Non-local redundancy: Erasure coding and dispersed replicas for robust retrieval in the Swarm peer-to-peer network

VIKTOR TRÓN*, Swarm Research Division, Switzerland
VIKTOR TÓTH and GYÖRGY BARABÁS, Division of Biology, Dept. IFM, Linköping University, Sweden
CALLUM TONER, DANIEL NICKLESS, DÁNIEL A. NAGY, and ÁRON FISCHER

This paper presents an architecture for achieving resilient content retrieval in the Swarm decentralised storage network using erasure coding and non-local replication. The objective is to guarantee high data availability despite partial node failure or missing chunks. The proposed approach integrates Reed–Solomon coding into the Swarm hash tree, enabling the reconstruction of original data from a subset of stored chunks. Parametrised redundancy schemes are introduced to meet varying reliability requirements under various assumed rates of chunk retrieval failure. Dispersed replication of singleton chunks extends fault tolerance to edge cases. The paper also proposes retrieval strategies that exploit redundancy to optimise latency and cost. The result is a modular and practical design for robust file storage ideally suited for the adversarial and unreliable context of peer-to-peer decentralised networks.

CCS Concepts: • **Networks** → **Network algorithms**; **Network protocols**; • **Information systems** → **Information storage systems**.

Additional Key Words and Phrases: Web3, Swarm, Decentralised storage, Erasure coding, Reed–Solomon encoding

ACM Reference Format:
Viktor Trón, Viktor Tóth, György Barabás, Callum Toner, Daniel Nickless, Dániel A. Nagy, and Áron Fischer. 2025. Non-local redundancy: Erasure coding and dispersed replicas for robust retrieval in the Swarm peer-to-peer network. 1, 1 (April 2025), 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

This paper is structured as follows. Section 1 introduces the role of erasure codes[1] Section 2 details the integration of Reed–Solomon encoding into the Swarm hash tree and explains how parity chunks are incorporated at each layer. Section 3 provides a formal framework for selecting the number of parity chunks needed to meet predefined security thresholds, based on probabilistic failure models. Section 4 addresses the special case of single-chunk redundancy by introducing dispersed replicas. Section 5 outlines multiple retrieval strategies tailored for erasure-coded content, analysing their trade-offs in terms of latency, cost, and robustness before section 6 concludes.

*Corresponding author
[1]Error correcting codes that have a focus on correcting data loss are referred to as *erasure codes*, a typical scheme of choice for distributed storage systems [1].

Authors' addresses: Viktor Trón, viktor@ethswarm.org, Swarm Research Division, Neuchâtel, Switzerland; Viktor Tóth, nugaon@ethswarm.org; György Barabás, gyorgy.barabas@liu.se, Division of Biology, Dept. IFM, Linköping University, Linköping, Sweden; Callum Toner, callum@ethswarm.org; Daniel Nickless, dan@ethswarm.org; Dániel A. Nagy, daniel@ethswarm.org; Áron Fischer, aron.fischer@gmail.com.

Manuscript submitted to ACM

1 INTRODUCTION

Error correcting codes are widely utilised in the context of data storage and transfer to ensure data integrity in the face of a system fault. Error-correction schemes define how to rearrange original data by adding redundancy to its representation before upload or transmission, (*encoding*) so that corrupted data can be corrected or missing content recovered upon retrieval or reception (*decoding*). The different schemes are evaluated by quantifying their strength (*tolerance*, in terms of the rate of data corruption and loss) as a function of their cost (*overhead*, in terms of storage and computation).

Specifically in the field of computer hardware architecture, synchronising arrays of disks is crucial for providing resilient storage in data centres.

In *erasure coding*, however, the problem can be framed as follows: How does one distribute data into *shards* stored across the physical disks of an array or the physical nodes of a server cluster so that the data remains fully recoverable in the face of an arbitrary probability that any one or more physical carriers become faulty?

Similarly, in the context of Swarm's distributed immutable *chunk* store, DISC, the problem can be reformulated as follows: How does one encode the stored data into chunks distributed across *neighbourhoods*[2] in the network so that the data remains fully recoverable in the face of an arbitrary probability that any one chunk is not retrievable?[3]

Reed-Solomon coding (RS) [2, 5, 8, 9] is the father of all error correcting codes (ECC) and also the most widely used and implemented.[4] When applied to data of m fixed-size blocks (message of length m), it produces an encoding of $m + k$ *codewords* (blocks of same size) in such a way that obtaining any m out of $m + k$ blocks is enough to reconstruct the original data. Conversely, k puts an upper bound on the number of *erasures* allowed (the number of unavailable blocks) for full recoverability, i.e., it expresses (the maximum) *loss tolerance*.

k is also the count of *parities*, quantifying the data blocks added during the encoding on top of the original count of blocks, in other words, it expresses the *storage overhead*. While RS is optimal for storage (since loss tolerance cannot exceed the storage overhead), it has high bandwidth demands[5] for local repair processes.[6] The decoder needs to retrieve m chunks to recover a particular unavailable chunk. Hence, ideally, RS is used on files which are supposed to be downloaded in full,[7] but it is inappropriate for use cases needing only local repairs.[8]

When using RS, it is customary to use *systematic* encoding, which means that the original data forms part of the encoding, with the parities added to it.[9]

2 ERASURE CODING IN THE SWARM HASH TREE

Swarm uses the *Swarm hash tree* to represent files. This structure is a Merkle tree [6], whose leaves constitute the consecutive segments of the input data stream. These segments are turned into chunks and are distributed among the Swarm nodes for storage. The consecutive chunk references (either in the form of an address or an address and an encryption key) are written into a chunk at a higher level. These so-called *packed address chunks* (PACs) constitute the intermediate chunks of the tree. The branching factor b is chosen so that the references to its children fill up a

[2] Neighborhoods are groups of nodes which are responsible for sharing the same chunks.
[3] We will assume that the retrieval of any one chunk fails with equal and independent probability.
[4] For a thorough comparison of an earlier generation of implementations of RS, see Plank et al. [7].
[5] Both the encoding and the decoding of RS codes takes $O(mk)$ time (with m data chunks and k parities). However, we found computational overhead both insignificant for a network setting, as well as undifferentiating.
[6] Entanglement codes [3, 4] require a minimal bandwidth overhead for a local repair, but at the cost of storage overhead that is in the multiples of 100%.
[7] Or in fragments large enough to include the data span over which the encoding is defined, such as videos.
[8] E.g., Use cases requiring random access to small amounts of data (e.g., path lookups) benefit from simple replication, instead of RS, to optimise on bandwidth. Replication is, of course, suboptimal in terms of storage [10].
[9] Our library of choice implementing exactly such a scheme is https://github.com/klauspost/reedsolomon.

full chunk. With a reference size of 32 or 64 (hash size 32) and a chunk size of 4096 bytes, the value of b is 128 for unencrypted, and 64 for encrypted content (Figure 1).

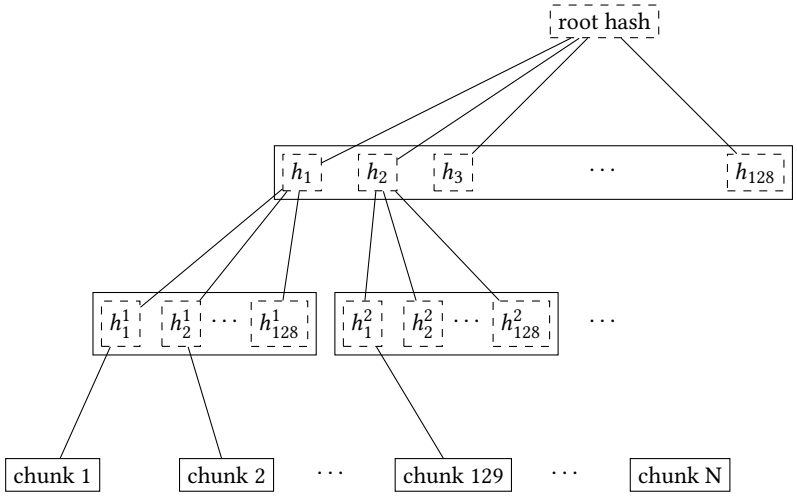

Fig. 1. The Swarm tree is the data structure encoding how a document is split into chunks.

Note that on the right edge of the hash tree, the last chunk of each level may be shorter than 4K: in fact, unless the file is exactly $4 \cdot b^n$ kilobytes long, there is always at least one *incomplete chunk*. Importantly, it makes no sense to wrap a single chunk reference in a PAC, so it is attached to the first level where there are open chunks. Such *"dangling" chunks* will appear if and only if the file has a zero digit in its b-ary representation.

During file retrieval, a Swarm client starts from the root hash reference and retrieves the corresponding chunk. Interpreting the metadata as encoding the span of data subsumed under the chunk, it decides that the chunk is a PAC if the span exceeds the maximum chunk size. In case of standard file download, all the references packed within the PAC are followed, i.e., the referenced chunk data is retrieved.

PACs offer a natural and elegant way to achieve consistent redundancy within the Swarm hash tree. The input data for an instance of erasure coding is the chunk data of the children, with the equal-sized bins corresponding to the chunk data of the consecutive references packed into it. The idea is that instead of having each of the b references packed represent children, only m would, and the rest of the $k = b - m$ would encode RS parities (see Figure 2).

The *chunker* algorithm that incorporates PAC-scoped RS encoding would work as follows:

(1) Set the input to the actual data level and produce a sequence of chunks from the consecutive 4K segments of the data stream. Choose m and k such that $m + k = b$ is the branching factor (128 for unencrypted, and 64 for encrypted content).
(2) Read the input one chunk at a time. Count the chunks by incrementing a counter i.
(3) Repeat Step 2 until either $i = m$ or there is no more data left.
(4) Use the RS scheme on the last $i \leq m$ chunks to produce k parity chunks resulting in a total of $n = i + k \leq b$ chunks.

(5) Concatenate the references of all these chunks to result in a packed address chunk (of size $h \cdot n$) on the level above. If this is the first chunk on that level, set the input to this level and spawn this same procedure from Step 2.

(6) When the input is consumed, signal the end of input to the next level and quit the routine. If there is no next level, record the single chunk as the root chunk and use the reference to refer to the entire file.

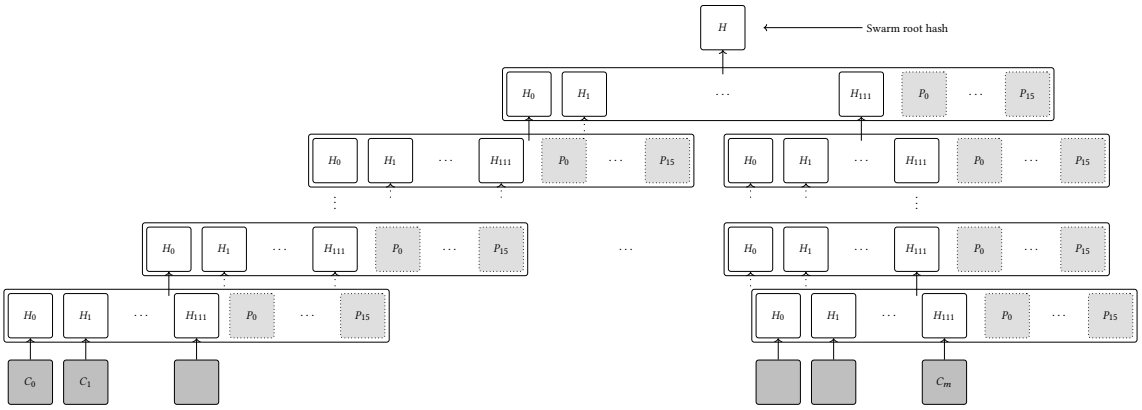

Fig. 2. The Swarm tree with extra parity chunks using $m = 112$ out of $n = 128$ RS encoding. Chunks P_0 through P_{15} are parity data for chunks H_0 through H_{111} on every level of intermediate chunks.

This pattern repeats itself all the way down the tree. Thus, hashes H_{m+1} through H_{127} point to parity data for chunks pointed to by H_0 through H_m.[10]

3 LEVELS OF SECURITY AND THE NUMBER OF PARITIES

Non-local redundancy is presented here as a scheme of encoding that allows strategies of retrieval in order to guarantee data availability. With packed address chunks set as the scope of erasure codes, it is crucial that we use the right number of shards and parities among the children of an intermediate node in the Swarm hash tree representing a file. Given assumptions about chunk retrieval error rates and the number of parities used, one can calculate the degree of certainty that the data can be recovered without error. One can even apply the same logic backwards: given some level of certainty with which we want recovery to be error-free, we can compute how many parities should be used to provide that level of safety. In what follows, we give a formal exposition of how to find these parity counts.

Let there be m original chunks and k parity chunks, such that any m chunks out of the total $n = m + k$ ones are fully recoverable after the loss of any k of them. In the process of retrieving the n chunks, what is the likelihood of overall data corruption, given a per-chunk probability of error ϵ?

By "overall data corruption", we mean that more than k chunks are damaged in the data retrieval process. We assume that each chunk's probability of error is independent of other chunks. In that case, the problem boils down to the independent drawing of n chunks, each of which undergo a *Bernoulli trial* of being faulty with probability ϵ. The total number of faulty chunks out of n independent Bernoulli trials is given by the *binomial distribution*:

$$B(i, n, \epsilon) = \binom{n}{i} \epsilon^k (1 - \epsilon)^{n-i}. \tag{1}$$

[10] Since parity chunks P_i do not have children, the tree structure does not have uniform depth.

This expression is the probability mass function for the binomial distribution, yielding the probability that out of n chunks, exactly i will be faulty—assuming that the per-chunk probability of error is ϵ.

Since there are k parities out of the n chunks, the system can tolerate up to k chunk errors. The probability that no more than k errors accumulate can be expressed by summing Equation 1 over i up to k:

$$P(k, n, \epsilon) = \sum_{i=0}^{k} \binom{n}{i} \epsilon^k (1-\epsilon)^{n-i}, \tag{2}$$

which is the cumulative distribution function of the binomial distribution.

One typical question is the following: given the number of chunks n and a value α such that we want the overall probability of data corruption to be below this value, how many out of the n chunks should be parities? Since $P(k, n, \epsilon)$ is the probability that *no more than* k errors accumulate, $1 - P(k, n, \epsilon)$ is the probability of more than k errors; i.e., that *at least* $k + 1$ errors accumulated and therefore the data are corrupted. We want to keep this probability below α, so we can write

$$\alpha \geq 1 - P(k, n, \epsilon). \tag{3}$$

Rearranging, we have

$$1 - \alpha \leq P(k, n, \epsilon). \tag{4}$$

That is, we are looking for values of k which will satisfy this inequality (Figure 3). This can be obtained by inverting the cumulative distribution function in k, resulting in the quantile function $Q(1 - \alpha, n, \epsilon)$. While this inverse has no convenient closed-form expression, it can be efficiently evaluated numerically for any set of input parameters. As with any cumulative distribution function, $P(k, n, \epsilon)$ is monotonically increasing in k. Applying the inverse on both sides of Equation 4 therefore does not flip the direction of the inequality, and gives $k \geq Q(1 - \alpha, n, \epsilon)$. Or if we look for the smallest k satisfying this condition:

$$k = Q(1 - \alpha, n, \epsilon). \tag{5}$$

This is the formula yielding the minimum number of required parities to keep the overall probability of data corruption below α.

Figure 4 presents the number of parities needed as a function of error rate for various levels of security. Figure 5 presents the number of parities needed to keep the probability of overall data corruption at a given level for various values of the per-chunk error rate.

The same type of problem can also be phrased slightly differently: given a number of chunks n, how many parities k should be added to them to keep the overall data corruption probability below some level α? In this case, the total number of chunks is $n + k$ (instead of having n chunks, out of which k are parities), and so Equation 5 is modified to be

$$k = Q(1 - \alpha, n + k, \epsilon). \tag{6}$$

While this equation has no closed-form solution for k, one can easily find the k satisfying it as long as k is bounded in a relatively small range. In our case, the maximum number of chunks, $n + k$, is 128, and so k is at most $128 - n$. This makes it simple to find the value of k compatible with Equation 6. The number of parities in Tables 1-3 were obtained using this method.

In principle, the exact parity counts can be made user-configurable. However, to make non-local redundancy a transparent and easy-to-use feature, we opted for a simplified yet intuitive interface. First of all, we set our maximum tolerated error rate of integrity at 10^{-6}, in other words our security constant expressing our certainty at 6 nines,

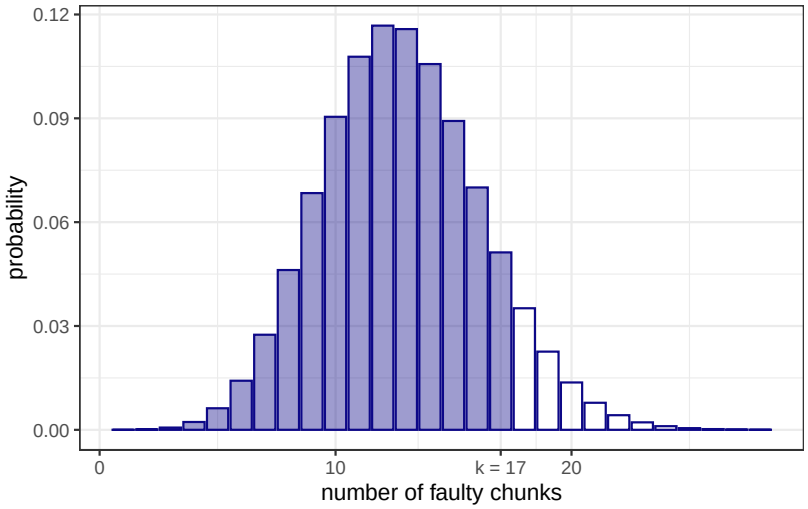

Fig. 3. The point at $k = 17$ along the binomial distribution, where the probability of exceeding this many errors becomes less than $\alpha = 10\%$. Here, the total number of chunks is $n = 128$, and the per-chunk error rate is $\epsilon = 0.1$.

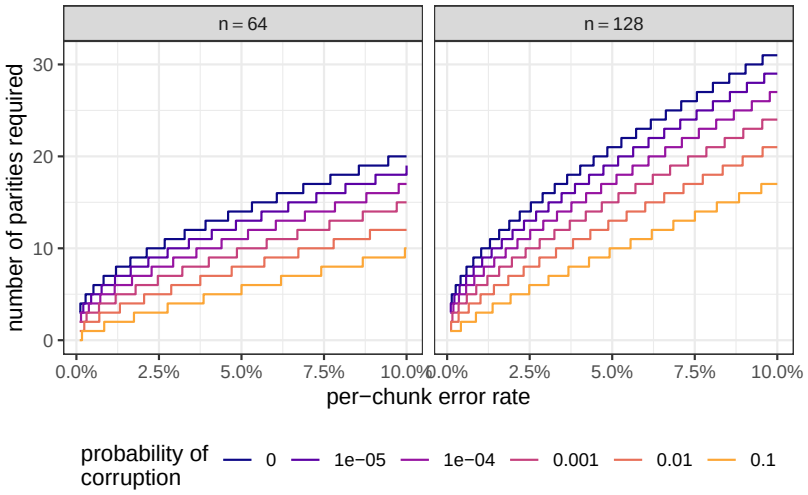

Fig. 4. The number of parities needed (ordinate) as a function of the per-chunk error rate ϵ (abscissa), for keeping the probability of overall data corruption below given limits (colours) and for $n = 64$ chunks (left panel) and $n = 128$ chunks (right panel).

99.9999%. Second, we propose to use a handful of named security levels of (non-local) redundancy which correspond to assumptions about the maximum error rates of individual chunk retrievals expressed as discrete percentages. Table 1 lists the security levels with the corresponding assumption about the maximum error rate of chunk retrieval.

If the number of file chunks is not a multiple of m, it is not possible to proceed with the last batch in the same way as the others. We propose that we encode the remaining chunks with an erasure code that guarantees at least the same

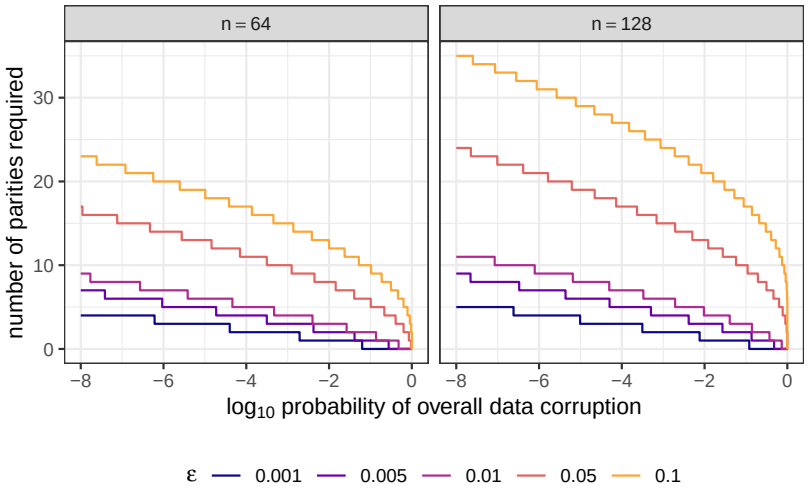

Fig. 5. The number of parities required (ordinate) to keep the probability of overall data corruption at a given level (abscissa), for various values of the per-chunk error rate ϵ (colours) and for $n = 64$ chunks (left panel) and $n = 128$ chunks (right panel).

Table 1. Security levels for non-local redundancy UI and corresponding assumptions about uniform and independent error rates of individual chunk retrieval. In subsequent columns we specify the composition of full chunks for the security levels for unencrypted (columns 4 and 5) and encrypted (columns 6 and 7) content.

security		error rate	unencrypted		encrypted	
level	name	of chunk retrieval	chunks	parities	chunks	parities
0	NONE	0%	128	0	64	0
1	MEDIUM	1%	119	9	59	9
2	STRONG	5%	107	21	53	21
3	INSANE	10%	97	31	48	31
4	PARANOID	50%	38	90	19	90

level of security as the others.[11] Overcompensating, we still require the same number of parity chunks even when there are fewer than m data chunks. However, we can also just calculate the necessary parities for all possible incomplete chunks and security levels. Figure 6 plots the number of parities against the number of chunks required:

Tables 2 and 3 show the number of chunks that are maintainable for a given number of parities k across various security levels. Since encrypted chunks are referenced with the hash address followed by the decryption key, an encrypted reference takes up 2 hash-sized segments. Parity chunks added to an encrypted PAC, however, are calculated based on the encrypted shards and are themselves not encrypted, hence their references only use a single hash. Thus, the number of effective hash-sized segments used is obtained as twice the number of chunks plus the number of parities. Since this can be an odd number and less than 128, in some security levels even the full chunks are not completely full.

As a final note, one should keep in mind that the probability of a failed data retrieval, $\alpha = 10^{-6}$, is not the same as the probability of a failed file retrieval. This is because α is only valid for one 128-chunk segment (64-chunk segment for encrypted content) of a file, not a file as a whole in general. Assuming that retrieval errors may occur independently to any chunk, we can use α and the size of a file to calculate the probability that a file as a whole is successfully retrieved.

[11] Note that this is not as simple as choosing the same redundancy. For example, a 50-out-of-100 encoding is much more secure against loss than a 1-out-of-2 encoding, even though the redundancy is 100% in both cases.

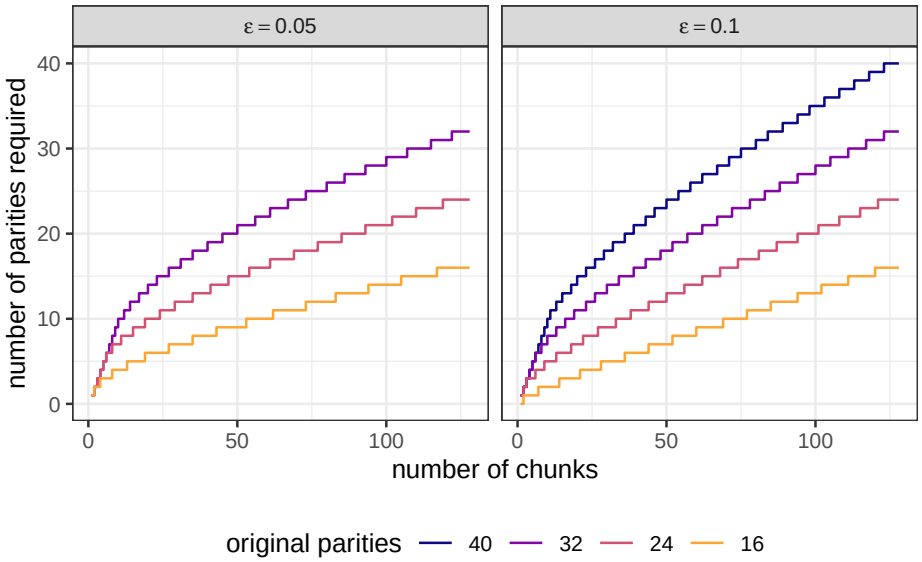

Fig. 6. Number of chunks (abscissa) and the corresponding required number of parities (ordinate) such that will maintain the same overall probability of no data corruption as would be the case with 128 chunks, an original number of parities indicated by the colours, and a likelihood ϵ of an erroneous retrieval of a single chunk indicated in the panel headers.

This probability is $1-\alpha$ for each 128-chunk segment of a file, so if a file consists of s 128-chunk segments, then the probability is $(1-\alpha)^s$. In terms of bytes: a file of g bytes consists of $g/2^{12}$ chunks (because 2^{12} bytes is 4KB), which then make up for $s = g/(2^{12} \cdot 2^7)$ 128-chunk segments (because $128 = 2^7$). This means that the probability P_F of a successful file retrieval is

$$P_F = (1-\alpha)^{g/2^{19}}, \tag{7}$$

an exponentially decreasing function of the file size g. For example, a file of 1GB ($s = 2^{30}$ bytes) with $\alpha = 10^{-6}$ has $P_F = 0.998$, for a failure probability of $1 - P_F = 0.2\%$.

4 DISPERSED REPLICAS

This leaves us with only one corner case: it is not possible to use our m-out-of-n scheme on a single chunk ($m = 1$) because it would amount to $k + 1$ copies of the same chunk. The problem is that copies of the same chunk all have the same hash and therefore are automatically deduplicated. Whenever a single chunk is left over ($m = 1$) (i.e., the root chunk itself), we would need to replicate the chunk in a way that (1) ideally, the replicas are dispersed in the address space in a balanced way, yet (2) their addresses can be known by retrievers who ideally only know the reference to the original chunk's address.

Our solution uses Swarm's special construct, the *single owner chunk* (SOC; Figure 7). Replicas of the root chunk are created by making the chunk data the payload of a number of SOCs. The addresses of these SOCs must be derivable from the original root hash following a deterministic convention shared by uploaders and downloaders.

Table 2. The number of parities (first column in each table) to be appended to a given number of chunks (second and third column of each table, given as a range) so that the probability of an unsuccessful data retrieval remains below $\alpha = 10^{-6}$. The second column is for unencrypted chunks, while the third one is for encrypted chunks. The tables are for security levels 1-3, to be continued for security level 4 in Table 3.

MEDIUM

parities	chunks	
	unencrypted	encrypted
2	1	-
3	2-5	1-2
4	6-14	3-7
5	15-28	7-14
6	29-46	14-23
7	47-68	23-34
8	69-94	34-47
9	95-119	47-59

STRONG

4	1	-
5	2-3	1
6	4-6	2-3
7	7-10	3-5
8	11-15	5-7
9	16-20	8-10
10	21-26	10-13
11	27-32	13-16
12	33-39	16-19
13	40-46	20-23
14	47-53	23-26
15	54-61	27-30
16	62-69	31-34
17	70-77	35-38
18	78-86	39-43
19	87-95	43-47
20	96-104	48-52
21	105-107	52-53

INSANE

parities	chunks	
	unencrypted	encrypted
5	1	-
6	2	1
7	3	1
8	4-5	2
9	6-8	3-4
10	9-10	4-5
11	11-13	5-6
12	14-16	7-8
13	17-19	8-9
14	20-22	10-11
15	23-26	11-13
16	27-29	13-14
17	30-33	15-16
18	34-37	17-18
19	38-41	19-20
20	42-45	21-22
21	46-50	23-25
22	51-54	25-27
23	55-59	27-29
24	60-63	30-31
25	64-68	32-34
26	69-73	34-36
27	74-77	37-38
28	78-82	39-41
29	83-87	41-43
30	88-92	44-46
31	93-97	46-48

The address of a SOC is the hash of its ID and the Ethereum address of its owner. In order to create valid SOCs, uploaders need to sign the SOC with the owner's identity, therefore the owner of the SOC must be a consensual identity with their private key publicly revealed. [12]

The other component of the address, the SOC ID, must satisfy two criteria: (1) it needs to match the payload hash up to 31 bytes and (2) it must provide the entropy needed to mine the overall chunk into a sufficient number of distinct neighbourhoods. (1) is added as a validation criterion for the special case of replica SOCs, while (2) takes care that we can find replicas uniformly dispersed within the address space. This construct is called *dispersed replica*:

[12]This has the added benefit that third parties can also upload replicas of any chunk.

Table 3. As Table 2, but for the PARANOID security level.

PARANOID

parities	chunks	
	unencrypted	encrypted
19	1	-
23	2	1
26	3	1
29	4	2
31	5	2
34	6	3
36	7	3
38	8	4
40	9	4
43	10	5
45	11	5
47	12	6
48	13	6
50	14	7
52	15	7
54	16	8
56	17	8
58	18	9
59	19	9

PARANOID (continued)

parities	chunks	
	unencrypted	encrypted
61	20	10
63	21	10
65	22	11
66	23	11
68	24	12
70	25	12
71	26	13
73	27	13
75	28	14
76	29	14
78	30	15
80	31	15
81	32	16
83	33	16
84	34	17
86	35	17
87	36	18
89	37	18
90	38	19

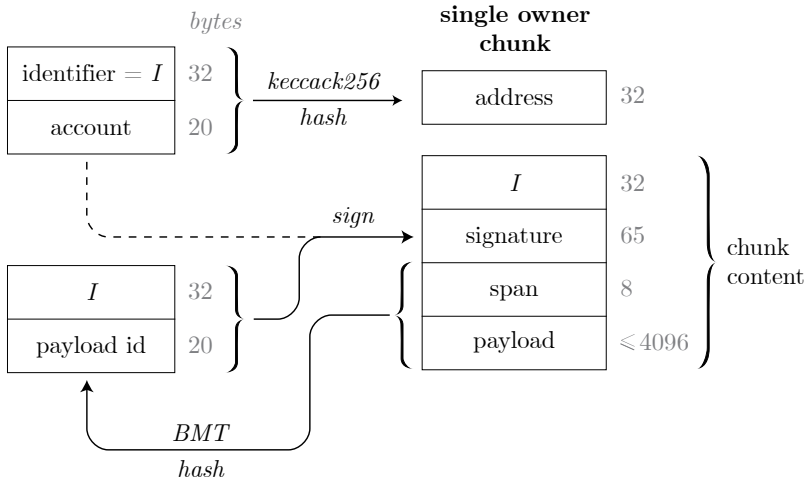

Fig. 7. Single owner chunk (SOC). Unlike content-addressed chunks, SOCs obtain their integrity through the signature of their (single) owner and cross-owner immutability through hashing the owner's address in the chunk address (effectively achieving access control via namespacing).

Let us assume c is the content-addressed chunk we need to replicate; n is the number of bits of entropy available to find the nonces that generate 2^k perfectly balanced replicas; initialise a chunk array ρ of length 2^k and start with n-bit integer $i = 0$ and replica counter $C = 0$.

(1) Create the SOC ID by taking $addr(c)$ and changing the last byte (at index position 31) to i.
(2) Calculate the the SOC address by concatenating ID id and owner o[13] and hash the result using the Keccak256 base hash $a_i := H(id \oplus o)$, and record $c_i = SOC(id, o, c)$.
(3) Calculate the bin this hash belongs to by taking the k-bit prefix as big-endian binary number j between $0 \le j < 2^k$.
(4) If $\rho[j]$ is unassigned, then let $\rho[j] := c_i$ and increment C.
(5) If $C = 2^k$, then quit.
(6) Increment i by one, if $i = 2^n$, then quit.
(7) Repeat from Step 1.

With this solution, we are able to provide an arbitrary level of redundancy for the storage of data of any length. [14]

Then, depending on the strategy, the downloader can choose which address to retrieve the chunk from. The obvious choice is the replica closest to the requesting node's overlay address. In other words, the last item of the sorted chunk array ρ using the comparison function:

$$i < j \Leftrightarrow PO(Overlay(node), \textsc{Address}(\rho[i])) < PO(Overlay(node), \textsc{Address}(\rho[j])) \tag{8}$$

If the probability of any replica being faulty is ϵ, then, assuming independence, the probability that n parities are faulty is ϵ^n. Here we can write $n = k + 1$; that is, we have one "original" chunk and the rest of them are the k parities. Keeping the overall error probability below α then means that

$$\epsilon^{k+1} = \alpha \tag{9}$$

must be satisfied. Taking logarithms on both sides and rearranging, we get

$$k = \frac{\log(\alpha)}{\log(\epsilon)} - 1. \tag{10}$$

This is the number of parities of a singleton chunk required to keep the overall data corruption probability below α. The base of the log in Equation 10 is arbitrary. This means that if we use base-10 logarithms and assume that $\alpha = 10^{-6}$, we get the simpler

$$k = \frac{6}{|\log_{10}(\epsilon)|} - 1. \tag{11}$$

For example, if the per-chunk error rate is ten percent ($\epsilon = 0.1$), then $|\log_{10}(\epsilon)| = |\log_{10}(1/10)| = 1$, and so $k = 6/1 - 1 = 5$ parities are needed. If instead the per-chunk error rate is just one percent ($\epsilon = 0.01$), then only $k = 6/2 - 1 = 2$ parities are necessary.

In particular, for the same per-chunk error rates as in Table 1, we get:

5 PREFETCHING STRATEGIES FOR RETRIEVAL

When downloading, systematic per-level erasure codes allow for different *prefetching strategies*:

[13] The SOC owner of dispersed replicas has the arbitrary private key `0x010...00` and the corresponding ether address is `0xdc5b20847f43d67928f49cd4f85d696b5a7617b5`.

[14] Note that if n is small, then generating all 2^k balanced replicas may not be achievable, and if $n < k$, this is certainly not possible. In general, given n, k at least m miss has a probability of $(1 - m/2^k)^{2^n}$.

Table 4. For a given per-chunk error rate (first column), how many parities (second column) are required of a single chunk to keep the overall data corruption probability below $\alpha = 10^{-6}$?

security level	error rate	parities required	dispersed replicas
NONE	0%	0	0
MEDIUM	1%	2	2
STRONG	5%	4	4
INSANE	10%	5	8
PARANOID	50%	19	16

NONE = *direct with no recovery; frugal*

No prefetching takes place, RS parity chunks are ignored if present. Retrieval involves only the original chunks, no recovery.

DATA = *prefetching data but no recovery; cheap*

Prefetching data-only chunks, RS parity chunks are ignored if present, no recovery.

PROX = *distance-based selection; cheap*

For all intermediate chunks, first retrieve m chunks that are expected to be the fastest to download (e.g., the m closest to the node).

RACE = *latency optimised; expensive*

Initiate requests for all chunks within the scope (max $m + k$) and will need to wait only for the first m chunks to be delivered in order to proceed. This is equivalent to saying that the k slowest chunk retrievals can be ignored, therefore this strategy is optimal for latency at the expense of cost.

All in all, strategies using recovery can effectively overcome the occasional unavailability of chunks, be it due to faults such as network contention, connectivity gaps in the Kademlia table, node churn, overpriced neighbourhoods, or even malicious attacks targeting a specific neighbourhood.

Similarly, given a typical model of network latencies for chunk retrieval, erasure codes in RACE mode can guarantee an upper limit on retrieval latencies.[15]

6 CONCLUSION

This work presents a comprehensive strategy for enhancing data availability in decentralized storage by embedding erasure coding directly into the Swarm chunk tree. By employing Reed–Solomon encoding at the level of packed address chunks, the system achieves non-local redundancy without compromising the deterministic structure of content addressing. The construction allows for user-configurable security levels, defined by quantifiable probabilities of successful retrieval, and supports efficient decoding even under partial network failure. For edge cases involving singleton chunks, the introduction of dispersed replicas ensures resilience through address-space diversification. Furthermore, a spectrum of retrieval strategies is proposed to balance cost, latency, and robustness depending on application needs. Together, these mechanisms form a robust foundation for scalable, fault-tolerant storage in adversarial or unreliable environments.

[15] For instance, in the temporally sensitive case of real-time video streaming, for any quality setting (bitrate and FPS), buffering times can be guaranteed if the batch of chunks representing a time unit of media is encoded using its own scope(s) of erasure coding.

ACKNOWLEDGMENTS

We thank Andrea Robert for her comments and thorough editing work which have greatly improved the paper.

REFERENCES

[1] SB Balaji, M Nikhil Krishnan, Myna Vajha, Vinayak Ramkumar, Birenjith Sasidharan, and P Vijay Kumar. 2018. Erasure coding for distributed storage: An overview. *Science China Information Sciences* 61 (2018), 1–45.

[2] Johannes Bloemer, Malik Kalfane, Richard Karp, Marek Karpinski, Michael Luby, and David Zuckerman. 1995. *An xor-based erasure-resilient coding scheme.* Technical Report. International Computer Science Institute. Technical Report TR-95-048.

[3] Vero Estrada-Galinanes, Ethan Miller, Pascal Felber, and Jehan-François Pâris. 2018. Alpha entanglement codes: practical erasure codes to archive data in unreliable environments. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 183–194.

[4] Vero Estrada-Galinanes, Racin Nygaard, Viktor Tron, Rodrigo Saramago, Leander Jehl, and Hein Meling. 2019. Building a disaster-resilient storage layer for next generation networks: The role of redundancy. *IEICE Technical Report; IEICE Tech. Rep.* 119, 221 (2019), 53–58.

[5] Jun Li and Baochun Li. 2013. Erasure coding for cloud storage systems: A survey. *Tsinghua Science and Technology* 18, 3 (2013), 259–272.

[6] Ralph C Merkle. 1980. Protocols for public key cryptosystems. In *Proc. 1980 Symposium on Security and Privacy, IEEE Computer Society*. IEEE, 122.

[7] James S Plank, Jianqiang Luo, Catherine D Schuman, Lihao Xu, Zooko Wilcox-O'Hearn, et al. 2009. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage.. In *FAST*, Vol. 9. 253–265.

[8] James S Plank and Lihao Xu. 2006. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *Network Computing and Applications, 2006. NCA 2006. Fifth IEEE International Symposium on*. IEEE, 173–180.

[9] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *J. Soc. Indust. Appl. Math.* 8, 2 (1960), 300–304.

[10] Hakim Weatherspoon and John D Kubiatowicz. 2002. Erasure coding vs. replication: A quantitative comparison. In *Peer-to-Peer Systems: First InternationalWorkshop, IPTPS 2002 Cambridge, MA, USA, March 7–8, 2002 Revised Papers 1*. Springer, 328–337.