On the Erasure Recoverability of MDS Codes under Concurrent Updates[*]

Marcos K. Aguilera
HP Labs, 1501 Page Mill Rd.
Palo Alto, CA 94304, USA

Ramaprabhu Janakiraman
Washington University, One Brookings Dr.
Saint Louis, MO 63130, USA

Lihao Xu
Washington University, One Brookings Dr.
Saint Louis, MO 63130, USA

Abstract— **We consider a fault-tolerant distributed storage system that protects data on k disks using a systematic linear (n, k) MDS code. In such a system, updates to data blocks require corresponding updates to check blocks. Concurrent fault-prone access by multiple writers can drive the system into an inconsistent state with reduced tolerance for disk failures. We show tight bounds on the erasure recoverability of an (n, k) MDS code in this scenario. The bounds depend not just on the minimum distance of the code, but also on the maximum number of concurrent faulty writers and the manner in which they attempt to update the check blocks (one at a time/all at once.)**

I. Introduction

Erasure codes [1] are space-efficient schemes to encode data into redundant fragments to protect against erasures of some of the fragments. Such erasure codes have been used traditionally in communication systems and, more recently, in storage systems as a way to protect data against node crashes (e.g., [2]–[5]). More precisely, an (n, k) MDS erasure code takes as input k *data blocks* and produces $n - k$ additional *check blocks*, for a total of n blocks called a *stripe*. The erasure code guarantees that any k blocks in the stripe can reconstruct the original k data blocks. By storing each block in a separate node, data are protected against the simultaneous failure of up to $n-k$ nodes.

Erasure codes are often used with *immutable* data: the blocks are encoded once, perhaps transmitted, received, and reconstructed in case some blocks are lost. But in this paper, we are interested in *mutable data* stored in a *distributed storage system*. More precisely, data blocks are encoded and stored at different nodes; later some party, a *writer*, may wish to modify one or more of the data blocks; for that, the writer needs to also modify the check blocks to maintain consistency. But because the system is distributed, the writer cannot modify all blocks simultaneously. Therefore, if the writer crashes during these modifications, blocks may be left in an undesirable inconsistent state, from which it is impossible to recover from block erasures. This paper is about how much inconsistency can be tolerated without losing data. The answer depends on the number of writer crashes and erasures, the erasure code, and how the writers update the blocks.

More generally, consider a set of k or fewer writers that want to update one data block each. Suppose that writer i wants to update data block i. Mathematically, let vector $\mathbf{v}$ denote the k data blocks, and $\mathbf{w} = \mathbf{v}\mathbf{G}$ denote the n blocks in a stripe, where $\mathbf{G} = \{g_{i,j}\}$ is the generator matrix for the erasure code in systematic form, and arithmetic is in some finite field. Updating

[*]This work was partially supported by NSF grants CCR-0208975, CNS-0322615, and IIS-0430224.

```
(1) read old contents of data block $i$ from storage node $S_i$
(2) compute $\gamma =$ new content $-$ old content
(3) in some order
    (3.0) tell $S_i$ to add $\gamma g_{i,i} = \gamma$ to its content
    (3.1) tell $S_{k+1}$ to add $\gamma g_{i,k+1}$ to its content
    (3.2) tell $S_{k+2}$ to add $\gamma g_{i,k+2}$ to its content
    ...
    (3.$n-k$): tell $S_n$ to add $\gamma g_{i,n}$ to its content
```

Fig. 1. Protocol for writer i to update data block i.

the i-th data block means changing $\mathbf{v}$ to $\mathbf{v}' = \mathbf{v} + \gamma\mathbf{e_i}$ for some value γ, where $\mathbf{e_i}$ is the i-th canonical basis vector. This requires to change $\mathbf{w}$ to $\mathbf{w}' = \mathbf{w} + \gamma\mathbf{e_i}\mathbf{G}$. In other words, we need to add $\gamma g_{i,j}$ to each check block y_j, for $j = k+1$ to n. Figure 1 shows the sequence of steps to do that in a distributed system where y_j is stored in node S_j.

Note that steps (3.0) to $(3.n-k)$ can be executed in any order, or even in parallel. Also note that the whole protocol can be executed concurrently by multiple writers that update distinct data blocks; after they all finish execution, the check blocks will be consistent with the erasure code. This is true regardless of how the writers interleave their execution. For example, writer 1 (updating data block 1), and writer 2 (updating data block 2) may interleave their execution as follows: writer 1 executes steps 1 to 3.2, then writer 2 executes steps 1 to 3.4, then writer 1 executes its remaining steps, and writer 2 execute its remaining steps. Intuitively, this works because the updates are in terms of "add" operations, which are all commutative.

If a writer crashes after executing some but not all of its steps, is the erasure code still useful to protect blocks against erasures? The answer depends on the exact manner of execution of steps (3.0) to $(3.n-k)$. We consider three possibilities:

- Serial updates: a writer executes these steps one after the other in some given order. For example, first (3.0) then (3.1), (3.2), etc. Therefore, if the writer crashes, some prefix of these steps actually take place.
- Parallel updates: a writer executes steps (3.0) to $(3.n-k)$ all concurrently. If the writer crashes, some arbitrary subset of those steps take place.
- Hybrid updates: the check blocks are divided in chunks of equal size. The writer updates the data block and each chunk in series, but within each chunk blocks are updated in parallel. If the writer crashes, all blocks are updated in all but the last modified chunk; in the last modified chunk, some arbitrary subset of blocks are updated.

Note that the protocol is much faster with parallel updates than with serial updates, and hybrid is in the middle.

We assume that after a writer crashes, it is possible to tell exactly which check blocks it updated before crashing. This can be easily determined by keeping an extra bit per writer at each check block; this bit is set when the writer modifies the block. This assumes that a writer only executes one update; For multiple updates, the writer can flip the bit at each update.

With serial updates, we show that for an MDS code to tolerate d erasures and t writer crashes, it is necessary and sufficient that its minimum distance $d_{min} \geq 2 + (t+1)(d + t/2 - 1)$. Note that when no writers crash, this meets the well-known *Singleton bound*. We also show that the exact order of the serial updates *does* matter. In fact, we show that the near-optimal ordering is for the data block to be updated after half of the check blocks. Thus, with $n - k$ check blocks, the writer should first update $(n-k)/2$ check blocks, then update the data block, then update the remaining check blocks. This increases erasure tolerance by about $t/4$ erasures over the original method where the data block is updated first.

With parallel updates, the corresponding bound is $d_{min} \geq 2 + 2^t(d + t/2 - 1)$, and so much more powerful codes are needed than with serial updates. Intuitively, this is so because when t writers crash, the check blocks may be left in one of 2^t states. And with hybrid updates, we show that with s groups of size at most $r = \lceil \frac{n-k}{s} \rceil$ each, the erasure recoverability is the same as with serial updates for sufficiently large s. Our proofs are based on combinatorial arguments explained through balls-in-bins games where an adversary is trying to maximize the damage.

To summarize, in this paper we consider MDS erasure codes in a setting where multiple parties want to update different data blocks simultaneously, and they may fail by crashing while doing so (and meanwhile some blocks may be erased as well.) We consider three schemes that differ on how blocks are updated: in series, in parallel, or hybrid. For each scheme, we show how many erasures can be tolerated as a function of the erasure code parameters and the number of writer crashes. It is worth mentioning that the distributed update model in this paper is the core of a distributed storage system that we built [6], which allows not only allows concurrent updates, but also online reconstruction of data and concurrent reads.

The paper is organized as follows. We first describe related work in Section II. In Section III we describe our model and assumptions. Our main results are given in Sections IV and V. We conclude with some comments and open questions in Section VI.

II. Related Work

The simplest application of erasure codes to storage is for archival storage, where some immutable data needs to be preserved for a long time (e.g., [2]). In those cases, there is no need to worry about data updates. Myriad [3] proposes erasure codes for *disaster tolerance*. Updates of check blocks do not happen during a write, but are instead deferred and done in batches that are applied atomically. This ensures that all check blocks are updated consistently, but it requires the use of expensive two-phase commit protocols to ensure atomicity. Zhang and Lian [7] also propose a general scheme to use erasure codes for distributed storage. However, this scheme does not handle concurrent updates; instead, it assumes some external mechanism, like a transaction for each operation. This appears to be expensive, but no performance data is provided. LH^*_{RS} [8] use erasure codes to implement an expandable and distributed data structure, where redundancy increases with the amount of data, to ensure a minimum availability. However, the paper does not address crashes of writers. Other storage systems that allow concurrent updates are proposed in [4], [5]. In both of these works, when data is updated the storage nodes keep around old versions of data, so recovery becomes very easy. However, by keeping many versions of data, those schemes lose the space-optimality of erasure codes.

III. Model

We consider a systematic linear (n, k) MDS erasure code that encodes k *data blocks* into $n - k$ *check blocks* for a total of n blocks.

Those n blocks are kept in distinct storage nodes that are accessible via a network. Storage nodes accept requests to manipulate data from a number of readers and writers in the system. The goal is to keep the blocks consistent with the erasure code, so that if some storage nodes erase their data, data can be recovered.

Data is *mutable*: there are k or fewer *writers* in the system that want to update a distinct data block in the erasure code. Writers may execute concurrently, and we assume that writer i updates data block i, perhaps multiple times. The system is *asynchronous*: some writers may execute faster than other writers, and there are no bounds on their relative speeds. To support concurrent updates, a writer executes the steps in Figure 1. It is easy to show that after all writers have completed those steps, (1) the data blocks will contain the updated values, and (2) the check blocks will be consistent with the erasure code. This is true regardless of how the concurrent execution interleaves.

However, the system is subject to various failures. A writer may fail by *crashing*, which causes it to stop executing its steps (forever). As a result, the writer may execute some but not all steps in Figure 1. Such a crash of the writer causes a *fork* in the encoded data. Furthermore, a storage node may inadvertently lose its data, and we assume that this loss is detectable (instead of assuming that the data undetectably changes to some unknown garbage). This is called a *data erasure*. If a data block is erased, we consider it to be *recoverable* if it is possible to retrieve either (1) its value before it was updated by a writer, or (2) its value after it was updated by a writer W. Moreover, if writer W does not crash, then only (2) is allowable.

IV. Forks, Erasures, and Erasure Recoverability

We now analyze in detail the dependency between the number of forks, number of erasures, and erasure recoverability for a given (n, k) code.

Assume that the system starts from a valid codeword. Subsequently, writers update individual data blocks any number of times. During this execution, t writers crash, and d blocks are erased[*]. Then, we observe the state of the system after this execution completes, and try to recover the erased blocks. If for a given t and d it is possible to recover all blocks in all executions, we say that the (n, k) code is (t, d)-*recoverable*.

The goal of this section is to find values of d for a given value of t such that an (n, k) code is (t, d)-recoverable. Clearly,

[*]While executing, if a writer finds that a data block is erased in protocol step (1), it aborts execution. If it finds that a block is erased in protocol step (3.x), it ignores that step and continues execution.

if $t = 0$ then $d \leq n - k$, which is the erasure correcting limit of an (n, k) MDS code.

Our proofs consider a *balls-into-bins* game where each bin represents a state indicating the outcome of each write attempt by a crashed writer. With t crashed writers, this state can be represented using t bits, and the i^{th} state bit indicates whether the last write of the i^{th} crashed writer is discarded (bit is 0) or recovered (bit is 1). Each ball represents the observed value of a block at the end of execution. A ball goes into a bin if and only if the value of the corresponding block can be correctly used toward recovery to the state represented by the bin. Note that such a recovery might not be possible, if there are not enough balls in that bin.

Also, a ball can go into multiple bins if the corresponding blocks can be used to recover to multiple states. For example, balls corresponding to data blocks that no crashed writer intended to modify go into all bins. In general, a data block may go into multiple bins, while each check block goes into a single bin. As we shall see in Section V, this difference between data blocks and check blocks can be utilized for greater erasure recoverability.

Starting from a valid codeword and picking a combination of forks and erasures, an adversary can place the n balls into bins (some of them multiple times), and then remove d distinct balls (if a ball is in multiple bins, all instances are removed.) If, after the adversary is done, at least one bin has k or more balls, then the state is recoverable. If, for a given (t, d), all possible strategies for the adversary result in recoverable states, then the code is (t, d)-recoverable.

Lemma 1 *Consider two cases where forks and erasures occur in some combination: 1) forks happen first, followed by erasures. 2) forks and erasures can occur in any order. Recoverablity under case 1 implies recoverability under case 2.*

Proof. This is clear since the erasure of a block does not alter the subsequent update of another block. ■

This property allows us to safely assume in the rest of this section that all forks precede erasures.

Now, we can proceed to obtaining upper bounds for d. We first consider the following two variations of the algorithm:

SERIAL UPDATE: Each writer performs the steps of Fig 1 in serial order. If the writer crashes, an arbitrary **prefix** of check blocks would have been updated.

PARALLEL UPDATE: Each writer performs the update steps in parallel. If the writer crashes in the middle, an arbitrary **subset** of check blocks could have been updated.

Theorem 1 *An (n, k) code with serial-update is (t, d)-recoverable if and only if $d \leq d_{\text{SERIAL}} = \lceil \frac{n-k}{t+1} - \frac{t}{2} \rceil$.*

Proof. When t writers crash under serial update, we can order them into $q \leq t$ groups, where all writers crashing after updating the same number of check blocks go into the same group. Let $x_1, x_2, \cdots, x_q$ of the writers crash after updating $a_1, a_1+a_2, \cdots, a_1+a_2+\cdots+a_q$ check blocks, with a_{q+1} check blocks untouched by any writer. Now, $x_1 + x_2 + \cdots + x_q = t$ and $a_1 + a_2 + \cdots + a_{q+1} = n - k$. Let bin i represent the set of writes by all crashed writers not in the first $i - 1$ groups. Thus there are $q+1$ bins, which partition the set of $n-k$ check blocks into $q + 1$ sets of size $a_1, a_2, \cdots a_{q+1}$. All check blocks in set i go into bin i. The $k - t$ data blocks untouched by any crashed writer go into all bins. Also, $t - \sum_{j=0}^{i-1} x_j$ of the data blocks go into bin i (define $x_0 = 0$), since this is the number of data blocks useful in restoring to a state in which the writes of all crashed writers in the previous $i - 1$ groups are lost.

Thus the number of balls in bin i are $k + a_i - \sum_{j=0}^{i-1} x_j$. Thus the total number of balls is:

$$\sum_{i=1}^{q+1}(k + a_i - \sum_{j=0}^{i-1} x_j) = (q + 1)k + n - k - \sum_{i=1}^{q}(q - i + 1)x_i$$
$$\geq (q + 1)k + n - k - q(t - q) - q(q + 1)/2$$
$$\geq (q + 1)k + n - k - t(t + 1)/2 \tag{1}$$

Since there are $q + 1$ bins, the *average* number of balls per bin is:

$$\frac{\#\text{balls}}{\#\text{bins}} \geq k + \frac{n - k - t(t + 1)/2}{q + 1}. \tag{2}$$

Since $q \leq t$, there is ≥ 1 bin with $\geq k + \lceil \frac{n-k}{t+1} - \frac{t}{2} \rceil$ balls. Thus, if $k + d$ is no greater than this value, removing d balls from each bin would still leave at least one bin with k or more balls, and hence the code would be (t, d)-recoverable.

This is also a tight lower bound on the number of balls in each bin, i.e., the adversary has a strategy for placing balls in bins such that no bin has more than $k + d_{\text{SERIAL}}$ balls [6]. ■

Theorem 2 *An (n, k) code under parallel-update is (t, d)-recoverable if and only if $d \leq d_{\text{PARALLEL}} = \lceil \frac{n-k}{2^t} - \frac{t}{2} \rceil$.*

Proof. The proof is similar to the previous one: since writers can update any subset of check blocks before crashing, there are now 2^t bins, with each bin tagged by the bit representation $a_1...a_t$, a_i being 0 or 1 according to whether the i^{th} crashed writer's write is lost or restored when going back to that state.

For this case, each of the $k-t$ untouched data blocks go into every bin. For each data block i that crashed writers intended to update, add a ball to each bin for which the corresponding bit a_i is 0 or 1 according to whether that writer updated the data block itself (each such block goes to half of the bins). In addition, there are the $n - k$ check blocks that go into distinct bins depending on the set of writers that updated it.

Now the total number of balls is given by $(k - t)2^t + \frac{t}{2}2^t + n - k = k2^t - t2^{t-1} + n - k$. From this, the 'if' part clearly follows.

The bound is also tight i.e., a strategy exists for the adversary to place balls in bins such that no bin has more than $k+d_{\text{PARALLEL}}$ balls. A greedy strategy by the adversary suffices [6]. ■

Hybrid-updates: latency versus erasure recoverability

In the previous section, we saw that with serial updates we get much better erasure recovery capability than with parallel updates. However, the downside is that with serial updates the latency of *each write* operation can be much higher.

One compromise that suggests itself is to divide the check blocks into s check blocks of at most r blocks each, where $r = \lceil \frac{n-k}{s} \rceil$. Blocks in a single chunk are updated in parallel, but the chunks themselves are updated serially. This results in an (s, r)-hybrid-update. In this section, we will derive bounds on the erasure correcting capability of this *hybrid-update* scheme.

Theorem 3 *An (n, k) code under (s, r)-hybrid-update is (t, d)-recoverable if $d \leq d_{\text{SERIAL}}$ when $r \leq d_{\text{SERIAL}}$.*

Proof. As before, associate a bin with the bit representation of whether each crashed writer's write was incorporated or not. The $k-t$ untouched data blocks go into all bins.

Let $C_1,\ldots,C_q$ be the chunks where some writer crashed while trying to update, ordered by update order of the protocol (i.e., C_1 is updated before C_2, etc). Note that $q \le s$. For $i=1\ldots q$, let X_i be the set of writers that crashed while trying to update C_i, and let $x_i = |X_i| \neq 0$. Define $X_0 = \emptyset$. For $i = 1\ldots q$, let A_i be the set of check blocks updated by all processes in X_i and no processes in X_{i-1}, and let $a_i = |A_i|$. For convenience, also define A_{q+1} to be the set of check blocks updated by no crashed writer. Let B_i be the set of check blocks updated by some but not all processes in X_i, and let $b_i = |B_i|$.

Note that (I) a block in A_i was updated by all processes in $\cup_{j\ge i} X_j$ and by no processes in $\cup_{j<i} X_j$, because chunks are updated in series, and (II) A_i's and B_i's together partition the set of all check blocks. Each block in A_i or B_i goes in exactly one bin, because it is a check block. Moreover, the set of all blocks in B_i go in at most $2^{x_i}-2$ bins, and the set of all blocks in A_i go in at most 1 bin. Therefore, the set of all blocks in $\cup_{i=1}^{q+1} A_i$ go in at most $q+1$ bins.

Let λ be the number of X_i's with more than one crashed writer. Since (*) every X_i has at least one crashed writer and there are q X_i's and (**) t is the total number of crashed writers, we have $q+\lambda \le t$. Thus, $q \le t-\lambda$. Also, $b_i = 0$ whenever $x_i < 2$, and $b_i \le r$ for all i, hence (III) $\sum_{i=1}^{q} b_i \le \lambda r$. By (II) and (III) we have $\sum_{i=1}^{q+1} a_i = n-k-\sum_{i=1}^{q} b_i \ge n-k-\lambda r$.

Let $d(\lambda)$ be a lower bound on the number of balls in any bin for given λ, i.e., no strategy exists for the adversary to place balls in bins such that all bins contain fewer than $d(\lambda)$ balls. Then, by counting the balls of blocks in all $A_i's$, as in (1) but with the additional constraint that at least λ of the x_i are >1,

$$\begin{aligned}
\#\text{balls} &= (q+1)k + n - k - \sum_{i=1}^{q} b_i - \sum_{i=1}^{q}(q-i+1)x_i \\
&\ge (q+1)k + n - k - \lambda r - \sum_{i=1}^{q}(q-i+1)x_i \\
&\ge (q+1)k + n - k - \lambda r - q(t-q) - \frac{q(q+1)}{2} + \frac{\lambda(\lambda-1)}{2} \\
&\ge (q+1)k + n - k - \lambda r - \frac{t(t+1)}{2} + \lambda^2 \;\; \{q \le t-\lambda\}
\end{aligned} \tag{3}$$

Since there are at most $q+1$ bins to put these balls, and $q \le t-\lambda$,

$$d(\lambda) \ge k + \left\lceil \frac{n-k-\lambda r - \frac{t(t+1)}{2} + \lambda^2}{t-\lambda+1} \right\rceil \tag{4}$$

Now consider $\lambda = 0$, i.e., $x_i = 1$ for all i so that writers always crash singly in distinct chunks. In this case, there exists a tight bound $d(0) = k + d_{\text{SERIAL}}$. Now we prove $d(\lambda) \ge d(0)$ for $\lambda \ge 0$ when $r \le d_{\text{SERIAL}}$:

$$\begin{aligned}
&\text{Let } 0 \le \epsilon = d_{\text{SERIAL}} - \left(\frac{n-k}{t+1} - \frac{t}{2}\right) < 1 \\
&d(\lambda) - d(0) \ge \left\lceil \frac{n-k-\lambda r - \frac{t(t+1)}{2} + \lambda^2}{t-\lambda+1} \right\rceil - d_{\text{SERIAL}} \\
&\ge \frac{n-k-\lambda r - \frac{t(t+1)}{2} + \lambda^2}{t-\lambda+1} - \frac{n-k-\frac{t(t+1)}{2}}{t+1} - \epsilon \\
&= \left\{\frac{n-k}{t+1} - \frac{t}{2} - r + \lambda\right\}\left\{\frac{\lambda}{t-\lambda+1}\right\} - \epsilon \\
&\ge 0, \quad \text{for } r \le d_{\text{SERIAL}}, \lambda \ge 0, \epsilon < 1 \text{ and integer } d(\lambda).
\end{aligned} \tag{5}$$

Thus $d(\lambda) \ge d(0)$ for $\lambda \ge 0$. In other words, the strategy of making each writer crash on a distinct chunk is optimal for the adversary. Hence, for $r \le d_{\text{SERIAL}}$, $d(0)$ is a lower bound on minimal number of balls in any bin. In this case, up to $d_{\text{SERIAL}} = \lceil \frac{n-k}{t+1} - \frac{t}{2} \rceil$ erasures can be tolerated. Hence, when $r \le d_{\text{SERIAL}}$ and $d \le d_{\text{SERIAL}}$, the code is (t,d)-recoverable. ■

We note that with enough chunks, i.e., when $r \le d_{\text{SERIAL}}$, the hybrid-update scheme has as good erasure tolerance as the serial-update scheme, but with lower update latency. How about when $r > d_{\text{SERIAL}}$? In this case, the analysis is more difficult, and we have yet to obtain a tight bound, but have devised a dynamic programming solution that computes erasure recoverability in $O(t(n-k)^2)$ time.

Here is an interesting application of the above results. Assume that we are given $t \ge 0$ and $d > 0$, and we want to estimate (1) how many check blocks δ are needed to make the code (t,d)-recoverable, and (2) the write latency ρ (in network round trips to storage nodes). This is a simple corollary of the results above:

Corollary 1 *We have*

1) *for serial updates:* $\delta = 1+(t+1)(d+t/2-1)$; $\rho = 2+\delta$;
2) *for parallel updates:* $\delta = 1+2^t(d+t/2-1)$; $\rho = 2$.
3) *for hybrid updates with* $r = d_{\text{SERIAL}}$*:* $\delta = 1+(t+1)(d+t/2-1)$; $\rho = 2+\lceil \frac{\delta}{d_{\text{SERIAL}}} \rceil$.

V. Does Update Order Matter?

In the previous section, we have assumed that writers using serial updates executes steps (3.x) in the order shown: (3.0), (3.1), etc. Thus, the data block is updated before any check blocks. Other orders are possible, like (3.1), (3.2), (3.0), etc. Are these orders all equivalent?

Since check blocks are symmetric for a linear MDS code, the only thing that matters is *when* the data block is updated. In the δ_1*-serial-update* scheme, a writer updates δ_1 check blocks in some fixed order, followed by the data block, followed by the remaining $\delta_2 = n-k-\delta_1$ check blocks in some fixed order. We show that the choice of δ_1 affects erasure recoverability. Intuitively, this is because data blocks are more valuable than check blocks as the former may be used to recover to multiple states: in our balls-and-bins scenario, balls corresponding to data blocks may appear in multiple bins, which is not true for check blocks.

Thus, we ask: (a) What is the erasure recoverability for a given δ_1? (b) What is the erasure recoverability for an optimal choice of δ_1?

Theorem 4 *Let* $D(x,y) = \lceil \frac{x}{y+1} - \frac{y}{2} \rceil$. *An* (n,k) *code under* δ_1*-serial-update is* (t,d)*-recoverable if and only if* $\forall t_1 \in \{0,\cdots,t\}$

$$\begin{aligned} d \le \max\{D(\delta_1,t_1), D(\delta_2,t_2), D(n-k+t_1t_2,t)\} \\ \textit{where } \delta_2 = n-k-\delta_1 \quad t_2 = t-t_1 \end{aligned} \tag{6}$$

Proof sketch. For δ_1-serial updates, the analysis is very similar to that for the serial case. Divide the set of crashed writers into two sets Y and Y': Y (of size t_1) has writers that crashed before updating their data blocks and Y' (of size $t_2 = t-t_1$) has the other crashed writers. Divide the set of check blocks into two (possibly overlapping) sets X and X': X (of size at least δ_1) contains check blocks updated by all writers in Y', and X' (of size at least $\delta_2 = n - k - \delta_1$) contains check blocks updated by no writers in Y.

We can bound the recoverability by independently applying to these two sets the analysis as for the original serial update case, but with parameters (δ_1, t_1) and (δ_2, t_2) instead of $(n-k, t)$. Note that $D(n-k, t)$ is just d_{SERIAL}. To get a bound of $D(n-k+t_1t_2, t)$ in the third term of the max in the theorem, we count the total number of balls in all bins: we have $\delta_1 - t_1(t_1+1)/2$ balls in bins corresponding to Y, and $\delta_2 - t_2(t_2+1)/2$ balls in bins corresponding to Y'. We add these together and apply the analysis for the original serial update case again where the maximum total number of bins is $t+1$ (not $t_1+1+t_2+1 = t+2$).

Finally, since the adversary will choose a worst-case t_1, it has to hold for all valid t_1. ∎

Theorem 5 *An (n,k) code under δ_1-serial-update is (t, D_{min})-recoverable for some δ_1 and at most (t, D_{max})-recoverable for any δ_1, where:*

$$D_{min} = \lceil \frac{n-k}{t+1} - \frac{t+1}{4} \rceil \text{ and } D_{max} = \lceil \frac{n-k}{t+1} - \frac{t-1}{4} \rceil \quad (7)$$

Proof. Given the bound on tolerable erasures for a given δ_1 from the previous result, we can formulate the proof in terms of the following game: assume the "good guy" or mover picks $\delta_1, \delta_2 : \delta_1 + \delta_2 = n - k$; the adversary then picks $t_1, t_2 : t_1 + t_2 = t$. In this game, the payoff to the mover is $\max\{D(\delta_1, t_1), D(\delta_2, t_2), D(n-k+t_1t_2, t)\}$. The proof proceeds in two parts:

1) **Upper bound:** Show that for each choice of (δ_1, δ_2) by the mover, the adversary has a feasible choice of (t_1, t_2) such that the payoff to the mover, $D \leq D_{max}$.
2) **Lower bound:** Show that a choice $(\delta_1, \delta_2) = (\lfloor \frac{n-k}{2} \rfloor, \lceil \frac{n-k}{2} \rceil)$ has payoff $D \geq D_{min}$.

Proof of upper bound: For a given choice of δ_1, δ_2, let the adversary pick t_1, t_2 such that:

$$t_i \geq X_i = \sqrt{(D_{max} - \frac{1}{2})^2 + 2\delta_i} - (D_{max} + \frac{1}{2}), \quad i = 1, 2$$

$$\text{Then, } (t_i + D_{max} + \frac{1}{2})^2 \geq (D_{max} - \frac{1}{2})^2 + 2\delta_i$$

$$\Rightarrow t_i(t_i+1) + 2D_{max}(t_i+1) \geq 2\delta_i$$

$$\Rightarrow \frac{\delta_i}{t_i+1} - \frac{t_i}{2} \leq D_{max} \Rightarrow \lceil \frac{\delta_i}{t_i+1} - \frac{t_i}{2} \rceil \leq D_{max}, \quad i = 1, 2$$

Thus $D(\delta_1, t_1) \leq D_{max}$ and $D(\delta_2, t_2) \leq D_{max}$. Also, $D(n-k+t_1t_2, t)$ reaches its maximum value for $t_1 = t_2 = t/2$ where it is $\leq D_{max}$. Hence if such a choice of t_1, t_2 exists, $D \leq D_{max}$. It remains to be shown that the adversary has a feasible choice of *integers* $0 \leq t_1 \leq t, 0 \leq t_2 \leq t, t_1 + t_2 = t$ satisfying both constraints above.

Given that $\delta_1 + \delta_2 = n - k$, it can be verified that $X_1 + X_2$ reaches maximum value for $\delta_1 = \delta_2 = \frac{n-k}{2}$. Hence,

$$X_1 + X_2 \leq 2\left\{\sqrt{(D_{max} - \frac{1}{2})^2 + n - k} - (D_{max} + \frac{1}{2})\right\}$$

$$\frac{n-k}{t+1} - \frac{t-1}{4} \leq D_{max} \Rightarrow n - k \leq \frac{t^2-1}{4} + D_{max}(t+1)$$

$$\Rightarrow X_1 + X_2 \leq t - 1$$

Furthermore, when $\delta_1 < D_{max}$ or $\delta_2 < D_{max}$, a trivial strategy exists for the adversary such that $D \leq D_{max}$ by setting $t_1 = 0$ or $t_2 = 0$ respectively, hence we can safely assume that all non-trivial choices for the mover involve $\delta_1 \geq D_{max}$ and $\delta_2 \geq D_{max}$. In this case, it is clear that $X_1 \geq 0$ and $X_2 \geq 0$.

The constraints $t_1 \geq X_1$, $t_2 \geq X_2$, and $t_1 + t_2 = t$ form an isosceles right triangle in the first quadrant with axes t_1 and t_2. All feasible choices of t_1 and t_2 for the adversary lie on its hypotenuse of length ℓ, where $\ell^2 = 2(t - X_1 - X_2)^2 \geq 2$, i.e., $\ell \geq \sqrt{2}$. Thus the hypotenuse passes through at least one pair of integer points (t_1, t_2), and hence the adversary has a strategy that for any choice of δ_1 and δ_2 by the mover, there exists a choice t_1 and t_2 such that the payoff $D \leq D_{max} = \lceil \frac{n-k}{t+1} - \frac{t-1}{4} \rceil$.

Proof of lower bound: Let the mover pick $\delta_1 = \lfloor (n-k)/2 \rfloor$ and $\delta_2 = \lceil (n-k)/2 \rceil$. The best strategy for the adversary clearly is to pick $t_1 = \lfloor \frac{t}{2} \rfloor$ and $t_2 = \lceil \frac{t}{2} \rceil$. In this case, the total number of balls $\geq n - k - t_1(t_1+1)/2 - t_2(t_2+1)/2$ and total number of bins $\leq t+1$. Hence,

$$D \geq \lceil \frac{n-k}{t+1} - \frac{t}{2} + \frac{t^2-1}{4(t+1)} \rceil = \lceil \frac{n-k}{t+1} - \frac{t+1}{4} \rceil = D_{min}$$

∎

Remarks: The recoverability of a "half-half" split is at most one erasure away from optimal. In many cases (e.g., When $n-k$ is even, and t is odd), it is optimal. A more precise analysis reveals it is optimal in $\geq 75\%$ of cases when both $n-k$ and t are odd and and $\geq 50\%$ of cases when t is even. Loosely speaking, such a modified serial scheme tolerates about $t/4$ more erasures compared to the original serial scheme.

VI. Concluding remarks

Erasure codes are increasingly finding use in storage applications. We have considered a typical system where distributed storage is protected from data loss by using systematic linear MDS codes. Clients accessing storage try to maintain consistency by updating check blocks when writing to a data block, but may fail partway due to system crashes (e.g., O/S failure) or component failures (e.g., hardware failure.)

In this paper we have studied how concurrent partial updates affect the erasure recoverability of an MDS code. We obtain tight bounds relating the number of writer crashes, the number of erasures, and the minimum distance of the MDS code used.

Some questions naturally beg to be answered: 1) What are the corresponding bounds for non-MDS codes, and can they be related simply to the minimum distance of the code? 2) Can these bounds be extended to *array* codes, where data blocks and check blocks of a codeword may co-exist on a disk?

References

[1] V. Pless, *Introduction to the Theory of Error-Correcting Codes*, Wiley-Interscience, 1998.

[2] J. Kubiatowicz et al., "Oceanstore: An architecture for global-scale persistent storage," in *ASPLOS*, 2000.

[3] F. Chang et al., "Myriad: Cost-effective Disaster Tolerance," in *FAST*, 2002.

[4] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch, "A decentralized algorithm for erasure-coded virtual disks," in *DSN*, 2004.

[5] G. R. Goodson, J. J. Wylie, G. R. Ganger, , and M. K. Reiter, "Efficient byzantine-tolerant erasure-coded storage," in *DSN*, 2004.

[6] M. K. Aguilera, R. Janakiraman, and L. Xu, "Using erasure codes efficiently for storage in a distributed system," Tech. Rep., Available at http://www.nisl.wustl.edu/Papers/Tech/iguana.pdf, Dec 2004.

[7] Z. Zhang and Q. Lian, "Reperasure: Replication protocol using erasure-code in peer-to-peer storage," in *SRDS*, 2002.

[8] W. Litwin and T. Schwarz, "LH* RS : A high-availability scalable distributed data structure using reed solomon codes," in *SIGMOD*, 2000.