Lazy Means Smart: Reducing Repair Bandwidth Costs in Erasure-coded Distributed Storage

Mark Silberstein[1], Lakshmi Ganesh[2], Yang Wang[3], Lorenzo Alvisi[3], Mike Dahlin[3,4]
[1]Technion, [2]Facebook, [3]The University of Texas at Austin, [4] Gooogle

ABSTRACT

Erasure coding schemes provide higher durability at lower storage cost, and thus constitute an attractive alternative to replication in distributed storage systems, in particular for storing rarely accessed "cold" data. These schemes, however, require an order of magnitude higher recovery bandwidth for maintaining a constant level of durability in the face of node failures. In this paper, we propose *lazy recovery*, a technique to reduce recovery bandwidth demands down to the level of replicated storage. The key insight is that a careful adjustment of recovery rate substantially reduces recovery bandwidth, while keeping the impact on read performance and data durability low. We demonstrate the benefits of lazy recovery via extensive simulation using realistic distributed storage configuration and published component failure parameters. For example, when applied to the commonly used RS(14,10) code, lazy recovery reduces repair bandwidth by up to 76% even below replication, while increasing the amount of degraded stripes by 0.1 percentage points. Lazy recovery works well with a variety of erasure coding schemes, including the recently introduced bandwidth efficient codes, achieving up to a factor of 2 additional bandwidth savings.

1. INTRODUCTION

Erasure coding schemes, e.g. Reed-Solomon (RS) codes, are an attractive alternative to replication in distributed storage systems (DSS) as they enable an optimal balance between storage cost and data durability. They are considered a particularly good fit for storing rarely accessed "cold" data [3, 15], which constitutes an increasingly large fraction of the data in large-scale storage systems [21, 24].

However, broader adoption of erasure codes in cold-storage DSS is hindered by their excessive network demands when recovering data after node failures – the well-known *repair bandwidth problem* [25, 18, 27, 26]. Specifically, when a storage node fails, due to a hardware failure, for example, its contents must be promptly restored on another node in order to avoid data loss. Recovery of a single data block in an RS(n,k) erasure coded storage entails transferring k blocks from k surviving nodes over the network. In comparison, only one block is transferred to recover a single block in replication-based systems. This k-fold increase in recovery traffic results in sharp growth of the background steady-state network load, in particular in large-scale data centers where failure rates are high. For example, in a real Facebook DSS, up to 3% of all the storage nodes fail each day [27]. Using RS(14,10)-encoded storage in this system results in hundreds of terabytes of daily recovery traffic through Top-Of-Rack (TOR) switches [26]. Thus, the network is busy even when the DSS itself is idle and does not serve any external users (as is usually the case for a cold-storage system), which increases its power consumption and prevents the nodes to enter sleep mode. Further, this traffic constitutes a significant portion of the total network volume, and grows with system scale and hard disc capacity. All these factors imped the adoption of standard erasure codes.

A common practice to cope with the growing network traffic is by throttling the network bandwidth available for recovery tasks. Doing so, however, increases the number of degraded stripes in an uncontrolled manner (stripes with one or more blocks lost or offline), which in turn dramatically affects read performance and data durability. A popular policy to smoothen this negative effect is to prioritize the recovery of the stripes with higher number of failed nodes [9, 26], thereby better utilizing the available limited bandwidth. However in general, network throttling does not effectively reduce the overall failure recovery traffic because the vast majority of failures gets recovered promptly – it was found that on average 98% of degraded stripes in the Facebook DSS had only one failed block [26].

In this paper we propose *lazy recovery*, a technique to reduce the volume of network recovery traffic in an erasure-coded DSS to the level of 3-way replication but without loosing the durability advantages of the former. The idea is simple: the recovery of degraded stripes can be delayed as long as the risk of data loss remains tolerable, thereby leveraging a non-linear tradeoff between recovery bandwidth and the probability of data loss. The key challenge, however, is to find a practical mechanism to put the system into the desired operating point on the tradeoff curve. For example, delaying the recovery until the number of alive nodes per stripe falls below a certain recovery threshold as was first suggested in TotalRecall peer-to-peer storage system [2], is not effective in a large-scale DSS scenario. We suggest new lazy recovery scheme which enables a fine-grain control of the system behavior and achieves significant bandwidth reduction at the expense of relatively small decrease in data durability and availability.

Realistic evaluation of our scheme requires estimating its influence on durability, availability and recovery bandwidth in a petabyte-scale DSS over several years of operation. Performing such evaluation on a real system is unrealistic. Therefore, we implemented a detailed distributed storage simulator — *ds-sim* [1], which simulates long-term steady-state DSS behavior. The simulator takes as an input a system configuration and a data encoding and recovery scheme, and runs by simulating failures in major failure domains (disk, machine, rack), latent block failures, machine replacement, as well as enforces a global network limit.

---

[1]*ds-sim* is available for download at https://code.google.com/p/ds-sim

We simulate a 3PB system using failure and recovery distributions from real traces of the failures of large-scale compute clusters [30] as well as publications characterizing production environments [7, 9, 30, 31, 23]. We evaluate several popular erasure coding schemes and show that our lazy recovery mechanisms are able to significantly reduce the repair bandwidth. For example, the repair bandwidth of RS(14,10) codes is reduced by a factor of 4, even below the level of 3-way replication. Furthermore, lazy recovery works well in conjunction with recently introduced locally-repairable (LRC) codes [25, 18], further reducing their repair bandwidth by half.

The main contributions of this paper are:

- A new mechanism for enabling significant repair bandwidth reduction in erasure coded storage and designed for a large-scale DSS

- A detailed DSS simulator *ds-sim* for evaluating long-term durability, availability and recovery bandwidth requirements of various storage schemes. *ds-sim* simulates realistic failure scenarios such as correlated failures, latent disc block failures and hardware replacement.

- Evaluation of the bandwidth requirements of different storage schemes using long-term traces and failure distributions from production large-scale storage systems.

The rest of this paper is organized as follows. In the next section, we provide an overview of erasure coding and the associated repair bandwidth problem. We describe the lazy recovery scheme in detail in section 3, present our distributed storage simulator in section 4, and evaluate lazy recovery in section 5. Section 6 describes related work, and section 7 concludes.

2. ERASURE CODES AND THE REPAIR BANDWIDTH PROBLEM

In this section we briefly explain the use of erasure codes in a DSS on the example of Reed-Solomon codes, and then explain the repair bandwidth problem. In an RS(n,k) storage system, each set of k data blocks of size b is encoded into n blocks of size b, forming a single data *stripe*. Each stripe comprises k *systematic* blocks, which store the original data content, and *n-k* additional parity blocks. 3-way replication can be thought of as a trivial form of RS coding with $k = 1$ and $n = 3$. The blocks in each stripe are distributed across n different storage nodes to provide maximum failure resilience. In a failure-free case, the original data can be retrieved *without* additional decoding by reading the contents of the respective systematic block. If, however, one or more of the systematic blocks is unavailable, the stripe is termed *degraded*. The contents of missing blocks can be reconstructed from *any* k stripe blocks from the surviving nodes.

As in replicated storage systems, the contents of failed nodes must be recovered to avoid eventual data loss. Recovery poses significant network bandwidth demands, and may induce significant load even when there are no external I/O requests. Reconstructing a single data block requires the entire stripe's worth of data (k blocks) to be read. This translates to a bandwidth inflation of k; i.e., repairing b bytes requires $b * k$ bytes to be transfered. Compare this with
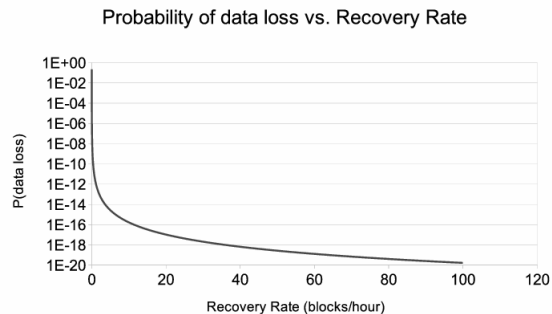

Figure 1: Theoretical tradeoff between durability and recovery rate.

replication: repairing a lost block (replica) requires only one other block (replica) to be read. *Our goal is to make the steady-state bandwidth requirements of erasure coded storage commensurate with that of replicated storage, while preserving the failure resilience advantages of erasure coding.*

In the rest of this paper, we define a stripe as *durable* if enough of its blocks survive (even if not online) such that the stripe data can be reconstructed. On the other hand, a stripe is considered *available* if all its systematic blocks are online, so that no reconstruction is required to read the stripe.

In the next section, we show how lazy recovery limits recovery bandwidth while maintaining the high durability and availability of erasure-coded storage.
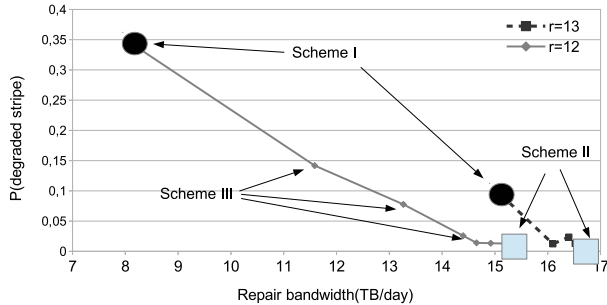
3. LAZY RECOVERY

The basic idea behind the lazy recovery scheme is to decrease the recovery rate, thereby reducing the required network bandwidth, but without significant impact on durability. Figure 1 provides intuition behind this approach. The graph shows the probability of a single data block loss over 10 years — a quantitative measure of the durability of data stored in a DSS – as a function of data recovery rate for the RS(14,10) encoding scheme. We use Markov chain model as in [12] to produce the graph. While it is well known that Markov models inflate the values of Mean-Time-To-Data Loss (MTTDL), they are still useful for qualitative analysis of system behavior [12]. This simple experiment highlights the diminishing returns of increasing recovery rate as the system becomes more durable: a single block loss probability of 10^{-19} over 10 years is, indeed, 10 times higher than 10^{-20}, but it might save half of the recovery network traffic while still being sufficient for practical purposes.

The main challenge, however, is *to design a practical mechanism to exploit this tradeoff in a real system.*

Scheme I: Lazy recovery for all.

One possible solution inspired by the work done in the context of P2P storage networks [22, 2] is to postpone the reconstruction of failed blocks until the number of available blocks in a stripe reaches a given recovery threshold r. For example, for RS(15,10) and $r = 13$, the system will wait for two blocks in a stripe to fail before triggering the stripe recovery. Intuitively, the probability of permanent data loss with lazy recovery using RS(15,10) encoding should be roughly equivalent to that of the original RS(14,10) with eager recovery,

Figure 2: Impact of lazy recovery on the average amount of degraded chunks for RS(15,12). The highlighted points denote the cases where recovery is triggered only after r failures (lazy naive) and when the permanent failures are always prioritized (lazy prioritized).

since in both schemes recovery is triggered when 13 blocks are still alive.

Delaying recovery yields two advantages: first, we can recover two blocks for almost the same network cost as recovering one—to recover one block, one must read ten and write one (a total of 11x bandwidth per recovery), while to recover two, one still reads ten, but writes two (total $12\times$ bandwidth, or $6\times$ amortized bandwidth per recovery); Second, if a block is unavailable due to a transient event, such as a network outage, delaying its recovery allows it more time to come back on its own (e.g. when network connectivity is restored), thus avoiding a redundant repair.

This basic lazy recovery scheme might seem similar to the standard practice of delaying recovery of failed nodes by a fixed amount of time, (usually 15 minutes [9]) to avoid unnecessary repairs of short transient failures. The main difference, however, is that the lazy scheme does not transfer any data until the recovery is required regardless of how much time passed after the failure, and then restores multiple blocks in a batch, thereby transferring strictly less data than the standard delayed recovery scheme.

Lazy recovery was originally proposed in the context of TotalRecall peer-to-peer storage system, but it is not efficient enough for a DSS. Our simulation shows (Figure 2) that decreasing the recovery threshold by one may dramatically increase the number of degraded stripes. For example, for the RS(15,10) scheme, the recovery threshold $r=12$ results in about 30% of all stored stripes to be always degraded. On the other hand, increasing the repair threshold to $r = 13$ helps to lower the number of degraded stripes, but moves the system straight to the other extreme of the tradeoff function (Figure 2), losing all the bandwidth savings.

Scheme II: Lazy recovery only for transient failures.

The inefficacy of Scheme I stems from the fact that stripes that become degraded through permanent failure events stay degraded without being repaired for too long (since a crashed hard disc will never recover on its own, whereas transient failures eventually recover even without explicit recovery procedure). Hence, we need to refine our scheme to distinguish between permanent disk failures and transient ma-
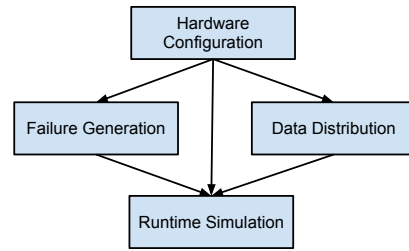
Figure 3: Distributed Storage Simulator

chine failures: permanent failures trigger repair process immediately as they are detected, while transient failures are handled lazily. In a controlled environment such as a data center, enough information exists to distinguish between permanent events such as hardware upgrades, and transient events such as machine crash, restart, and software upgrade.

Separating recovery policies for permanent and transient failures improves the efficacy of the original lazy approach. However, as we see in Figure 2, this scheme is still not capable of providing fine-grained control over the choice of the operating point of the tradeoff function.

Scheme III: Dynamic recovery threshold.

Scheme II can be improved by dynamically adjusting the recovery policy depending on the state of a whole system. We introduce a system-wide limit on the number of degraded stripes with permanently lost blocks. Whenever a permanent failure event causes the limit to be exceeded, we temporarily raise the system-wide recovery threshold until the number of such stripes is reduced. Note that enforcing system-wide limits and applying global policy changes such as the one used in this scheme can be efficiently carried out in a centrally-managed well-maintained DSS but might be unrealistic in a much less controlled peer-to-peer storage environment.

In what follows, we first describe our evaluation methodology, and then present results to demonstrate the efficacy of this enhanced lazy recovery scheme.

4. EVALUATION METHODOLOGY

Evaluating the efficacy of lazy recovery in reducing repair bandwidth and its implications on availability and durability poses a challenge. It is not practical to run a prototype and measure those metrics, since it requires a large number of machines to run for years to get a statistically meaningful result.

As is common in other studies of storage systems [20, 9, 25], we use a combination of simulation and modeling: on the one hand, we build a distributed storage simulator *ds-sim* to estimate how repair bandwidth and availability are affected by a combination of failure events, hardware configuration, coding scheme and recovery strategy. On the other hand, to capture unrecoverable data loss events, which are extremely rare especially for erasure codes, we use a Markov chain model to compare the durability of different coding schemes.

4.1 Simulation For Bandwidth And Availability Estimation

ds-sim simulates the system behavior over several years. For example, in our simulations we used one decade. The

inputs include hardware configuration specifications, such as disk sizes and global network capacity, statistical properties of the failure and recovery distributions of the storage system components, and the data encoding scheme. The simulator returns steady-state and instantaneous values of network bandwidth utilization, number of degraded stripes, number of permanently lost blocks, as well as various other dynamic system properties.

ds-sim consists of four main building blocks (Figure 3):

Storage system configuration.

ds-sim simulates a commonly used 3-tier tree structure of the storage components, including *racks*, *machines*, and *disks*. Each higher-level component can have multiple lower-level components as its children. If a parent component fails, all its children are marked unavailable, effectively simulating failures in a single failure domain.

Storage scheme simulation.

As in real systems, the data is stored in *blocks* [10, 28, 3]. Multiple blocks form a *stripe*. Data within each stripe is either replicated or erasure-coded: for n-way replication, a stripe comprises all replicas of a block; for for $RS(n, k)$ encoding scheme, a stripe comprises k original blocks and $n-k$ parity blocks. *ds-sim* randomly chooses n racks to store n blocks of a stripe in different failure domains, following the standard practices in production settings [9].

Failure events generation.

ds-sim generates failure and recovery events for each hardware component using either synthetic probability distributions or failure traces. Some events, e.g. machine failures, are generated offline before the simulation, but others, such as lazy recovery events, depend on the dynamic system behavior and are created during the runtime simulation.

We incorporate separate failure and recovery distributions for each storage component: disk, machine, and rack.

- *Disk Failures:* These include both latent failures and permanent disk failures. Latent failures damage random disk sector affecting a single data block. They are detected and recovered during periodic reads of the entire disk content, a technique called scrubbing [7, 9]. Permanent disk failures are assumed to be unrecoverable, permanently damaging all blocks on the disk.

- *Machine Failures:* These include transient failures (which do not damage the component disks), and permanent failures (which also take down the component disks). Transient failures are commonly attributed to network slowdown or maintenance. Permanent failures represent server hardware upgrades, which reportedly occur once in three years [8]. Consequently, recovery from permanent machine failures is assumed to start immediately after failure. However, recovery from temporary machine failures begins after a specified timeout, e.g. 15 minutes [9], if the lazy policy dictates it.

- *Rack Failures:* Rack failures are assumed to be transient (that is, they do not damage the component servers).

While we do not explicitly simulate correlated failures for each component in isolation, the dominant failure domains are simulated via higher-level components inducing failures on all the components they contain.
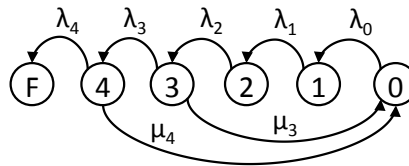
Figure 4: Markov model for evaluating durability of lazy recovery, permitting three unrecovered failures.

Note that *ds-sim* allows users to specify different models or parameters for different groups of components. For example, one groups of disks can be configured as 2-year old and another group can be configured as new, so they can have different failure rates.

Runtime simulation.

Finally, by combining all the above information, *ds-sim* performs a runtime simulation, recording all instantaneous properties of the system including repair bandwidth and the number of degraded stripes. Note that *ds-sim* is *not* designed to estimate the read or write performance of the system. The simulator tracks every block in the system. A stripe to which the block belongs is marked as available, unavailable, degraded, or lost, depending on the state of its respective blocks and the simulated storage scheme. For n-way replication schemes, the degraded state is irrelevant because a stripe is either available when at least one of its block is online, or unavailable otherwise. For an $RS(n, k)$ erasure-coding scheme, a stripe is marked as degraded if there are less than n blocks, and as unavailable or lost if there are less than k blocks.

4.2 Markov Model For Durability Estimation

ds-sim can be used to estimate the durability of the system. However, it has been observed [12] that for erasure-coding schemes, durability loss events happen so rarely that the simulation requires a very large number of iterations to get a statistically meaningful result. Therefore, we resort instead to a Markov model to obtain durability results [13]. Compared to simulation, Markov models are known to inflate the absolute values of durability, but they are useful for comparison purposes [14].

We use standard parallel repair model for replication and erasure codes [12]. Lazy recovery is modeled by removing the recovery transition from states that have more available chunks than the recovery threshold. Figure 4 presents the model for the lazy recovery scheme with four parity nodes and the recovery delayed until three nodes fail. The state labels represent the number of failed nodes. Note that this approach ignores the dynamic recovery threshold increase in our actual scheme.

Armed with *ds-sim* and the Markov model, we now proceed to evaluate the efficacy of our lazy recovery scheme.

5. RESULTS

We simulate a DSS storing 3PB of data, running for one decade. Figure 6 describes each type of failure we simulate, and lists our failure model parameter choices and their sources. $W(\gamma, \lambda, \beta)$ refers to a Weibull distribution, while $Exp(\lambda)$ refers to an exponential distribution. Figure 5 lists

Parameter	Value
Total data	3 PB
Disk cap	750 GB
Disks/machine	20
Machines/rack	11
Bandwidth cap for repair	650 TB/ day [5]
Duration	10 yrs
Num iterations	25,000

Figure 5: Simulation Parameters

the storage system parameters we used. We ask the following questions: *How effective is our lazy recovery scheme in controlling repair bandwidth?* and *what is its impact on data availability and durability?*

Figures 7 and 8 answers both questions. It compares several representative storage schemes along the dimensions of interest. The label for the lazy schemes is constructed as *n-k-r*, where (n, k) identifies the encoding scheme, and r is the recovery threshold. The candidate schemes we evaluate are: 3-way replication, the original RS(14,10), RS(14,10) with lazy recovery (*14-10-12*), and RS(15,10) with lazy recovery (*15-10-12*).

We also compare lazy recovery with two recently introduced repair-efficient erasure coding schemes: *Xorbas(16,10,12)* [27] and *Azure(16,12,14)* [18]. Finally, we combine these repair efficient codes with lazy recovery (*Xorbas+LAZY*), (*Azure+LAZY*), setting the repair threshold to 12 and 14 respectively.

All the results are normalized against 3-way replication. The data label at the top of each bar group shows the actual value of the replication data point.
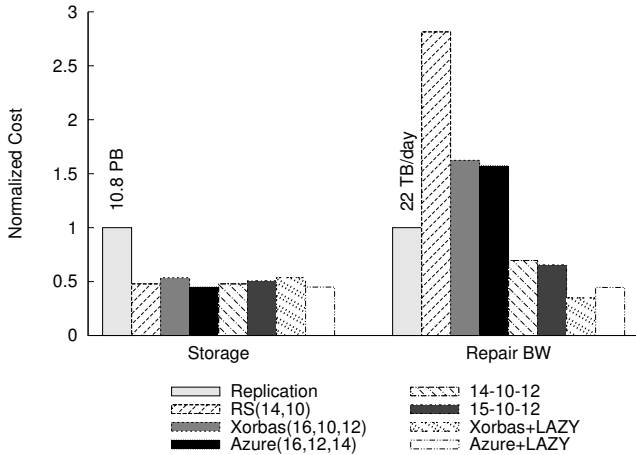

Figure 7: The storage requirements and repair bandwidth of lazy recovery versus non-lazy schemes

We see from Figure 7 that the lazy recovery schemes yield a 4x reduction in repair bandwidth compared to basic erasure coding, with 70% of this savings coming from masking of transient failures, while the remaining 30% is from block recovery amortization (this breakup is not shown in the graph). We outperform repair-efficient Azure and Xorbas coding schemes in this respect by more than a factor of
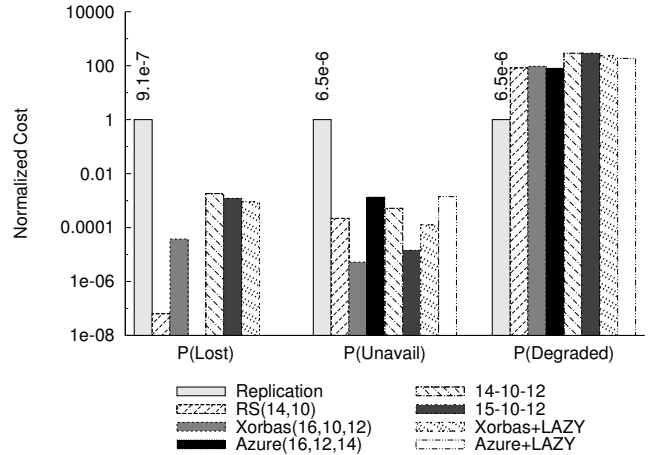

Figure 8: The probability of data loss, unavailable data and the portion of degraded stripes of lazy recovery versus non-lazy schemes

2. In fact, we even outperform full replication, which, being less reliable (and non-lazy), experiences a larger number of recovery events. We note that the average repair bandwidth computed by the simulator is lower by about a factor of 4 than that reported by Facebook [26], in part because the size of the Facebook DSS is by 50% larger than that of size of the simulated DSS. However the exact DSS size is not specified in the paper, and we deduce it using other publications about Facebook DDS [27].

The impact of lazy recovery on durability and availability is shown in Figure 8. First, we compare the fraction of degraded data in each of the candidate schemes. We see that lazy recovery increases this fraction by over 2 orders of magnitude compared to 3-way replication. When compared to RS(14,10), however, we only increase this fraction by a factor of 2, from 0.1%, to 0.2%. Arguably, negligible availability loss should result from 0.2% of the cold data being degraded. We observe a more significant impact on the probability of data loss. Yet, it is still about two orders of magnitude better than in replicated storage. Further, the actual change in durability depends on the coding scheme. For example, Xorbas+LAZY is only 6× more likely to loose data than the original Xorbas scheme.

Figure 9 shows that the bandwidth savings we predict using our failure models are even more modest that what we obtain with actual failure traces [30]. We used trace 19 and 20 from the CFDR repository to inform *ds-sim*'s failure events generator. Each row in the figure represent one trace. As we can see, here lazy recovery achieves significant repair bandwidth savings, up to 20× for the (15-10-12) scheme, with only twice higher number of degraded stripes.

6. RELATED WORK

Erasure coding in general, and repair bandwidth optimization in particular, both have a rich literature; we provide a brief overview of each here, and explain how our solution fits in their context. We then close this section with a description of previous uses of lazy recovery in storage systems.

Erasure coding in storage systems..

Failure Type	Implication	MTBF	MTTR
Latent error	Chunks corrupted	$Exp(\frac{1}{1yr})$ [7]	$W(6, 1000, 3)$ [7] (scrubs)
Disk Failure	Chunks lost	$W(0, 52yrs, 1.12)$ [7]	$W(36s, 108s, 3)$ [31]
Machine Failure	Chunks unavailable	$Exp(\frac{1}{0.33yr})$ [9],traces [30]	GFS traces [23]
Machine Loss	Chunks lost	0.008%/month [31]	Based on spare b/w
Rack Failure	Chunks unavailable	$Exp(\frac{1}{10yrs})$ [9]	$W(10hrs, 24hrs, 1)$ [23]

Figure 6: Storage Component Failure Models

	14-10-13	14-10-12	15-10-12
P(Degraded)	0.006	0.011 (x1.6)	0.012 (x1.8)
	0.005	0.010 (x1.9)	0.011 (x2.2)
Repair b/w	51	4.1 (x12)	2.5 (x20)
TB/day	43	3.2 (x13)	2.3 (x18)

Figure 9: Lazy Recovery Evaluation Using Traces. The number in braces is the improvement over RS(14,10)

Erasure coding schemes have been adopted in both industrial and research storage systems [23, 18, 17, 22, 2, 16, 1], due to their provably optimal storage-durability tradeoff. Among them, Reed-Solomon codes are widely used; however, in the presence of node failures, they require an order of magnitude higher recovery bandwidth compared to replication schemes. To address this problem, recent systems [5, 18, 26] turn to bandwidth-efficient erasure coding as we will describe next.

Bandwidth-efficient erasure coding..

New erasure coding schemes have been proposed to optimize repair bandwidth: Regenerating codes [6] achieve theoretically optimal recovery bandwidth for a given storage footprint, but they are currently impractical since they require splitting data into an exponential number of chunks [29]. Recent work [19] describes the application of bandwidth efficient codes for distributed storage, but they require twice as much storage, and reduce the bandwidth only by half. Another set of codes being adopted at Facebook (Xorbas) [5, 27] reduces bandwidth demands by half for the first failure, with about 15% extra storage cost. Local Reconstruction Codes [18] (LRC) in Windows Azure Storage uses additional parity blocks constructed using subsets of the systematic blocks, which allows repair to be accomplished using fewer block reads on average. Piggiback codes [26] suggest an elegant scheme that reduces the repair bandwidth by 25%.

Instead of designing new coding schemes, our work seeks to reduce repair bandwidth by delaying repair to the time when it is really necessary. We believe that this approach is orthogonal to the choice of the coding scheme and can be combined with any scheme to provide better bandwidth usage. For example, as we have shown in evaluation, it is effective for both Reed-Solomon codes and bandwidth efficient codes like the Xorbas(16,10), and Azure(16,12) codes.

Another idea complementary to our work is to optimize recovery by minimizing the amount of redundant information read from different nodes [20].

Lazy recovery in storage systems.

Lazy recovery is not a new idea in storage systems. Total-Recall [2] introduced the idea of lazy recovery in peer-to-peer storage. Giroire et al. [11] show how to tune the frequency of data repair for peer-to-peer storage systems. Chun et al.[4] show the efficacy of tuning the recovery rate in replicated storage systems in wide area network settings.

As far as we know, our work is the first to apply this idea to erasure coding schemes in data centers, and evaluate it with realistic failure models for this setting.

7. CONCLUSION

In this paper, we show that our refined lazy recovery helps amortize repair bandwidth costs, making erasure coding a viable alternative for cold data storage. Further, we demonstrate the inherently non-linear relationship between repair bandwidth needs and number of degraded stripes, and show that this curve needs to be carefully straddled to achieve the desired reduction of bandwidth overhead while retaining control over degraded stripes (and hence data durability). Our lazy recovery scheme achieves this by dynamically adjusting the recovery threshold for permanent failures depending on the whole system state, while performing lazy recovery of transient ones. We show through simulations that this scheme reduces repair bandwidth by a factor of 4 for the popular Reed Solomon 10-of-14 code—making it comparable to 3-way replication overheads—while retaining high levels of durability and availability.

8. REFERENCES

[1] M. Abd-El-Malek, W. V. C. II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa minor: Versatile cluster-based storage. In *FAST*, 2005.

[2] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker. Total recall: System support for automated availability management. In *NSDI*, volume 4, pages 25–25, 2004.

[3] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[4] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *NSDI*, 2006.

[5] A. Dimakis. Technical talk. `http://ita.ucsd.edu/workshop/12/files/abstract/abstract_764.txt`.

[6] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Trans. Inf. Theor.*, 56(9):4539–4551, Sept. 2010.

[7] J. Elerath and M. Pecht. A highly accurate method for assessing reliability of redundant arrays of inexpensive disks (raid). *Computers, IEEE Transactions on*, 58(3):289 –299, march 2009.

[8] Erasure Coding for Distributed Storage Wiki. `http://csi.usc.edu/~dimakis/StorageWiki/doku.php`.

[9] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *OSDI*, pages 61–74, 2010.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[11] F. Giroire, J. Monteiro, and S. Perennes. Peer-to-peer storage systems: a practical guideline to be lazy. In *GlobeCom*, 2010.

[12] K. Greenan. *Reliability and power-efficiency in erasure-coded storage systems*. PhD thesis, UCSC, 2009.

[13] K. Greenan, E. L. Miller, and J. Wylie. Reliability of xor-based erasure codes on heterogeneous devices. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2008)*, pages 147–156, June 2008.

[14] K. M. Greenan, J. S. Plank, and J. J. Wylie. Mean time to meaningless: Mttdl, markov models, and storage system reliability. In *Proceedings of the 2nd USENIX conference on Hot topics in storage and file systems*, HotStorage'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[15] Hadoop Scalability at Facebook. `http://download.yandex.ru/company/experience/yac/Molkov.pdf`.

[16] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[17] HDFS RAID . `http://wiki.apache.org/hadoop/HDFS-RAID`.

[18] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in windows azure storage. In *USENIX ATC*, 2012.

[19] Y. HuâĂă, H. C. H. ChenâĂă, P. P. C. Lee, and Y. Tang. Nccloud: Applying network coding for the storage repair in a cloud-of-clouds. In *FAST*, 2012.

[20] Y. HuâĂă, H. C. H. ChenâĂă, P. P. C. Lee, and Y. Tang. Rethinking erasure codes for cloud file systems: Minimizing i/o for recovery and degraded reads. In *FAST*, 2012.

[21] R. T. Kaushik and M. Bhandarkar. Greenhdfs: towards an energy-conserving, storage-efficient, hybrid hadoop compute cluster. In *Proceedings of the 2010 international conference on Power aware computing and systems*, HotPower'10, pages 1–9, Berkeley, CA, USA, 2010. USENIX Association.

[22] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[23] Large-Scale Distributed Systems at Google: Current Systems and Future Directions. `http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf`.

[24] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 213–226, Berkeley, CA, USA, 2008. USENIX Association.

[25] D. S. Papailiopoulos, J. Luo, A. G. Dimakis, C. Huang, and J. Li. Simple regenerating codes: Network coding for cloud storage. *CoRR*, abs/1109.0264, 2011.

[26] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *USENIX HotStorage 2013*, 2013.

[27] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. *Proceedings of the VLDB Endowment (to appear)*, 2013.

[28] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *MSST*, 2010.

[29] I. Tamo, Z. Wang, and J. Bruck. Mds array codes with optimal rebuilding. In *ISIT*, pages 1240–1244, 2011.

[30] The computer failure data repository. `http://cfdr.usenix.org`.

[31] The Hadoop Distributed File System. `http://www.aosabook.org/en/hdfs.html`.