Combinatorial and Rectangular Layouts as Building Blocks for Local Reconstruction Codes

Thomas Schwarz, SJ*, John Rose, SJ†
*Marquette University, Milwaukee, WI, United States of America
thomas.schwarz@marquette.edu
†Xavier Institute of Engineering, Mahim Causeway, Mahim, Mumbai, Maharashtra, India
johnrose@xavierengg.com

***Abstract*—Modern data centers protect the contents of their data using erasure correcting codes. In recent years, locally repairable codes have been proposed that allow dealing with the most frequent case, a single unavailable disk or a single unreadable sector by only using redundant information in the rack where the failure has been detected. Recently, Pâris proposed a simple method for creating locally repairable layouts where several local layout in different racks are "bundled" together by creating additional inter-rack reliability stripes that provide protection against rack failure and offer additional opportunities for recovery if a local layout fails in its protection task. We propose here a flat rack-internal layout that is more dispersed than a rectangular layout that also provides fast recovery for single failures. The research question is to what extent dispersion improves reliability. As we will see by example, dispersion does provide better robustness, but fails to significantly improve the five year probability that inter-rack codes have not to be used.**

***Index Terms*—Disk Array, Reliability, Flax XOR-code**

I. Introduction

As the world's generation of data increases, its proportion of data stored decreases according to research group IDC. The same source reckons that 19 ZB of data generated between 2017 until 2025 will be stored [1]. Some of this data is mission critical, some of high value and the majority of it will be stored in large data centers. To protect the data there against storage media failure, typically erasure control codes are deployed. Such a code gathers equally sized blocks of data on different devices into reliability stripes and then adds to each stripe a number of parity stripes. While this technology lowers the overhead of protecting data against failure to a fraction compared with replication, it also introduces two problems.

First, changing data in one block of a stripe requires recalculating the parity. Modern practices avoid this by choosing small block sizes, storing the same file over several blocks, and thus evade the need to rewrite blocks. Second, in case of a failure, a large number of devices needs to be accessed in order to reconstruct the data originally on the failed device, which is then stored elsewhere. Since 2010, codes have been invented that limit the necessary reconstruction traffic for the most common case, where a single block in a stripe needs to be reconstructed. As data centers place devices into racks, and as usually the network bandwidth within a rack far exceeds the bandwidth between racks, we also want to limit data reconstruction limit as much as possible. Recently Pâris proposed to bundle racks using a separate code while still using a rack-internal erasure control code [12]. In this vain, we investigate a variant of the rectangular layout that minimizes both reconstruction traffic and the number of disks accessed in case of a single failed device.

After reviewing shortly relevant work, in the next section, we define the "combinatorial layout", followed by an evaluation of the robustness – the probability that an array with f failures still stores all the original data – and the five year survival probability, which in our case means the probability that data outside the rack does not need to be used in order to reconstruct data lost on failed devices. Our calculations show that difference in robustness do not necessarily lead to large differences in survival probabilities and thus contributes to the understanding of the interplay between survival probability, code layout, and robustness.

II. Related Work

Large data storage systems have reliability requirements beyond the capabilities of RAID Level 6 arrays. For example, the Google File System as well as Windows Azure Storage replicate their active data three-fold [3], [8]. Unfortunately, triple replication consumes also large hardware resources. Erasure control codes have emerged as the preferred solution to protect data against storage failures. An erasure control code adds parity data to a collection of user data blocks. Each parity block is calculated from some or all the data blocks. For example, the popular Reed-Solomon type linear code uses Galois field operations to calculate the contents of a parity block as a linear combination of the contents of the data blocks in the code, Figure 1. A number of data objects, in Figure 1 $D1$, $D2$, $D3$, and $D4$ of equal size are complemented with two parity objects P and Q of the same size. P contains the bitwise exclusive-or of $D1$, $D2$, $D3$, and $D4$, $P = D_1+D_2+D_3+D_4$, whereas the contents of Q are calculated by interpreting the bytes as elements of a Galois Field and calculating the contents byte for byte as a linear combination of the contents of the data objects using certain Galois field elements α_1, α_2, α_3, and α_4 as scalar coefficients, i.e., $Q = \alpha_1 D_1+\alpha_2 D_2+\alpha_3 D_3+\alpha_4 D_4$. If any data object is unavailable, its contents can be calculated from the other data objects and the Q parity, whereas the contents of any two objects can be calculated from all the other ones. As we can see from this discussion, the introduction of erasure correcting coding introduces two problems. First, if we change the contents of any data block, we need to update the contents of all the parity blocks. Second, in case of failure, we need to access the contents of all or all but one objects in such a reliability stripe. In large data centers with thousands of disks, the first problem is solved by forming reliability stripes of relatively small contiguous objects in order to create a single, larger storage object. The contents of such an object are not changed, but later removed. For high reliability, it is only important that all constituent objects are placed on different storage devices. Ceph is a successful example for such an object-oriented storage system [17].

The second problem is a challenge for large data centers. Data centers use racks of highly connected disks or flash drives. Provisioning large bandwidth between racks is more expensive than within racks and so typically, bandwidth between racks is limited. The reconstruction traffic after device failures can interfere with normal data operations. This would induce us to select the components of a reliability stripe among the devices of a single rack. However, racks as a whole can fail. In the case of disks, it is known that vibrations in a failing disk can cause neighboring disks to fail. For this reason, it is desirable to provide for inter-rack reliability, selecting components in different racks for a reliability stripe. To reconcile these diametrically opposed motivations, various types of codes have been proposed or utilized. The idea is to allow the use of data in the same rack for the most frequent failure situation, where a few sectors on a single disk have become inaccessible or a single disk has failed. Some codes in this list are Regenerating Codes [2], Pyramid Codes [7], MDS Array codes [16], and Locally Repairable Codes [11], [14]. The literature has three major metrics, the *repair bandwidth*, the *disk I/O* measured in the number of bits read, and the *repair locality* the number of nodes that participate in a repair as proposed by [4], [10], [11].

While this codes are quite intricate and sophisticated, the basic idea can explained with a simple two-level pyramid code based on the RAID Level 6 and depicted in Figure 2. The Pyramid code calculates the Q parity of a stripe of 16 data objects. The P (aka XOR) parity is however distributed over four parity object, each one contains the exclusive-or of the data in only four objects. The ensemble can tolerate at least two failures. If all the individual P-parity objects are available, we can calculate the exclusive-or to obtain the P parity as defined for a normal Reed Solomon linear code. If there is only one lost data object in the small group of four data objects plus a parity object, then we can recover using only local data. In the example, we can recover the contents of $D_{1,3}$ using the $D_{1,1}$, $D_{1,2}$, $D_{1,4}$ and P_1. Similarly, we can recover the contents of $D_{3,2}$ and $P4$. However, using exclusive-or calculations, it is not possible to recover either $D_{2,2}$ and P_2, because the sub-stripe only has three out of five objects that are available. In this case we are so far lucky as we can recover quickly using only exclusive-or parity. Unfortunately, we did not recover all of the data and need to invoke the underlying Reed-Solomon code. It recovers using

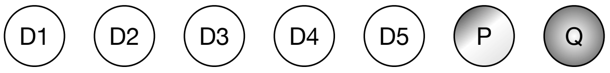

Fig. 1. A small Reed-Solomon linear code as used in a RAID Level 6 stripe.

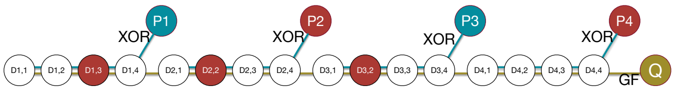

Fig. 2. Reliability stripe with a recoverable pattern of disk failures. Failed disks are shown in red. The P- and Q-parity disks are marked with a letter P or Q, respectively.

the Q-parity and all data objects but for $D_{2,2}$. As we can see, the additional parity objects created for a Pyramid code can be used to recover more failures than the Raid Level 6 stripe is capable of recovering. This becomes at the cost of having to store additional redundant data.

The Pyramid code as it stands is not yet capable of being divided up over several racks. To achieve this, we can store the components of the small reliability stripes in separate racks and then provide enough Q-parity in order to recover from a complete loss of a rack. This means, that we replace the single Q parity with four parities whose contents are calculated using Galois field operations.

We are investigating here in-rack layouts. For this reason, the work of Pâris that “bundles” several in-rack layouts in a flexible manner to create a a very robust total layout, is most relevant to our work [12].

III. Layout Definition

We now present our two in-rack layouts. Both are based on *flat XOR-codes* [6]. Thus, the data objects are organized into *reliability stripes* that consist of a number of data objects and one additional parity object. The contents of the parity object are the bitwise exclusive-or (XOR) of the data objects. Our layouts provide two failure tolerance within a rack to deal with latent sector failures or up to two disk failures without cross-rack traffic and in the case of one failure, uses one short reliability stripe in order to minimize reconstruction traffic and disturbance. We assume that an additional layer of protecting is provided by bundling one data object per rack into additional reliability stripe, as recently proposed by Pâris [12].

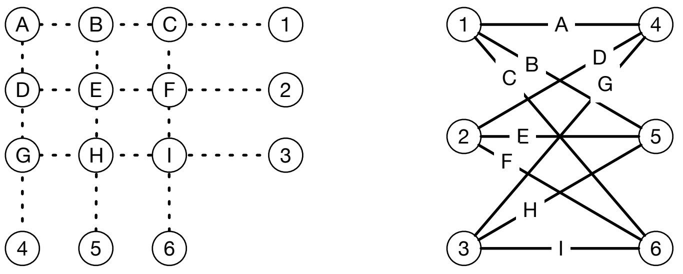

Fig. 3. Left: The two-dimensional layout for a storage array. A, B, . . ., I are data devices and 1, 2, . . ., 6 parity objects. Right: The corresponding graph visualization.

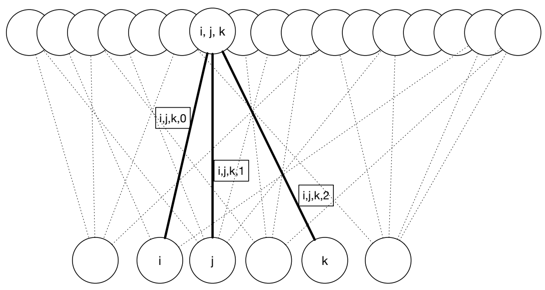

Fig. 4. Graph presentation of a combinatorial layout.

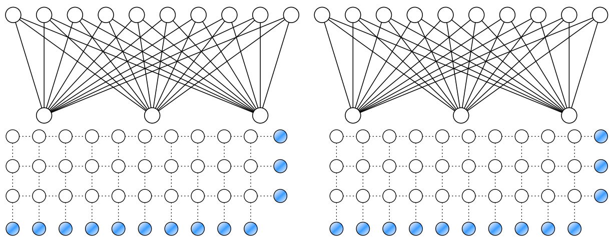

Fig. 5. The double rectangle layout used in the experiments. Top: the graph representation. Bottom: the layout itself.

We use a graph presentation developed by Xu and collegues for the definition of B-codes [18]. For our layouts, each data object needs to be in two different reliability stripes and two reliability stripes are either disjoint or intersect in exactly one data object. This implies that each data object is uniquely characterized by the two stripes to which it belongs. Mathematically, the data objects are the edges and the stripes are the vertices of a Mathematical graph. We present an example in Figure 3. There we have nine data objects (A, ..., I) and six parity objects (1, ..., 6). The left shows the data layout, where each data object is located in a vertical and a horizontal reliability stripe. The right side shows the graph presentation. For example, data object E is located in the same stripes as parity objects 2 (horizontally) and 5 (vertically). Correspondingly, the edge E is connecting the vertices 2 and 5. We adopt the convention to identifying the reliability stripe with the parity object.

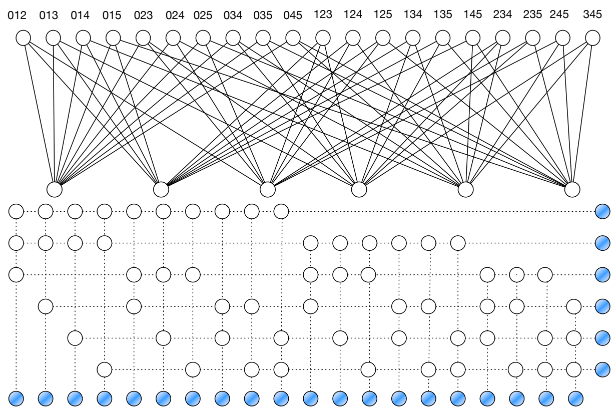

Fig. 6. Graph presentation (top) of and the combinatorial layout itself used in our experiments.

The graph in Figure 3 is bipartite, that is, its vertices are partitioned into two sets and edges are located only between vertices from the two partitions. In our setting, we have two types of vertices (corresponding to reliability stripes), those will a small edge rank (the number of edges adjacent to a given vertex) and those with a larger edge rank. Since each disk object forms part of a small and a large reliability stripe, the edges of the graph are between the small and the larger rank vertices. Thus the graph has to be bipartite.

It makes sense to only choose graphs with only two edge ranks k_1 and k_2. The smallest layout with these two ranks is the rectangular layout such as the one presented in Figure 3. A small layout is however vulnerable to a moderate number of objects on failed storage devices, whereas a larger one dilutes the impact of a moderate number of failures, but has of course also a higher chance of suffering from such a moderate number of failures. We propose here an alternative to the strict rectangular layout which we call the *combinatorial* layout. Its graph definition is presented in Figure 4. For the vertices with high edge rank, we choose s disks labeled from 0 to $s-1$. The vertices with low edge rank are then labeled by the Mathematical combinations of r, $(r < s)$ elements. Thus, there are $\binom{s}{r}$ of them. We then join a low-rank vertex with label $\{a_1, a_2, \ldots, a_r\}$ to each of high-rank vertices labeled a_1, a_2, ..., and a_r. The low rank is therefore equal to r. The high rank is equal to $\binom{s-1}{r-1}$. The number of data objects is $r\binom{s-1}{r-1}$. We label them as $\{a_1, a_2, \ldots, a_r, i\}$ connecting $\{a_1, a_2, \ldots, a_r\}$ with a_i.

For our experiments, we choose a layout with $s = 6$ and $r = 3$. This layout has exactly the same number of parity objects and data objects as a double rectangular layout with ranks r and $\binom{s-1}{r-1}$.

The combinatorial layout in a sense spreads the parity information of the same set of data objects more evenly over the same set of parity objects, as a comparison of the bottom of Figures 5 and 6 shows. Consequentially, we expect its robustness and also its resilience to be larger.

Recall that robustness is the probability of loosing data when f layout components have failed. It is such a function and it is calculated based on the assumption that each disk is equally likely to be one of the failed disks. Traditionally, resilience has been measured in Mean Time to Failure (MTTF), a notion that has suffered its share of criticism and discussion [5], [9]. We calculate resilience as the probability of data loss in a five year interval using simulation under simplifying assumptions.

IV. Experimental Evaluation

We used two measures to evaluate the resilience of layouts. First, we calculate robustness. For this purpose, we created a Python program for the combinatorial layout and the rectangular layout represented by the 10×3 rectangular layout, i.e., half of our double rectangular layout. The program randomly selected f objects for failure and then modeled the attempt to reconstruct the data of the failed objects. We called

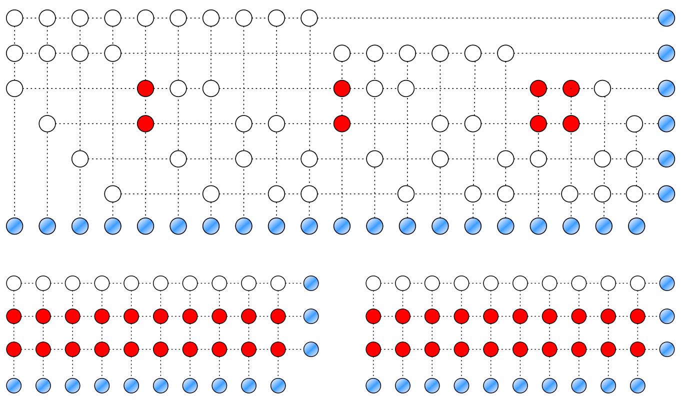
Fig. 7. The result of dispersion. In the combinatorial layout (top), there are only four vertical reliability stripes that have disks in positions 2 and 3, whereas in the double rectangular layout (bottom), all vertical stripes contain disks in positions 1 and 2. Since each pair in a vertical stripe combined with another pair in a vertical stripe forms a four-cycle, there are many more four cycles in the double rectangular layout.

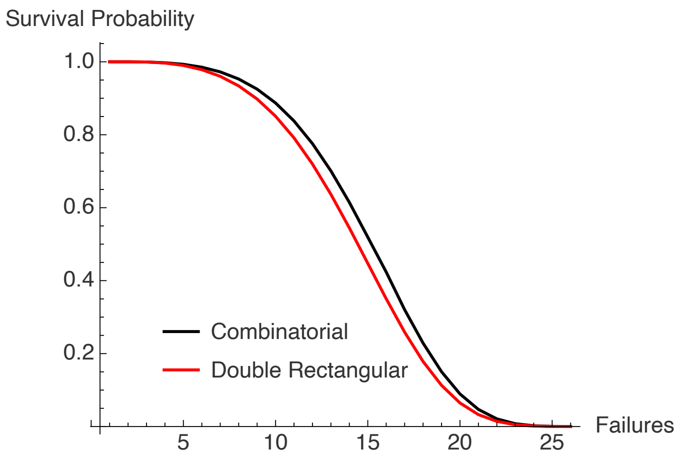

Fig. 8. Robustness of the combinatorial and the double rectangular layout.

the probabilities p_f that f failures do not lead to dataloss the *robustness* of the layout. From the data for the single rectangular layout we then calculated the robustness of the double rectangular layout. As Figure 8 shows, the robustness of the combinatorial layout is always higher than that of the double rectangular layout, confirming the old addage 'to not put all one's eggs into the same basket' in order to spread the risk. Of course, p_f is one for $f = 0, 1, 2$, since the layout is two failure tolerant, and p_f is zero if f exceeds the number of parity objects, i.e. 26 in our case.

We can identify sets of disks whose failure no longer allows all data to be reconstructed, the so-called *failure patterns*. A failure pattern in a flat layout contains either a cycle of failed edges or a path of failed edges between two failed vertices [13]. This means that the smallest failure pattern in both layouts consists of a failed edge between two failed vertices, or with other words, a failed data object where the parity object in both the data object's reliability stripes have failed as well. Since the number of data objects is 60, and there are $\binom{86}{3} = 102340$ patterns of three failed objects, the probability that three randomly chosen, failed objects can be survived is 0.999414.

We now count the number of four-failure patterns that lead to dataloss. Our first category are the 60 three-failure patterns plus an additional failed object. There are $60 * 83 = 4980$ of them in both layouts.

The next pattern is the four-cycle, consisting of four failed adjacent edges. This is a quadrangle of data objects $d_1, d_2,$, d_3, and d_4 such that the pairs (d_1, d_2), (d_2, d_3), (d_3, d_4), and (d_4, d_1) all share a reliability stripe. By necessity, two of the stripes are large and two of them are short. In the double rectangular pattern, Figure 5, we can select any pair of rows and any pair of adjacent columns for such a quadrangle. This means that there are $2\binom{10}{2}\binom{3}{2} = 270$ of them. In the combinatorial layout, a quadrangle would contain two data objects in the same vertical reliability stripe. The other two data objects would also be in a different vertical reliability stripe. The first object in the first pair and the first object in the second pair share the same horizontal reliability stripe and the same is true for the second objects. To count them all, we first select two horizontal stripes. Two of the data objects in a vertical reliability stripe are then determined, there are $6 - 2 = 4$ possibilities for the third one and by construction, all of them correspond to a vertical reliability stripe. Thus, we choose two out of six horizontal and then two out of four candidate vertical stripes. This gives us a total of $\binom{6}{2}\binom{4}{2}$ $= 90$ possibilities. Figure 7 graphically explains the difference between the two layouts and also suggests a different, but equivalent way of counting them.

The final pattern is the two-path, consisting of two failed vertices connected by two failed edges. In the double rectangle layout, any pair of the ten upper vertices and any pair of the three lower vertices is connected through each of the other vertices. This gives us $2 * (\binom{3}{2}10 + \binom{10}{2}3) = 330$ patterns. For the combinatorial layout, we obtain the same number. First, we select a random horizontal parity object, then one of the 10 data objects in its stripe, then one of the two other data objects in the vertical stripe, which gives us the vertical parity object that needs to be selected for a total of $6 * 10 * 2$ paths, each counted exactly twice, so that the number of paths is 60. Second, we start with a random vertical parity object among the 20, then select one of the three data objects in the vertical stripe, then one of the remaining nine data objects in the horizontal stripe, which leads us to the other vertical parity object. As we counted each path twice, the total number is $20 \cdot 3 \cdot 9/2$ or 270 for the same total of 330.

Together, we have 5580 combinations of four objects that lead to dataloss for the double rectangular as opposed to 5400 for the combinatorial layout. Our counting examplifies the reason why the robustness of the combinatorial layout is always at least as good as that of the double rectangular layout.

To recap, we can calculate the robustness p_f exactly for $f \leq 4$ and approximately, but with confidence intervals too small to depict in Figure 8, through simulation.

However, what are the ramifications of the differences in

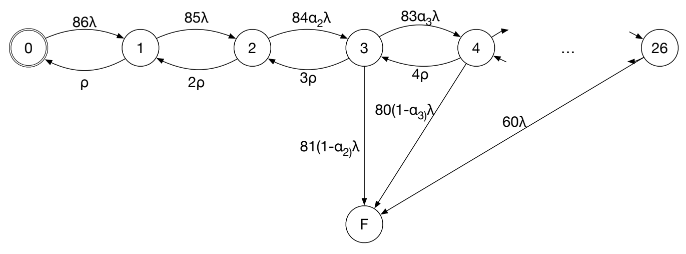

Fig. 9. Markov model to assess data loss probability.

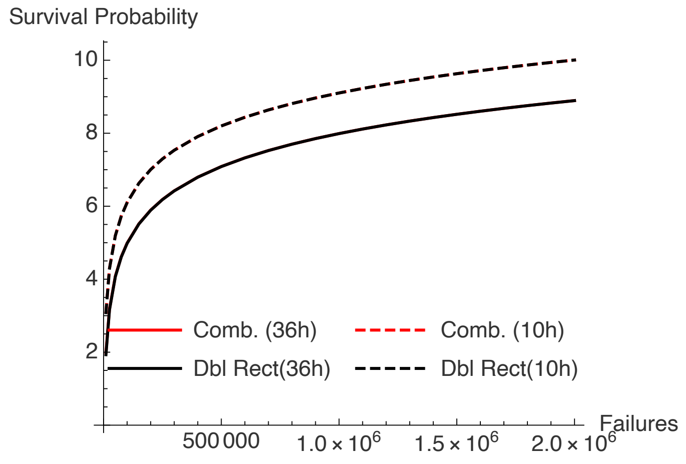

Fig. 10. Number of nines in the five-year data survival of the combinatorial and the double rectangular layout. The black and the red graphs differ by so little, that the red ones are obfuscated by the black graphs. Since we assume the existence of a bundling inter-rack code, data loss only means that we need to use this outer code to reconstruct the data.

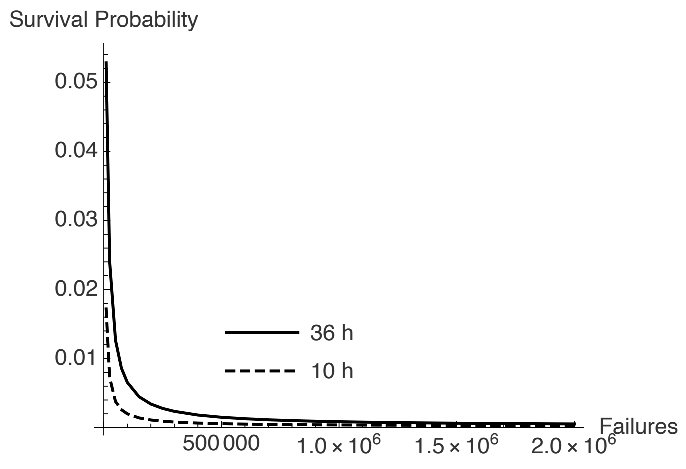

Fig. 11. Difference in the number of nines in the five-year survival of the combinatorial and the double rectangular layout.

robustness? To do so, we use the standard Markov model depicted in Figure 9. There, we have 26 non-failure states $0, \ldots, 26$ and an absorbing failure state F. We denote the failure rate of an individual object by λ, so that $1/\lambda$ is the Mean Time Between Failures (MTBF) of the objects, which is the MTBF of the individual disks. We have transitions out of state i at a rate $(86-i)\lambda$ corresponding to a failure of any of the $(86-i)$ alive objects. Depending whether the additional object failure leads to data loss or not, the transition moves to the next state $i+1$ or to the failure state. From each state $i, i > 0$, we have a repair transition to state $i-1$. We model repair transitions as having an exponentially distributed repair time ρ and assume that repairs are independent from each other. We obtain the probabilities that the i^{th} failure does not lead to the failure state from the robustness.

With these model, we can calculate the probability that our two layouts by themselves cannot reconstruct any data on lost devices. We use the Euler approximation of the state probabilities. In order to reduce the accumulation of numerical errors, we use the high floating-point-precision library Apflot [15]. We use two conservative reconstruction time of 36 hours and 10 hours, a time interval of a second and a precision of 50 decimal digits. We varied the MTBF of the devices between 10000 hours and 2,000,000 hours. Industry experience shows that sometimes device MTBF are more oriented towards the lower range, while for example, many disk manufacturers' data sheets promise 2,000,000 MTBF. To represent our probabilities that the ensemble preserves data for five years over this large range of MTBF, we give the probability in number of nines, that is, if the probability that all data survives without using inter-rack reliability is p, then we display $-\log_{10}(p)$. For instance, if the probability is 0.999,, then we have three nines of reliability.

Our results are depicted in Figure 10, where to our initial surprise, the curves are almost on top of each other. Figure 11 shows the difference between the curves for repair times of 10 hours and 36 hours, respectively. It is only substantial for very low MTBF values and the difference gets smaller with faster repair times. We can attribute this to the fact that the system is the vast majority of its life in State 0, occasionally ventures into State 1, and finds itself in the other states with exponentially smaller probability. Consequentially, the robustness mostly matters for small number of failures. As our two layouts have the same robustness for States 0, 1, 2, and 3 and only differ by a small fraction for States 4, this behaviour in retrospect should not come as a surprise. Furthermore, if the repair rates is smaller, than the differences should be even less pronounced. This calculation in fact contributes to our understanding of the interplay between erasure control codes and longevity of data. For instance it justifies the frequent approximation that assumes that all failures of more than N objects lead to dataloss.

V. Conclusions

We compared the impact of two erasure controlling layouts for inner-rack protection of stored data in a data center. Both layouts limit the amount of traffic read in order to reconstruct data on a failed device and should be used in conjunction with another code that gathers blocks from devices in different racks into reliability stripes protected with additional parity data. The first layout is a classic rectangular layout considered many times in the literature. The second takes the data and parity objects from two rectangular layouts and mixes them up, resulting in significant improvement in robustness. However, calculation of 5-year survival probability (where survival is understood to avoiding recourse to the inter-rack protection) show very little difference in the probabilities. This is to our knowledge the first example that demonstrates that robustness differences for larger number of failures can have very little effect on survival probability.

References

[1] A. Cave, "What will we do when the world's data hits 163 zettabytes in 2025?" https://www.forbes.com/sites/andrewcave/2017/04/13/what-will-we-do-when-the-worlds-data-hits-163-zettabytes-in-2025, 2017.

[2] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE transactions on information theory*, vol. 56, no. 9, pp. 4539–4551, 2010.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings, 19th ACM Symposium on Operating System Principles (SOSP)*, 2003.

[4] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin, "On the locality of codeword symbols," *IEEE Transactions on Information theory*, vol. 58, no. 11, pp. 6925–6934, 2012.

[5] K. Greenan, J. Plank, and J. Wylie, "Mean time to meaningless: MTTDL, Markov models, and storage system reliability." in *HotStorage*, 2010, pp. 1–5.

[6] K. M. Greenan, E. L. Miller, and J. J. Wylie, "Reliability of flat XOR-based erasure codes on heterogeneous devices," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 2008, pp. 147–156.

[7] C. Huang, M. Chen, and J. Li, "Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems," *ACM Transactions on Storage (TOS)*, vol. 9, no. 1, p. 3, 2013.

[8] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in Windows Azure storage," in *Proceedings, USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 15–26.

[9] I. Iliadis and V. Venkatesan, "Rebuttal to "beyond mttdl: A closed-form raid-6 reliability equation"," *ACM Trans. Storage*, vol. 11, no. 2, Mar. 2015.

[10] F. Oggier and A. Datta, "Self-repairing homomorphic codes for distributed storage systems," in *2011 Proceedings IEEE INFOCOM*. IEEE, 2011, pp. 1215–1223.

[11] D. S. Papailiopoulos and A. G. Dimakis, "Locally repairable codes," *IEEE Transactions on Information Theory*, vol. 60, no. 10, pp. 5843–5855, 2014.

[12] J.-F. Pâris, "Bundling together RAID disk arrays for greater protection and easier repairs," in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2019, pp. 256–261.

[13] T. Schwarz, A. Amer, and J. Rose, "Resar: Reliable storage at exabyte scale reconsidered," in *2017 2nd International Conference on Communication Systems, Computing and IT Applications (CSCITA)*. IEEE, 2017, pp. 84–89.

[14] I. Tamo, D. S. Papailiopoulos, and A. G. Dimakis, "Optimal locally repairable codes and connections to matroid theory," *IEEE Transactions on Information Theory*, vol. 62, no. 12, pp. 6661–6671, 2016.

[15] M. Tommila, "Apfloat arbitrary precision library." [Online]. Available: www.apfloat.org

[16] Z. Wang, I. Tamo, and J. Bruck, "Long MDS codes for optimal repair bandwidth," in *2012 IEEE International Symposium on Information Theory Proceedings*. IEEE, 2012, pp. 1182–1186.

[17] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.

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