavailable. The static resilience analysis reveals that there is little difference with MDS codes, especially when the node availability is large (this will be further discussed in next section).

Self-repairing codes however mimic the MDS property in the repair domain instead, namely, they satisfy that repair (of single or multiple faults) can be performed by contacting a fixed number of live nodes, this number depending only on the number of faults, and not on which specific encoded fragment is missing. Note that this is in contrast to pyramid and hierarchical codes. Thus, one may say, self-repairing codes try to combine the good properties of both pyramid codes as well as regenerating codes. Two instances of self-repairing codes are known up to date [7], [8].

That repair can be done by contacting $d = 2$ nodes has the advantage of allowing fast repair, which can be further sped up thanks to parallel repair of multiple faults: since each newcomer may monopolize up to 2 live nodes depending on resource constraints, there is room for several pairs of live nodes to upload repair data at the same time (see Figure 6(a) for an example). 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 can also be achieved.

VI. Quantitative comparison

Self-repairing codes, pyramid and hierarchical codes sacrifice the MDS property in order to achieve regeneration using $d < k$ live nodes. Due to the inherent asymmetry of the role of different encoded pieces in hierarchical and pyramid codes, there is no rigorous analysis to quantify this sacrifice. The trade-offs for homomorphic self-repairing codes have been explicitly determined, and are presented in Figures 7(a) and 7(b). For MDS codes, the possibility to reconstruct the original object by using data stored in any random x storage nodes is binary, it is always possible to reconstruct the data if any k or more storage nodes are used. In contrast, for self-repairing codes, this is probabilistic, as can be observed from the corresponding values of ρ_x shown in 7(a). A natural question to ask is then, if multiple nodes fail and no repairs are carried out, then what is the probability that the data still remains available in the system, i.e., what is the static resilience of the system? Specifically, if only p_{node} fraction out of the n storage nodes are actually available and the failures have happened uniformly randomly, then what is the



(a) Comparison of the probability of reconstruction of object using encoded data in x random storage nodes

(b) Static resilience of homomorphic self-repairing codes

(c) Average traffic normalized with B/k per lost block for various choices of x_{th} (B is the size of the stored object)

Fig. 7. Comparison among traditional erasure codes, regenerating codes and self-repairing codes: static resilience analysis and storage-bandwidth trade-off

probability that the original data is still available. Figure 7(b) compares the static resilience of self-repairing codes with that of MDS codes (which includes traditional erasure codes as well as regenerating codes), and we observe that while there is some deterioration of the resilience, this is rather marginal. Finally, in a real system, one would not allow failures to accumulate indefinitely, and instead a regeneration process will have to be carried out. If this regeneration is triggered when precisely x_{th} out of the n storage nodes are still available, then, the total data transfer needed to regenerate data for each of the $n - x_{th}$ failed nodes is depicted in Figure 7(c). For self-repairing codes, the repairs can be done in sequence or in parallel, denoted using suffices seq and prl, and this is compared with traditional erasure codes (denoted using $eclazy$) when the repairs are done in sequence[6] as well as with RGC codes at MSR point (corresponding to equivalent $\alpha = B/k$ where B is the object size) for few choices of d. The bandwidth need has been normalized with the size of one encoded fragment, i.e., B/k. We notice that for up to a certain point, self-repairing codes have the least (and a constant of 2) bandwidth need for regeneration, and this is even when the regenerations 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 need to contact k nodes and regenerating codes need to contact $d \geq k$ nodes, some preliminary empirical studies have shown the regeneration process for such codes to be slow [6], which can in turn make the system vulnerable. In contrast, self-repairing codes, because of an extremely small fan-in $d = 2$ can support fast and parallel repairs [7] while dealing with a much larger number of simultaneous faults. A more thorough and comparative evaluation of how these various codes perform under realistic circumstances, for instance, taking into account topology connecting the storage nodes, placement strategies, multiple objects, etc. is an open question that the research community is currently engaged in investigating.

VII. Concluding remarks

There has been a long running effort to apply codes for storage systems, and this includes use of traditional erasure codes, as well as other codes such as low density parity check codes (LDPC) coming from communication theory, rateless (digital fountain) 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 has been '*the storage device is the erasure channel*'.

Such a simplification ignores various nuances of distributed networked storage systems, particularly with respect to the maintenance process and long term reliability. This realization has led to a renewed interest in designing codes tailor-made for distributed storage. This article surveys the major families of such novel codes, but does so at a very high level, to make the innovations accessible to a broad range of audience, while also serving as a one-stop starting point for anyone interested in delving deeper.

To conclude, it needs to be noted, that other techniques for data reliability and efficient storage maintenance are also being heavily investigated in the space of storage technology, for instance, snapshots, deduplication, etc. to name a few. How these aspects will be affected if, or, arguably when coding techniques become mainstream for large scale data-storage, are fascinating open questions for the storage technology community in general, and conversely, coding theorists also need to take into account these additional nuances when designing storage centric codes.

References

[1] 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.

[2] 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 , 2008. P2P '08.*

[3] J. L. Hafner, "WEAVER codes: highly fault tolerant erasure codes for storage systems", *4th conference on USENIX Conference on File and Storage Technologies, FAST'05.*

[4] 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, 2007. NCA 2007.*

[5] A.-M. Kermarrec, N. Le Scouarnec, G. Straub, "Repairing Multiple Failures with Coordinated and Adaptive Regenerating Codes", in the proceedings of the 2011 International Symposium on Network Coding (NetCod 2011).

[6] L. Pamies-Juarez, E. Biersack, "Cost Analysis of Redundancy Schemes for Distributed Storage Systems", preprint, available at http://arxiv.org/abs/1103.2662.

[7] F. Oggier, A. Datta, "Self-repairing Homomorphic Codes for Distributed Storage Systems", INFOCOM 2011.

[8] F. Oggier, A. Datta, "Self-Repairing Codes for Distributed Storage – A Projective Geometric Construction", ITW 2011.

[9] 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.

[10] K. W. Shum, "Cooperative Regenerating Codes for Distributed Storage Systems", to appear at *ICC 2011*, available at arXiv:1101.5257v1.

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

[6] For parallel repairs using erasure codes, the traffic is $k = 8$, and has not been shown in the figure.