Treeplication: An Erasure Code for Distributed Full Recovery under the Random Multiset Channel

Michael Gandelman and Yuval Cassuto, *Senior Member, IEEE*

***Abstract*—This paper presents a new erasure code called Treeplication designed for distributed recovery of the full information word, while most prior work in coding for distributed storage only supports distributed repair of individual symbols. A Treeplication code for k information symbols is defined on a binary tree with $2k-1$ vertices, along with a distribution for selecting code symbols from the tree layers. We analyze and optimize the code under a random-multiset model, which captures the system property that the nodes available for recovery are drawn randomly from the nodes storing the code symbols. Treeplication codes are shown to have full-recovery communication-cost comparable to replication, while offering much better recoverability.**

I. Introduction

To design a distributed storage system, one has to balance the costs of storage and communications to reach a certain degree of data availability. Traditional maximum distance separable (MDS) codes, such as Reed Solomon codes [18] and array codes [1], minimize the storage cost regardless of the communication costs if symbols are distributed across nodes in a network. To economize communication, a new field of coding theory – coding for distributed storage – has emerged, contributing many new codes with many advantages in distributed settings. The principal problem addressed by codes for distributed storage is the *repair* of lost symbols with 1) little communication from nodes storing the other symbols (regenerating codes [3], [5], [17], [23], [24]), or 2) communication from few other nodes (locally repairable codes (LRC) [8], [13], [15], [16], [19], [21], [22]). Going beyond the repair problem, in this paper we consider distributed recovery of the full information word, where each information symbol is recovered by a different available node, with little communication among the available nodes. This scenario, which we call *distributed full recovery* for short, is useful in distributed systems that cannot tolerate the high complexity and latency of centralized decoding when accessing the information symbols in parallel. An important use of such systems is *map-reduce* distributed computing [2], where large data units are processed in parallel by multiple machines, each working on one fragment of the data unit. Both regenerating codes and LRC codes assume *centralized* full recovery, where the former expresses this as the "data collector" function, and in the latter the code minimum distance specifies the centralized full-recovery capabilities. We henceforth use the term *data unit* to replace the term information word, in which every symbol is called a *data fragment*. Each symbol in the codeword representing the data unit is called a *code fragment*.

When using *replication*, one gets distributed full recovery trivially, because every node storing a data fragment can access it locally without any communication. However, replication fails full recovery if even one data fragment is missing from the set of available nodes; hence for adequate full-recovery availability many replicas need to be deployed in many nodes, entailing steep equipment costs. Using an *erasure code* instead of replication will attain the same full-recovery availability with fewer deployed fragments (some of which are parity fragments), while requiring some nodes to communicate in order to recover a data fragment from the parity fragment they store. The objective of this paper is to develop such an erasure code, in which the full-recovery communication costs are comparable to replication, but with a much better full-recovery availability.

Throughout the paper, a data unit is divided to k data fragments, and encoded by an erasure code to m code fragments, each stored in a distinct node. Some of the code fragments are systematic data fragments and others are parity fragments. As discussed above, we define the decoding operation to be distributed full recovery, namely, k nodes out of a set of n *available nodes* each recovers a different one of the k data fragments. The set of n available nodes is drawn randomly from the set of m nodes storing the data unit's code fragments, and we assume that the difference $m-n$ can be large, that is, the available node set has a highly punctured version of the length-m codeword. We defer the formal definition of the random drawing model to Section I-A. Note that replication is a special case of this setup, in which distributed full recovery succeeds if and only if each of the k data fragments is present (at least once) in the n available nodes. That said, replication has especially poor full-recovery probability in randomized settings, due to the *coupon collector problem* [12] requiring to draw many fragments ($\Theta(k \log k)$) to succeed with high probability. One should think about the parameter k as the degree of parallelization (number of nodes) needed to process a data unit, and thus code design is needed for specified k values and not in the limit of large k. Note that a constant k does not limit the scaling of the system, because the number of data units and the number of nodes storing their fragments can grow without bound even when k is fixed.

The erasure code we propose for communication-efficient distributed full recovery is called *Treeplication*, owing to its code fragments being generated from a binary-tree structure. A Treeplication code is defined over a perfect binary tree with k leaf vertices and $2k-1$ vertices in total. Each leaf vertex represents a data fragment, and each non-leaf vertex represents the bitwise exclusive-or (XOR) of its two children.

Michael Gandelman and Yuval Cassuto are with the Viterbi Department of Electrical Engineering, Technion – Israel Institute of Technology, Haifa Israel (emails: michaelgandelman@gmail.com, ycassuto@ee.technion.ac.il).

Part of the results in the paper were presented at the IEEE Information Theory Workshop, November 2018.

The tree structure allows on the one hand localized erasure correction that improves the communication efficiency of the code compared to existing erasure codes, and on the other hand spans large-degree parity symbols that improve decodability compared to replication. Once the code structure is set, the main contributions of this paper are showing how to select fragments from the tree to maximize the code performance under randomized node availability, and providing exact analytic evaluations of the code's decodability and communication-cost performance. After formal definition of the drawing model and of the Treeplication code, our results can be divided into three parts. The first part (Sections III, IV) focuses on the full-recovery decodability of Treeplication, where we provide analytic expressions for the decodability probability and an efficient algorithm to find the optimal fragment selection distribution given the size n of the available set. At the end of this part we show the advantage of Treeplication over replication in terms of full-recoverability, in particular, replication requires 60% more available fragments than Treeplication in order to achieve the same probability of full-recovery decodability. In the second part (Section V), we study the communication cost of Treeplication, defined as the total number of fragments communicated in a full recovery of a decodable fragment subset. The main results in this section are 1) an algorithm that finds the recovery schedule with minimal total communication cost, and 2) an analytic expression for the full distribution of the total communication cost under any fragment selection distribution. In Treeplication coding full recovery has the attractive properties that (like in replication) only k nodes (out of the n available nodes) participate in recovery, and each participating node has to send its fragment to at most one other node. Using the derived communication-cost distribution we show that the average communication cost of the optimal Treeplication codes from Section IV is lower than MDS codes by more than an order of magnitude. Finally, in the third part of the paper (Sections VI, VII) we move to study Treeplication in dynamic settings where the available fragments of data units change in time. To that end, in Section VI we propose a measure we call *tree health* that provides a tractable way to evaluate and compare the robustness of Treeplication-coded data units to future events of fragment loss. Then in Section VII we discuss methods to *augment* a Treeplication data unit with additional fragments, while having access to only a small subset of the nodes storing the data unit.

At a high level, the Treeplication scheme goes the opposite direction to most recent works on distributed-storage erasure coding: instead of taking an erasure code and making it more "access friendly", we take the replication scheme and gracefully make it more "storage-cost friendly". That said, it is possible that known regenerating and/or LRC codes (or improvements thereof) can be used toward efficient distributed full recovery. In addition to typically requiring large fragment sizes, the challenges in using current regenerating codes are that high repair efficiencies require more nodes to perform recovery, and that multiple simultaneous repairs (as needed for full recovery) require nodes to send their fragments to many other nodes, demanding high upload bandwidths. The main challenge of using LRC codes (and their relative availability codes [14]) is to obtain multiple simultaneous repair sets in heavily punctured recovery instances.

The structure of Treeplication is inspired by the similar structure of the fountain code proposed in [4], but, among several key differences, our codes are designed for optimal performance in fixed values of k, while the results in the prior work are asymptotic.

A. The random-multiset model

In classical distributed-storage erasure coding, one generates a codeword of m code fragments, out of which n fragments are available for decoding while the other $m-n$ are unavailable (viewed as erasures). It has been universally assumed that the m code fragments are *all distinct*, and thus the n available fragments form *a subset* of the code fragments. In this paper we lift this assumption, and generate m code fragments from a set of M distinct code fragments (with possible repetition). This makes the n available fragments *a multiset* of the M distinct code fragments. By doing so, we can fix the M distinct code fragments to have some desired structure, which in this paper contributes to efficient distributed full recovery of data units. An extreme case of this approach is standard replication, where $M = k$, and the distinct code fragments are simply the k data fragments. The M distinct code fragments of replication enjoy the very convenient structure that decoding is a trivial no-operation, but at the cost of many non-decodable multisets even for fairly large n (the coupon collector's problem). In the Treeplication coding scheme we have $M = 2k-1$ distinct code fragments in a tree structure that offers decoding efficiency, but with much better decodability performance than in replication. An important part of the Treeplication code design is the *fragment selection distribution*, specifying how the m code fragments are drawn from the M distinct code fragments. In addition, for probabilistic analysis of multiset decodability we need to define the *erasure channel* selecting a multiset of n available fragments from the multiset of the m generated ones. To obtain a simpler and cleaner analysis, we merge the fragment selection distribution and the erasure channel into one *random-multiset model*, specifying how the available n-multiset is drawn from the M distinct code fragments, without need to deal with an extra parameter m. In the paper, we use two main random-multiset models: in the uniform model (used in Section III) we draw with replacement n fragments from the $M = 2k-1$ tree vertices of Treeplication; in the non-uniform model (used in Section IV) we draw with replacement n_i fragments from the vertices of layer i of the Treeplication tree.

II. TREEPLICATION CODING

A distributed storage system stores *data units* across storage nodes. Each data unit is broken to k *data fragments*, each of size D. Throughout the paper we assume that $k = 2^s$, for some integer s. To encode the data unit in the storage system, we use a *binary tree* of depth s. The tree has 2^s leaf vertices representing the data fragments, and a total of $2^{s+1}-1$ vertices. Note that including the root there are $d \triangleq s+1$ *layers* in the tree. The layers are numbered from bottom to root, thus

for $i \in \{1, \ldots, d\}$, layer i has 2^{d-i} vertices. In the sequel, we use T_d to denote this tree, and T_ℓ refers to one of T_d's subtrees with $2^\ell - 1$ vertices in layers $\{1, \ldots, \ell\}$; both T_d and its T_ℓ subtrees are complete binary trees. The interpretation of the tree is that each vertex represents a *code fragment*: starting from level $i = 2$ each code fragment is the bit-wise exclusive-or (XOR) of its two children, and the leaves at level $i = 1$ are the "pure" data fragments also called *systematic code fragments*. An example of this tree representation is given in Fig. 1 for the case $k = 8$.

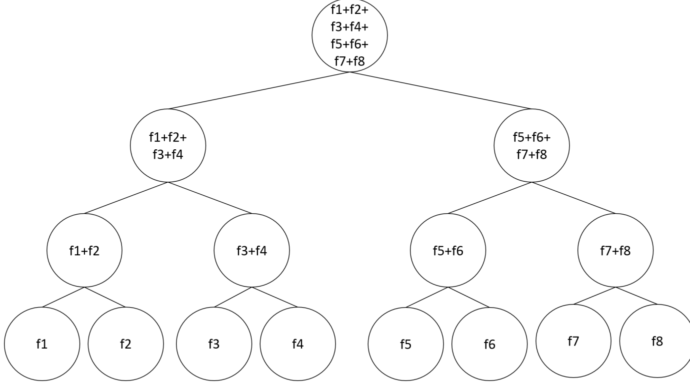


Fig. 1: Tree representation of Treeplication coding for $k = 8$ data fragments ($s = 3$,$d = 4$).

The following is a central definition for the analysis and design of Treeplication codes.

Definition 1. *A* ***Treeplication subset*** *for a tree T_ℓ is defined as a subset of the vertices of T_ℓ.*

A Treeplication subset represents the code fragments available in system nodes for useful operations such as data-fragment recovery. In the sequel, we refer to a vertex and its associated code fragment interchangeably. Moreover, when there is no risk of confusion, we refer to a Treeplication subset simply as *a subset*.

Definition 2. *A Treeplication subset of T_ℓ is said to be* ***decodable*** *if the corresponding code fragments are sufficient to recover the $k' \triangleq 2^{\ell-1}$ data fragments.*

Clearly, a decodable subset must have at least k' vertices, and also all subsets of size greater than $2k' - 3$ are decodable. Between k' and $2k' - 3$, decodability depends on the particular subset available for decoding. (Viewed as an erasure code with block length $2k' - 1$, T_ℓ can correct any single erasure, and many combinations of between 2 and $k' - 1$ erasures, but not more than $k' - 1$ erasures.) An example of a decodable subset of T_4 ($k = 8$) with exactly k vertices is illustrated in Fig. 2. Similarly, Fig. 3 illustrates a decodable subset with more than k vertices. From linearity of the code, a decodable subset with more than k vertices is redundant, and k vertices from the subset are always sufficient to decode the data unit. For instance, in the example presented in Fig. 3 there are $10 > 8$ vertices in the decodable subset, meaning that 2 elements of the subset may be discarded without affecting decodability (e.g. those corresponding to $f3$ and $f7 + f8$, or those corresponding to $f1 + f2 + f3 + f4$ and the root of the tree). The following lemmas further characterize decodable subsets.

Lemma 1. *If a subset of T_ℓ is decodable, then at least one immediate subtree: $T_{\ell-1}$ (left) or $T'_{\ell-1}$ (right) is decodable with only vertices from its subtree.*

Proof: Since the subtrees $T_{\ell-1}$,$T'_{\ell-1}$ are disjoint in their XOR arguments, having both non-decodable internally would mean that two additional code fragments are needed. This is a contradiction because there is only the root as a potential extra code fragment. ■

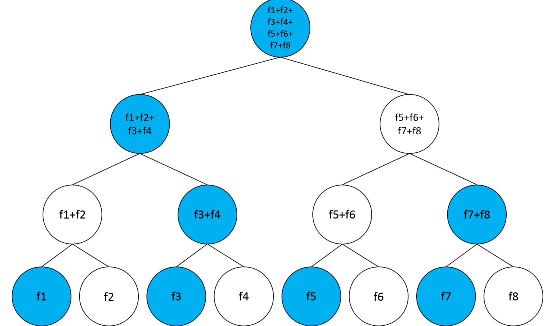


Fig. 2: Example of a decodable Treeplication subset with exactly $k = 8$ vertices (the filled vertices).

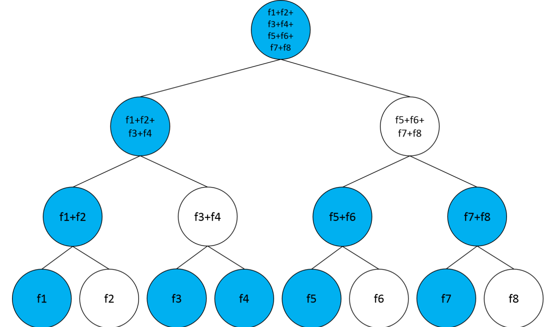


Fig. 3: Example of a decodable Treeplication subset with more than $k = 8$ vertices. Out of the 10 vertices in the subset, $k = 8$ are sufficient to decode the full data unit.

III. Decodability with Uniform Selection

Treeplication is intended for use in a fully distributed storage system where nodes decide in a decentralized way which fragments to store. In the simplest model for decentralized fragment selection, n code fragments are each chosen uniformly and independently (with replacement, so repetitions are possible) from the $2k - 1$ tree vertices. To accommodate multiplicities in the drawing of vertices, we next define Treeplication *multisets*.

Definition 3. *A* ***Treeplication multiset*** *for T_ℓ is defined as a multiset of vertices from T_ℓ.*

Every Treeplication multiset can be mapped to a Treeplication subset by removing vertex multiplicities, and for the sake

of decodability, it is sufficient to look at this subset. We now derive the probability of obtaining a decodable subset once the above-mentioned selection is performed. We first quantify the number of decodable subsets given $k+j$ *unique* vertices in the multiset, for $j=0,\ldots,k-1$; subsequently, we count n-multisets mapping to decodable $k+j$ unique vertices.

For a tree with d layers, define $D_{d,j}$ to be the number of decodable subsets with $2^{d-1}+j$ (unique) vertices. Note that $D_{d,j}=0$ if $j<0$ or $j>2^{d-1}-1$. We partition the decodable subsets to $D_{d,j}=t_{d,j}+r_{d,j}$, where $t_{d,j}$ is the number of subsets that include the root vertex *and* are non-decodable without it, while $r_{d,j}$ is the number of all other decodable subsets.

Lemma 2. *For a tree with d layers and a Treeplication subset with $2^{d-1}+j$ (unique) vertices, the following recursive relation applies to $r_{d,j}$*

$$r_{d,j}=\sum_{l=0}^{j}D_{d-1,l}D_{d-1,j-l}+\sum_{l=0}^{j}D_{d-1,l}D_{d-1,j-l-1}. \quad (1)$$

Proof: By the definition of $r_{d,j}$, the counted decodable subsets either do not have the root vertex or they are decodable without it. In either case the immediate subtrees of the root must both be decodable themselves. The first and second terms in (1), respectively, quantify these decodable subsets without and with the root vertex in them. ■

The more interesting is the term $t_{d,j}$, which we quantify next.

Lemma 3. *For a tree with d layers and a Treeplication subset with $2^{d-1}+j$ unique vertices, the following recursive relation applies to $t_{d,j}$*

$$t_{d,j}=2\sum_{l=0}^{j}D_{d-1,l}t_{d-1,j-l},\ d>1,j\geq 0;\ t_{1,0}=1. \quad (2)$$

Proof: When $d=1$ the tree is just the root vertex, and the root forms a decodable subset that is non-decodable without it (trivially); hence $t_{1,0}=1$. By Lemma 1, one immediate subtree of T_d must be decodable internally, and by definition of $t_{d,j}$ the other subtree must *not* be decodable internally. The former gives the term $D_{d-1,l}$ in (2) and the latter gives $t_{d-1,j-l}$. To understand why the latter is correct, observe that every decodable subset of T_{d-1} that contains its root can be mapped to a decodable subset of T_d by replacing the root of T_{d-1} by the root of T_d, assuming that the other subtree T'_{d-1} is decodable internally. Also, with this root swap it is clear that T_{d-1} is non-decodable without its root if and only if T_d is non-decodable without its root. Finally, the factor 2 in (2) quantifies the two options to choose the internally-decodable subtree among the left and right subtrees. ■

Once we have an efficient way to quantify decodable subsets, it is simple to derive the probability to get a decodable subset under independent uniform selection of each of the n code fragments.

Theorem 4. *For a tree with d layers and a n-multiset, each of whose elements is chosen uniformly and independently from the 2^d-1 vertices of T_d, the probability to get a decodable Treeplication subset is*

$$P_d=\sum_{j=0}^{n-2^{d-1}}\frac{D_{d,j}S(n,2^{d-1}+j)(2^{d-1}+j)!}{(2^d-1)^n}, \quad (3)$$

where $S(a,b)$ is the number of ways to partition a set of a elements into b non-empty subsets[1]*, known as the Stirling number of the second kind [9].*

Proof: Each decodable subset (with $2^{d-1}+j$ unique vertices) from those quantified by a $D_{d,j}$ can be obtained in $S(n,2^{d-1}+j)(2^{d-1}+j)!$ different ways by the uniform selection with replacement. The probability is obtained by normalizing by the total number of choices, decodable or not. ■

We compare the Treeplication scheme under uniform independent selection to standard (uncoded) replication with the same selection policy. For the same input block size k, in replication each choice is one of k data fragments, while in Treeplication it is one of $2k-1$ code fragments. The comparison results can be seen in the two middle columns of Table I below. The results show the advantage of Treeplication: to get to the same decoding-success[2] probability of 0.9, replication needs between 15%-30% higher n than Treeplication, which means a higher storage cost for the same availability performance.

TABLE I: Replication vs. Treeplication (uniform and non-uniform): minimum number of stored fragments n required for decoding probability of 0.9.

k	Replication	Treeplication (uniform)	Treeplication (non-uniform)
2	5	4	3
4	13	10	8
8	33	26	20
16	79	66	49
32	181	157	113

IV. Non-uniform Selection

The uniform selection assumption considered above may not render the optimal tree vertex selection, hence better results may be obtained. With this in mind, in order to improve decodability we now extend the analysis to non-uniform selection. In the non-uniform setup, we have $n=\sum_{i=1}^{d}n_i$, where n_i code fragments are chosen (with replacement) from layer i of the tree. Within each layer the selection is as before: each of the n_i code fragments is chosen uniformly and independently from the 2^{d-i} vertices of layer i. In our analysis, we map each n_i to $p_i=1-(1-\frac{1}{2^{d-i}})^{n_i}$, where p_i is the probability that a certain vertex in layer i is selected to the multiset at least once (same for all vertices in the layer). Note that p_i is monotonically increasing with n_i, and $p_i=0$ when $n_i=0$. It will be simpler for us to assume that a vertex in layer i is included in the

[1] In (3), the elements are vertices chosen with replacement (with possible repetition), and the subsets are the unique vertices that form the decodable subset.

[2] In replication, decoding success is when every data fragment is selected at least once.

multiset (at least once) with probability p_i, independently of the other vertices in the layer, although this assumption is not consistent with the specified discrete parameters $\{n_i\}_{i=1}^d$. This assumption is a reasonable approximation when n_i is of the same order as 2^{d-i}, as required to get decodability with high probability[3]. The following theorem gives an expression for the decoding probability with non-uniform selection.

Theorem 5. *For a tree T_d whose vertices are chosen with probability p_i in layer i, the probability to get a decodable Treeplication subset is*

$$Q_d = Q_{d-1}^2 + 2^{d-1} p_d \prod_{i=1}^{d-1} [(1-p_i)Q_i],\ d>1;\ Q_1 = p_1. \quad (4)$$

Proof: When $d=1$ the tree is just the root vertex, and the subset is decodable if it contains the root vertex, happening with probability p_1. For $i = 1, \ldots, d-1$, denote by B_i the probability that the subset elements in T_i can decode the leaves of T_i if and only if its root vertex is provided to the subset externally. Then $Q_d = Q_{d-1}^2 + 2Q_{d-1}p_d B_{d-1}$, because, similar to Lemmas 2,3, the subset is decodable if both its subtrees are decodable, or if one subtree is decodable, the root is present, and the other subtree is decodable if and only if its root is provided externally. The "only if" is required to not count in the second term probabilities already included in the term Q_{d-1}^2; the "if" part guarantees that the other subtree is decodable when the parent root is present and the other subtree is decodable. B_{d-1} can be calculated with the recursive expression $B_i = 2(1-p_i)Q_{i-1}B_{i-1}$, and the initial value $B_1 = 1 - p_1$. Expanding this expression to B_{d-1} and substituting in the previous equation gives (4). ■

By calculating efficiently Q_d for every *selection distribution* $\{n_i\}_{i=1}^d$, Theorem 5 is a useful tool to design non-uniform Treeplication allocations that, for any given n, maximize Q_d among all $\{n_i\}_{i=1}^d : \sum_{i=1}^d n_i = n$. To find the optimal Q_d efficiently, we first prove some properties of optimal selection distributions that significantly reduce the search complexity.

Lemma 6. *Every optimal selection distribution satisfies $p_i \le p_{i-1}$, $\forall i \in [2, d]$.*

Proof: Assume that $p_1, \ldots, p_{i-2}$ are set, and by contradiction that $p_i > p_{i-1}$. From (4) we have

$$Q_i = Q_{i-1}^2 + 2^{i-1} p_i \prod_{j=1}^{i-1} [(1-p_j)Q_j], \quad (5)$$

$$Q_{i-1} = Q_{i-2}^2 + 2^{i-2} p_{i-1} \prod_{j=1}^{i-2} [(1-p_j)Q_j]. \quad (6)$$

Denote $a := Q_{i-2}^2$ and $b := 2^{i-2} \prod_{j=1}^{i-2} [(1-p_j)Q_j]$. By substituting (6) and a, b into (5), we get

$$Q_i = (a + bp_{i-1})(a + bp_{i-1} + 2bp_i - 2bp_{i-1}p_i). \quad (7)$$

Since a, b are independent of p_i and p_{i-1}, we can see that exchanging between p_i and p_{i-1} in (7) results in an increase in Q_i because

$$(a + bp_{i-1})(a + bp_{i-1} + 2bp_i - 2bp_{i-1}p_i) < (a + bp_i)(a + bp_i + 2bp_{i-1} - 2bp_{i-1}p_i) \quad (8)$$

for any $0 \le p_{i-1} < p_i \le 1$. This is a contradiction. ■

The monotonicity of p_i in i implies the following lemma.

Lemma 7. *Every optimal selection distribution satisfies $n_{i-1} \ge 2n_i$, $\forall i \in [2, d]$.*

Proof: From Lemma 6 we have $1 - p_i \ge 1 - p_{i-1}$, and from monotonicity of the log function $\log(1-p_i) \ge \log(1-p_{i-1})$. Thus substituting the definition of p_i,p_{i-1} gives that

$$\frac{n_{i-1}}{n_i} \ge \frac{\log\left(1 - \frac{1}{2^{d-i}}\right)}{\log\left(1 - \frac{1}{2^{d-i+1}}\right)} \ge 2. \quad (9)$$

■

With Lemma 7 we can prove the following useful property of optimal selection distributions.

Proposition 8. *Every optimal selection distribution satisfies $n_i \ge \sum_{j=i+1}^d n_j$, $\forall i \in [1, d-1]$.*

Proof: By induction starting from $i = d-1$. True for base case $i = d-1$ because $n_{d-1} \ge 2n_d \ge n_d = \sum_{j=d}^d n_j$, where the first inequality is from Lemma 7. Assume true for i, then showing for $i-1$

$$n_{i-1} \ge 2n_i \ge n_i + \sum_{j=i+1}^d n_j = \sum_{j=i}^d n_j,$$

where the first inequality is from Lemma 7 and the second from the induction hypothesis. ■

Proposition 8 is the basis to Algorithm 1 that finds the optimal selection distribution based on searching the small subset of distributions that satisfy the above optimality conditions. In the algorithm, we denote by $\{n_i\}_{i=1}^\ell$ a selection distribution $n_1, \ldots, n_\ell, 0, \ldots, 0$, where the last $d-\ell$ elements of the distribution are 0. For any selection distribution S, we denote by $Q_d(S)$ the result of (4) with p_i corresponding to the n_i of S. In the algorithm, $Q^\star$ holds the maximum decoding probability among all selection distributions explored so far.

Thanks to the factor $1/2$ in the for loop of Algorithm 1, its running time is significantly reduced compared to trivial search. While trivial search needs to explore all compositions of n into up to d sets, only *non-squashing* partitions[4] [20] of n are explored by Algorithm 1. For example when $d = 6$,$n = 128$, trivial search requires 275584033 steps while Algorithm 1 only 25509. Although the focus of this paper is not asymptotic, it is instructive to examine the counts of non-squashing partitions in comparison with general partitions, in terms of their scaling with n. According to equation (1.3) in [6] (citing results from [10], [11]), the count of non-squashing partitions scales as $\exp\left(\frac{(\ln n)^2}{2\ln 2}\right)$; note that this count does not

[3]For verification, we compared the i.i.d model with uniform distribution to the true uniform results of Section III, and got almost the same results.

[4]A partition of an integer $n = n_1 + n_2 + \cdots + n_\ell$ with parts $n_1 \ge n_2 \ge \cdots \ge n_\ell \ge 1$ is called non-squashing if $n_i \ge n_{i+1} + \cdots + n_\ell$ for every $1 \le i \le \ell - 1$.

Algorithm 1 Find optimal non-uniform selection distribution

```
1: function SEARCH(n, d)
2:     Q* := 0
3:     Distribute(1, n, {})
4:     return Q*
5: end function
6: function DISTRIBUTE(j, B, {n_i}_{i=1}^{j-1})
7:     if B == 0 or j > d then
8:         if Q_d({n_i}_{i=1}^{j-1}) > Q* then
9:             Q* := Q_d({n_i}_{i=1}^{j-1})
10:        end if
11:        return
12:    end if
13:    for b ∈ {0, 1, ..., ⌊B/2⌋} do
14:        n_j := B − b
15:        Distribute(j + 1, b, {n_i}_{i=1}^{j})
16:    end for
17: end function
```

limit the number of non-zero parts to at most d, but because d is logarithmic in n (when n is a constant factor times k) and non-squashing forces exponential shrinkage of part sizes, there are not many partitions counted in this expression that have more than d parts (in any case this expression is an upper bound on the count of interest). In comparison, general partitions to up to d labeled parts count as $\binom{n+d-1}{d-1}$ (elementary counting), and thus scale as n^d, which for n of order 2^d gives scaling of $n^{\ln n/\ln 2} = \exp\left(\frac{(\ln n)^2}{\ln 2}\right)$. The factor 2 in the exponent compared to non-squashing implies that the search complexity in Algorithm 1 is roughly the square root of the complexity of trivial search on general partitions.

It turns out that, optimal Treeplication codes greatly outperform both replication and uniform Treeplication. The right column of Table I shows that compared to optimal Treeplication, the storage cost of replication is higher by 60% or more for $k \in \{8, 16, 32\}$. This is close to quadruple the advantage of uniform Treeplication. An attractive property of the optimal Treeplication distributions is that they have a vast majority of systematic data fragments, which means that the system behaves very similarly to an uncoded replication system, only with a much better full-recovery performance. For example, in the third row of Table I, the optimal distribution for $n = 20$ is $n_1 = 16, n_2 = 2, n_3 = 1, n_4 = 1$, namely, $4/5$ of the nodes have data fragments.

Further results comparing the three schemes are given in Fig. 4 where the decoding probability is plotted as a function of n for $k = 8, 16$.

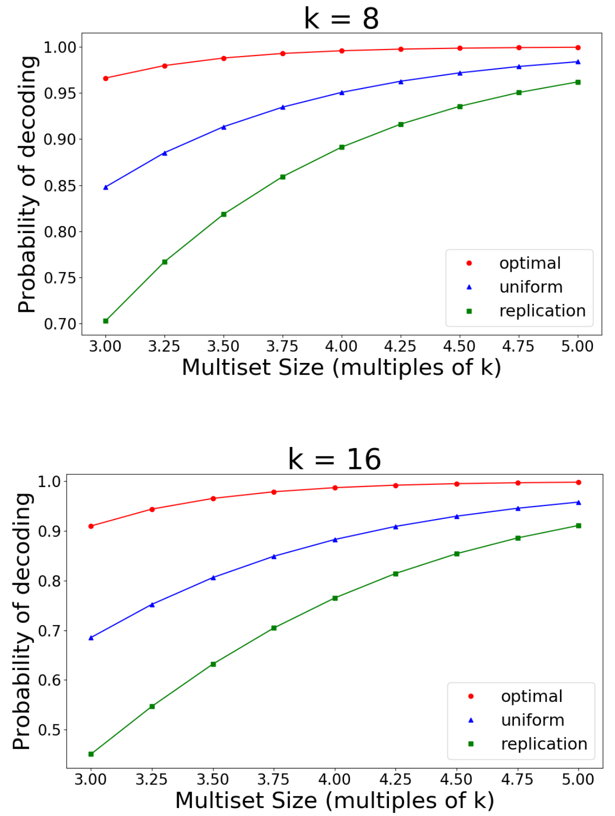


Fig. 4: Probability of obtaining a decodable subset as a function of the multiset size n given in multiples of k, for $k = 8, 16$. Three curves compare: optimal Treeplication, uniform Treeplication, and replication.

V. Fragment Recovery and Communication Cost

In the successful case of having a decodable subset of T_d, the distributed storage system needs to have the k data fragments recovered by the nodes storing the decodable subset. The recovery process is done in a distributed way, where each data fragment is recovered by one node storing a code fragment, using code fragments from other nodes if necessary. For any subset size $n \geq k$, k nodes are chosen to each recover a different data fragment, in a way that communication cost is minimized. Data fragments that appear as leaves (systematic code fragments) in the subset are trivially recovered locally with no communication; the remaining data fragments are recovered by non-leaf vertices that receive code fragments (both systematic and not) of other vertices to recover the assigned data fragment. The cost in terms of communications required for this recovery is the *total* number of code fragments communicated to recover all k data fragments, and it should be minimal.

A. Minimal-communication recovery algorithm

This sub-section presents Algorithm 2, which finds the minimal-communication recovery and counts the number of fragment transmissions. At the start of the algorithm, we have a decodable subset (with k or more fragments) mapped to a tree. Each fragment (tree vertex) is stored by a node in the system, and all nodes know the vertices in the subset. Algorithm 2 is run by each of these nodes, to determine which data fragment (leaf vertex) to recover (if any), and which fragments (tree vertices) to request from the other nodes in the subset. Before presenting the algorithm, we prove properties regarding node selection for minimal-communication recovery. In the sequel, a *present* resp. *missing* vertex is a vertex that is

in resp. *not in* the decodable subset. We define a *missing-vertex path* as a path in the tree, in which all vertices are missing.

Lemma 9. *If a Treeplication subset is decodable, then 1) there is no missing-vertex path between a leaf and the root, and 2) no vertex (present or missing) has missing-vertex paths connecting its two children with two leaves.*

Proof: The existence of a missing-vertex path from leaf to root contradicts decodability because in that case no present code fragment depends on that leaf. Two missing-vertex paths ending at vertices with a common parent vertex x imply that both subtrees directly under x are non-decodable (by condition 1 above), thus violating Lemma 1. ■

Proposition 10. *Suppose present vertex x recovers leaf vertex y if and only if there is a missing-vertex path between a child of x and y. Then in a decodable Treeplication subset, each missing leaf is recovered by a single unique vertex, and this vertex is the lowest one (in terms of its layer index) capable of recovering y.*

Proof: From condition 1 of Lemma 9, each missing leaf must have a path upward ending at a present vertex; from condition 2 there cannot be another leaf that is in missing-vertex path ending at a child of x. These prove that every leaf will be recovered by a unique present vertex. x is the lowest present vertex in the tree that can recover y, because it is at the end of a missing-vertex path from y, making it the lowest vertex whose code fragment has y as argument. ■

Building on Proposition 10, Algorithm 2 now finds the vertices recovering the missing data fragments with minimal communication (the present data fragments are recovered locally with no communication, and are not handled by Algorithm 2). Each of the vertices chosen for recovery is the lowest possible in the tree able to recover the corresponding data fragment, hence requires the least amount of communication. A vertex evaluated to “false” in line 9 is not recovering any data fragment, and can be discarded as redundant (this happens when the decodable subset has more than k vertices). The variable sum holds the aggregate number of code fragments communicated to the nodes recovering the data fragments. The explicit identities of the communicated code fragments can be extracted from the identities of the vertices reached in line 6. If Algorithm 2 terminates without recovering all missing data fragments, from Proposition 10 we know that the subset is not decodable.

Fig. 5 illustrates an example of a run of Algorithm 2. A red label f_j in vertex x shows the algorithm's finding in line 9 that the node storing x is to recover the missing data fragment f_j. A solid red arrow from vertex z to vertex x represents the algorithm's finding in line 6 that the present code fragment z is needed by the node storing x to recover its assigned data fragment. The sum output of Algorithm 2 is the number of red arrows in Fig. 5, which is the total communication cost to be incurred on recovering all data fragments: 5 code fragments (4 systematic and 1 not) are communicated to recover all 8 data fragments.

It is important to note that, our assumption that the input to Algorithm 2 is a decodable subset is only for simplicity

Algorithm 2 Fragment recovery

1: $\mathrm{sum} := 0$
2: **for** each present vertex x **do**
3: $\mathrm{sum}_x := 0$
4: **for** each path downwards from x **do**
5: traverse path until a present vertex or a missing leaf reached
6: **if** a vertex that is present reached **then** $\mathrm{sum}_x = \mathrm{sum}_x + 1$
7: **end if**
8: **end for**
9: **if** missing leaf was reached in a downward path **then** $\mathrm{sum} = \mathrm{sum} + \mathrm{sum}_x$ // x is recovering a leaf
10: **end if**
11: **end for**
12: return sum

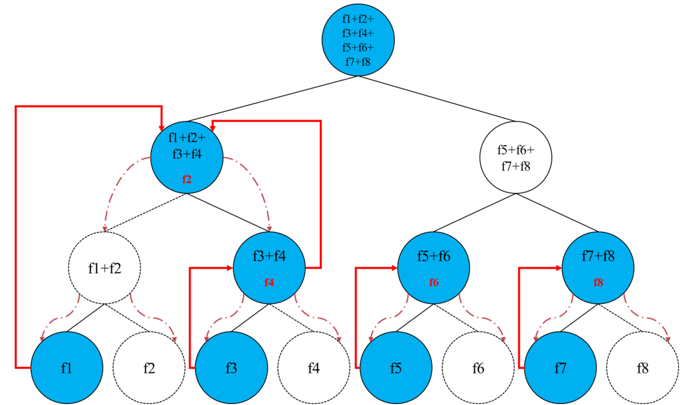


Fig. 5: Run of Algorithm 2 on a decodable Treeplication subset with 9 distinct code fragments.

of presentation. By a small change to the algorithm, we can detect a non-decodable subset (and return $\mathrm{sum} = \infty$) when the condition in line 9 evaluates to “true” fewer times than the number of missing leaves.

The following proposition gives the *worst case* communication cost of Treeplication.

Proposition 11. *Recovering a data unit from a decodable Treeplication subset requires communicating at most $k-1$ code fragments in total.*

Proof: We first prove that, we can assume without loss of generality that the decodable subset has exactly k present vertices. If not, we can remove the x vertices that evaluate to “false” in line 9 of Algorithm 2, and remain with only present leaves and vertices that each recovers a unique missing leaf; these together sum up to k. Now we observe in lines 5,6 of Algorithm 2 that the communication count is incremented only for the *first* present vertex reached in a path downward from x. So looking upward from a present vertex, it needs to be communicated to at most one vertex, which is the first present vertex in its path to the root. This shows at most k communicated fragments. To show upper bound of $k-1$, we observe that in any subset there is a present vertex that has no

present vertices above it in the tree (e.g. the root). This vertex is (or, if plural, these vertices are) not communicated during recovery, thus bounding the total communication at $k-1$ or less. ■

Note that the bound of $k-1$ is worst-case tight, as the subset in Fig. 2 in fact requires $k-1=7$ fragments to be communicated for recovery. The nice thing about the number $k-1$ is that it equals the number of code fragments a node needs for *centralized full recovery* with MDS codes, meaning that Treeplication supports distributed full recovery with no extra cost. Moreover, in the following we see that on average the communication cost is significantly lower than this worst case.

To evaluate the *average* recovery communication cost of Treeplication, we first show in Table II the empirical average number of code fragments communicated per instance of a decodable subset. Treeplication is implemented using the optimal (non-uniform) selection parameters $\{n_i\}_{i=1}^{d}$ found by Algorithm 1, and uniform sampling (with replacement) of n_i vertices in layer i. The minimal-communication recovery per simulation instance is found using Algorithm 2. We compare the communication cost to similar decentralized recovery using systematic MDS codes with block length $2k-1$ (identical to the vertex count of Treeplication) also simulated as a uniform and independent selection (of n fragments from the $2k-1$ code symbols). For both schemes we used the same $n=3k$, which is a common replication factor in pure replication settings such as the default in [7]. We can see that Treeplication is very economical in communication, requiring very small numbers of fragments per instance (in particular much smaller than the worst-case $k-1$). When using MDS codes, recovery of non-systematic fragments requires a heavy load of $k-1$ fragments per recovering node, which results in an order of magnitude or more higher communication cost per instance.

TABLE II: Treeplication vs. MDS: communication cost (n=3k).

k	# fragments Treeplication	# fragments MDS
4	0.35	1.82
8	1.18	10.64
16	2.88	49.62
32	6.552	213.10

Next, we derive analytic expressions for the expected communication cost of Treeplication.

B. Derivation of the expected communication cost

Throughout the forthcoming analysis, we assume decoding subsets are sampled according to the non-uniform selection of Section IV, that is, a tree vertex in layer i is present in the subset with probability p_i, independently from other vertices. We also use the notation Q_i from Section IV to denote the probability that a tree with i layers is decodable under this sampling (recall the efficient calculation of Q_i by Theorem 5). The objective of this analysis is to calculate the expected total communication cost to recover all fragments of a data unit, and the expectation is over all *decodable* subsets (we exclude from the analysis sampling instances that result in non-decodable subsets; in fact, this necessary exclusion makes the analysis more challenging.) The cost we analyze throughout is that of Algorithm 2, which is the minimal for every instance. Following the terminology of Section V-A, each data fragment is recovered by a unique present vertex, and to each of the k recovering vertices, zero or more code fragments are communicated. Thanks to the symmetry of data fragments, it is useful to define the expected cost to recover *a single* data fragment, and obtain the expected *total* cost as k times this number. We get the expectation of the communication cost per data fragment by deriving the full distribution of the communication cost, defined next.

Definition 4. *Let $C_d(N)$ be the probability that in a decodable subset of T_d a particular data fragment is recovered by communicating N code fragments.*

We have $\sum_{N\geq 0} C_d(N)=1$, and from symmetry $C_d(N)$ is the same for any data fragment. Since we are only interested in analyzing the cost of *decodable* subsets (we assume that for non-decodable subsets the algorithm will halt and not invoke any communications), we define $C_d(N)$ as a probability *conditioned on decodability*. The next definition is similar to $C_d(N)$, only defining the *joint* probability.

Definition 5. *Let $P_i(N)$ be the probability that a subset of T_i is decodable, and a particular data fragment is recovered by communicating N code fragments.*

We changed the tree index from d to i in Definition 5, because we will make a recursive use of $P_i(N)$. The following is a similar definition, only specific to recovery by the root.

Definition 6. *Let $A_i(N)$ be the probability that a subset of T_i is decodable, and a particular data fragment is recovered with N communicated code fragments, given that the root is present and that there is a missing-vertex path from one of its children to the leaf of the recovered data fragment.*

In the language of Proposition 10, Definition 6 enumerates the communication cost in cases where the root is chosen to recover the particular data fragment. One final definition is needed to carry out the recursive calculation of the communication-cost distribution.

Definition 7. *Let $F_i(N)$ be the probability that a subset of T_i is decodable, and has N present vertices with no present ancestors.*

Note that in particular $N=1$ in subsets where the root is present. Definition 7 is useful because it will help capture the fragment count sum_x, incremented in line 6 of Algorithm 2 every time a present vertex is reached in the traversal downward from x. We now give a series of lemmas that together facilitate the recursive calculation of $P_i(N)$, and in turn of $C_d(N)$.

Lemma 12. *$P_i(N)$ can be calculated by*

$$P_i(N) = Q_{i-1}P_{i-1}(N) + A_i(N)p_i \prod_{l=1}^{i-1}(1-p_l)$$
$$+ p_i \prod_{l=1}^{i-1}(1-p_l) \sum_{j=1}^{i-1}\left[2^{j-1}P_j(N) \prod_{\ell=1,\ell\neq j}^{i-1} Q_\ell\right], \; i > 1. \quad (10)$$

$$P_1(N) = 0, \; N > 0. \quad (11)$$

$$P_1(0) = p_1. \quad (12)$$

Proof: For (11), (12) in which $i = 1$, the decodability probability is p_1, and no communicated fragments are required ($N = 0$). When $i > 1$, we divide to three mutually exclusive cases, each corresponding to a summand in the right-hand side of (10).

Case 1: Both subtrees of the root are independently decodable. In that case we distinguish between the subtree that contains the leaf of the recovered data fragment (henceforth called the "recovered leaf") and the other subtree. Then the probability decomposes to a product of $P_{i-1}(N)$ for the recovery in the recovered leaf's subtree, times the probability Q_{i-1} that the other subtree is decodable. In the remaining cases one of the root's subtrees is not independently decodable, which implies that the root recovers a leaf vertex, either the recovered leaf (Case 2) or a different one (Case 3).

Case 2: The recovered leaf is recovered by the present tree root. Recall from Proposition 10 that this case implies a missing-vertex path between the recovered leaf and a child of the root. The probability to have a present root and this missing-vertex path is $p_i \prod_{l=1}^{i-1}(1-p_l)$. The remaining term $A_i(N)$ in the second summand is, by definition, the probability that N code fragments are communicated, given that recovery is by the root.

Case 3: The recovered leaf is recovered by a vertex other than the root, while the present root recovers a different leaf, which we call "another leaf". The former condition distinguishes from Case 2, and the latter distinguishes from Case 1 because it implies a root's subtree that is not independently decodable. Given a decodable subset, every missing-vertex path between a child of the root and another leaf defines a partition of the remaining $2^{i-1} - 1 - i$ tree vertices (vertices of T_i not in this path and not the root) to subtrees. Each such partition has one subtree of ℓ layers, for each $\ell \in \{i-1, i-2, \ldots, 1\}$. According to Lemma 1, the root and each of the vertices of the missing-vertex path, except the leaf, must have at least one immediate subtree that is independently decodable. For each of these vertices, the subtree in the direction of the missing-vertex path is not independently decodable by Lemma 9 (part 1). Thus the subtrees in such a partition must all be independently decodable. From the set $\{i-1, i-2, \ldots, 1\}$ we take j to be the number of layers of the special decodable subtree containing the recovered leaf. Given j, there are 2^{j-1} different missing-vertex paths ending in another leaf, each defining a different such partition. The third summand of (10) sums over all possible partitions the probability of recovering the recovered leaf with N fragments, within those partitions. The terms of this summand are: 1) p_i the probability that the root is present, 2) $\prod_{l=1}^{i-1}(1-p_l)$ the probability that the path defining the partition is a missing-vertex path (same probability for all partitions), 3) sum over all partitions, using the size index j, of the probability that the recovered leaf is recovered in its subtree with N fragments ($P_j(N)$), and the other subtrees are decodable $\prod_{\ell=1,\ell\neq j}^{i-1} Q_\ell$.

■

Next, we calculate $A_i(N)$ recursively from $A_{i-1}(\cdot)$ and $F_{i-1}(\cdot)$.

Lemma 13. *$A_i(N)$ can be calculated by*

$$A_i(N) = \sum_{l=1}^{2^{i-2}} \left[F_{i-1}(l)A_{i-1}(N-l)\right], \; i > 2 \quad (13)$$

$$A_2(1) = p_1, \quad (14)$$

$$A_i(N) = 0, \; \text{otherwise}. \quad (15)$$

Proof: When $i = 2$, the recovered leaf is a child of the root, hence both ends of the missing-vertex path are the recovered leaf itself. Conditioned on the existence of this missing-vertex path and the present root, the tree is decodable if and only if the other child of the root is present, which occurs with probability p_1. In this case $N = 1$ fragment is communicated.

For $i > 2$, we split the N communicated fragments to l coming from the root's subtree not including the recovered leaf (which we call the former subtree), and $N - l$ from the subtree of the recovered leaf (which we call the latter subtree). We know that the latter subtree is not independently decodable (from the existence of the missing-vertex path), so from Lemma 1 the former subtree must be independently decodable. Having l fragments communicated from the former subtree which is independently decodable occurs with probability $F_{i-1}(l)$ by definition. These l fragments together with the root fragment can recover the root of the latter subtree; requiring $N - l$ fragments from this subtree occurs with probability $A_{i-1}(N-l)$ because the conditions in the definition of $A_\ell(\cdot)$ are satisfied for the latter subtree when its root is known.

Summing over all possible values of l, we get (13).

■

Finally, the next lemma shows how to calculate $F_i(N)$ recursively from $F_{i-1}(\cdot)$.

Lemma 14. *$F_i(N)$ can be calculated by*

$$F_i(N) = (1-p_i)\sum_{l=1}^{N-1}\left[F_{i-1}(l)F_{i-1}(N-l)\right], \; N > 1 \quad (16)$$

$$F_i(1) = p_i\left[Q_{i-1}^2 + 2^{i-1}\prod_{j=1}^{i-1}[(1-p_j)Q_j]\right]. \quad (17)$$

Proof: When the root is present, only one vertex (the root itself) has no present ancestors. Hence $N = 1$, and the product in (17) gives the probability that the root is present and the tree

is decodable (note that the right-hand side of (17) is similar to (4), but not identical because here it is required that the root is present). When $N > 1$, the root is not present, and the number of present vertices with no present ancestors divides to l in one subtree and $N - l$ in the other. This gives the product $F_{i-1}(l)F_{i-1}(N-l)$ for the probability, and summing over all l and multiplying by the probability $1 - p_i$ that the root is not present, we get (16). ■

With Lemmas 13, 14 and 12, we can efficiently calculate $P_i(N)$ for any i and N, and for any selection distribution $\{p_i\}_{i=1}^d$. Then the expected communication cost for decodable subsets of d-layer Treeplication is taken simply as

$$E \triangleq 2^{d-1} \sum_{N \geq 0} N \cdot C_d(N) = 2^{d-1} \frac{\sum_{N \geq 0} N \cdot P_d(N)}{Q_d}, \quad (18)$$

and the equality follows from the elementary probability-theory relation $P(X|Y) = P(X,Y)/P(Y)$ (where Y is the event that the subset is decodable).

Evaluating E in (18) for $d = \{3,4,5,6\}$ ($k = \{4,8,16,32\}$) using the optimal non-uniform distribution with n, we get the results in Table II. For comparison, in each row we include the empirical results from Section V-A, and observe their good match.

TABLE III: Treeplication's expected communication cost ($n = 3k$).

k	# comm. fragments
4	0.357 (empirical: 0.350)
8	1.143 (empirical: 1.180)
16	2.830 (empirical: 2.880)
32	6.524 (empirical: 6.552)

VI. TREE HEALTH

So far in the paper, a Treeplication multiset was evaluated with respect to its instantaneous properties: whether it is decodable, and how efficiently it can recover the data unit. However, in a real distributed storage system, multisets evolve in time due to changes in node availability. Therefore, in this and the next sections, we extend the scope to consider *forward-looking properties* of the multiset, e.g., its robustness to possible future events of nodes going unavailable. Within that scope, one decodable multiset may be "better" than another decodable multiset because it is more likely to remain decodable after some fixed number of nodes go unavailable. We refer to this quality of Treeplication multisets as their *tree health*, and develop tools to evaluate and compare multisets with respect to a concrete tree-health measure. Comparing multisets' health is useful in storage systems because it allows to allocate resources efficiently to data units according to health, thereby maximizing the overall system resilience. The tree health is parametrized by an integer l, which quantifies the number of nodes becoming unavailable; l captures the degree of robustness expected from the multiset. Throughout the discussion we assume that the nodes becoming unavailable are drawn uniformly from the multiset.

The health of a Treeplication multiset depends on its specific present vertices (and multiplicities), and therefore we seek an efficient method to measure the health without exhaustively enumerating all possible combinations of l node losses. To that end, we next prove decodability conditions that facilitate the formulation of a tractable tree-health measure. First, we define a subset of the Treeplication tree we call a *diagonal*.

Definition 8. *A set of vertices from T_d is called a* ***diagonal*** *if it forms a path downward starting at an arbitrary vertex and ending in a leaf vertex.*

A special case of Definition 8 is a diagonal consisting of a single leaf vertex. A Treeplication tree decomposed to a union of disjoint diagonals is called a *diagonal cover*. Next we prove that every Treeplication tree has a (non unique) diagonal cover.

Proposition 15. *A Treeplication tree T_d can be decomposed as a diagonal cover with k diagonals, where 1 diagonal is of size d, and for each $i \in \{1,\dots,d-1\}$ there are 2^{d-i-1} diagonals of size i.*

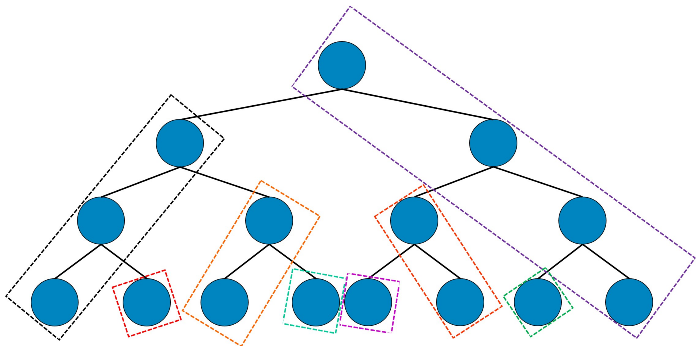


Fig. 6: A decomposition of T_4 as a diagonal cover with 8 diagonals.

The proof of Proposition 15 is immediate, by successively assigning diagonals where each diagonal is a path from the top-most vertex not previously assigned to a diagonal, to any leaf below it. Fig. 6 shows a tree of 4 layers decomposed into a diagonal cover of $k = 8$ diagonals. As stated by Proposition 15, one diagonal in the cover is of size 4, another of size 3, two of size 2, and four of size 1.

Using the decomposition to diagonal covers, we can prove the following (sufficient and necessary) decodability condition for Treeplication subsets.

Theorem 16. *A Treeplication subset for T_d is decodable if and only if it has at least one diagonal cover in which every diagonal has at least one present vertex.*

Proof: For sufficiency, we assume there is such a diagonal cover and need to prove decodability. We prove by induction on d. For $d = 1$, the diagonal cover consists of one leaf, and decodability follows trivially if this leaf is present. For the induction hypothesis, suppose the condition is sufficient for every T_i with $i \in \{1,\dots,d-1\}$. To prove the induction step, we examine the single diagonal of T_d with size d. Each vertex of this diagonal has under it a subtree where the condition is satisfied. Hence all these subtrees are independently decodable

by the induction hypothesis. Now we can show that any present vertex in this diagonal, together with vertices in the independently decodable subtrees, can recover the vertex below it in the diagonal. Applying this iteratively, we can recover the leaf of this diagonal, which is the only leaf not in the independently decodable subtrees.

For necessity, we first construct diagonals by taking each diagonal to include a leaf and all vertices in a path from it upward until the lowest present vertex (if a leaf is present, the diagonal only includes this leaf). If the subset is decodable, then by Lemma 9 (part 2) no two leaves share the same lowest present vertex in their paths upward. This implies that the diagonals are disjoint, and from Lemma 9 (part 1) each diagonal has one present vertex. To complete the diagonal cover, we extend each existing diagonal upward until reaching the top-most vertex not already in a diagonal. This extension results in a diagonal cover, and can only add present vertices to the diagonals, thus the condition is satisfied. ■

Theorem 16 provides a sufficient and necessary decodability condition on the diagonals of T_d's diagonal covers. By that, it reduces the decodability of the entire subset to the simpler condition that individual diagonals in a cover are non-empty, that is, have at least one present vertex each. One challenge still remaining is that the diagonal covers are not disjoint, and the union probability among all covers to meet this condition cannot be simply calculated as a sum of probabilities for the individual covers (in general the sum of individual probabilities will only give an upper bound on the decodability probability). This challenge motivates defining a tree-health measure that picks one diagonal cover, with respect to which the robustness of the Treeplication multiset is evaluated. The special diagonal cover proposed for this health measure is defined next.

Definition 9. *Given a Treeplication multiset for T_d, we call a diagonal cover a **l-principal diagonal cover** if its diagonals have maximum average probability of being non-empty after the loss of l multiset elements.*

The motivation to pick out principal diagonal covers from all possible covers is that the principality condition of Definition 9 makes those covers more likely to fulfill the sufficient condition of Theorem 16. In that sense, we replace the union of covers considered in Theorem 16 by one strong candidate that is the l-principal diagonal. Based on that motivation, we choose the following tree health measure.

Definition 10. *For a Treeplication multiset define the **principal l-health** as the average probability, over the diagonals of a l-principal diagonal cover, that the diagonal is non-empty after the loss of l multiset elements.*

Next we give more definitions that will be useful for finding l-principal diagonal covers and calculating the principal l-healths of multisets.

Definition 11. *Given a Treeplication multiset for T_d we denote by $v_{i,j}$ the number of times the j-th vertex of the i-th layer of T_d appears in the multiset. We call $v_{i,j}$ the **vertex weight** of the i,j vertex.*

For the purpose of Definition 11 we number the tree vertices from left to right in each layer. Additional weight definitions are given next for multisets, diagonals and diagonal covers.

Definition 12. *Given a Treeplication multiset for T_d we define the **multiset weight** as the sum over all vertices of T_d of the vertex weights.*

Note that the multiset weight is simply the number of elements in the multiset, also denoted n in Sections III,IV.

Definition 13. *For a Treeplication multiset and a diagonal of T_d we define the **diagonal weight** as the total number of times the vertices of this diagonal appear in the multiset.*

Definition 14. *For a Treeplication multiset and a diagonal cover of T_d we define the **weight profile** $W \triangleq \{w_j | j \in \{1, \ldots, k\}\}$, where w_j is the weight of the j-th diagonal in the cover.*

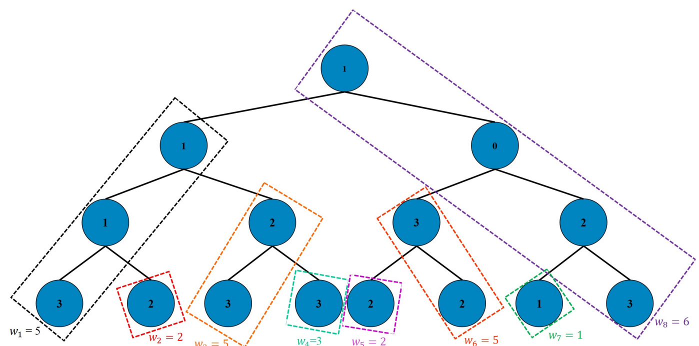


Fig. 7: A diagonal cover of a sample Treeplication multiset ($k = 8$). Each vertex is marked by its vertex weight, and each diagonal is marked by its diagonal weight w_j.

Note that for any diagonal cover, the sum $\sum_{j=1}^{k} w_j$ equals the multiset weight. Fig. 7 shows for the diagonal cover of Fig. 6 the vertex and diagonal weights of a sample multiset.

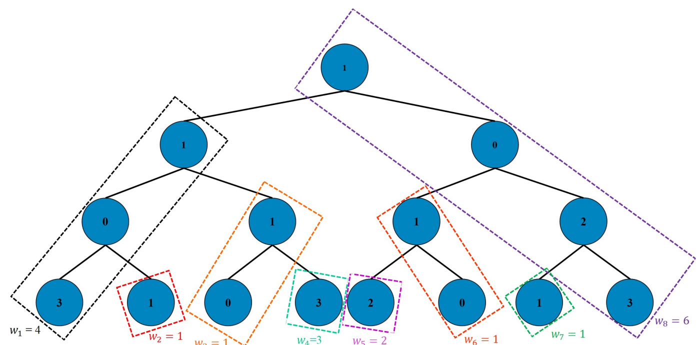


Fig. 8: Diagonal cover and Treeplication multiset from Fig. 7 after loss of $l = 10$ symbols, no weight zero diagonals.

Recall from Definition 10 that the principal l-health of the multiset is defined with respect to loss of l uniformly chosen elements. In the sequel, we assume that the choice of these elements is done by drawing a sequence of l multiset elements (without replacement). Figs. 8 and 9 illustrate two possible outcomes for the diagonal cover in Fig. 7 after the loss of $l =$

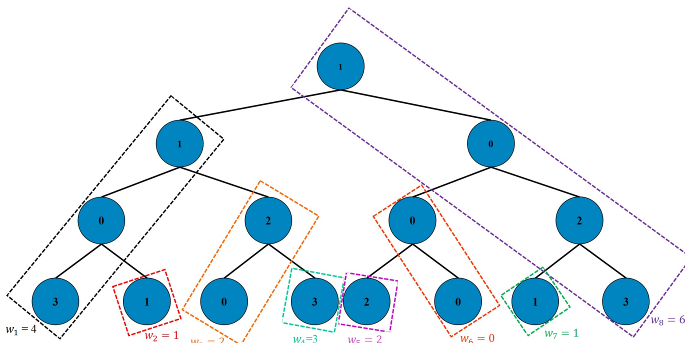


Fig. 9: Diagonal cover and Treeplication multiset from Fig. 7 after loss of $l = 10$ symbols, one weight zero diagonal.

10 elements from the sample multiset. In Fig. 8 no diagonal has weight zero, while in Fig. 9 w_6 dropped to zero.

Toward finding l-principal diagonal covers, we next calculate the average probability, under the uniform drawing model, that a diagonal in the cover remains with non-zero weight.

Proposition 17. *For a Treeplication multiset with weight n and a diagonal cover of weight profile $W = \{w_j | j \in \{1, \dots, k\}\}$, the average probability, over the diagonals of the cover, that the diagonal will have non-zero weight after the loss of l multiset elements, equals*

$$1 - k^{-1} \sum_{j=1}^{k} \frac{\binom{n-w_j}{l-w_j} l!}{n!/(n-l)!}. \tag{19}$$

Proof: The numerator is the number of drawing sequences that leave the j-th diagonal with zero weight. The denominator is the total number of possible drawing sequences. Averaging over the k diagonals and taking the complement gives (19). ∎

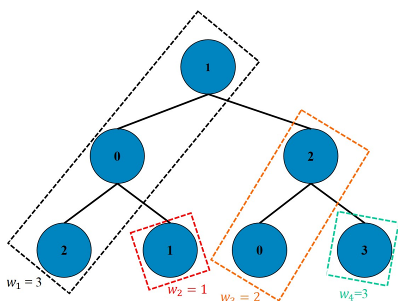


Fig. 10: One diagonal cover for a sample Treeplication multiset.

Figs. 10,11 show two different diagonal covers with respect to the same Treeplication multiset. For $l = 3$, the probability (19) equals 0.890 for Fig. 10 and 0.935 for Fig. 11. We later see that the cover in Fig. 11 is a l-principal diagonal for $l = 3$ (as well as for all other l values), which means that 0.935 is maximal for $l = 3$ among all possible covers.

We propose Algorithm 3 for constructing a diagonal cover for a Treeplication multiset. Algorithm 3 can be explained in words as growing the diagonals upward, where lower-weight

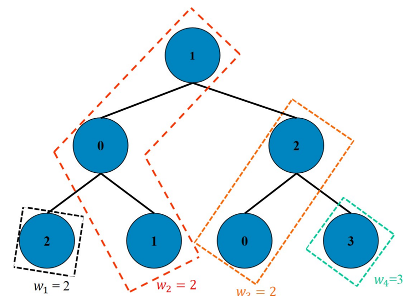


Fig. 11: Another diagonal cover for the Treeplication multiset from Fig. 10 that has a higher average probability in (19).

diagonals are preferred when choosing to which diagonal to add a vertex.

Algorithm 3 Obtain Principal Diagonal Cover

```
1:  for j ∈ [1, k] do
2:      g_j := {(1, j)} // init each diagonal to contain a unique leaf
3:  end for
4:  for i ∈ [2, d] do
5:      for ℓ ∈ [1, 2^(d−i)] do
6:          y = j : (i − 1, 2ℓ − 1) ∈ g_j   // diag. of left child
7:          z = j : (i − 1, 2ℓ) ∈ g_j   // diag. of right child
8:          if w_y < w_z then // add vertex to lighter diagonal
9:              g_y = g_y ∪ {(i, ℓ)}
10:         else
11:             g_z = g_z ∪ {(i, ℓ)}
12:         end if
13:     end for
14: end for
15: G := {g_j | j ∈ [1, k]}
16: return G
```

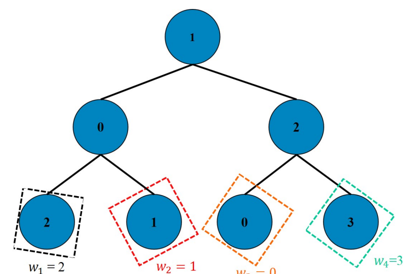


Fig. 12: Sample run of Algorithm 3 after initialization (lines 1-3).

Algorithm 3 assigns each leaf to a different diagonal in line 2 (diagonal initialization), as illustrated in Fig. 12 for the Treeplication multiset in Figs. 10,11. Lines 4 and 5 loop through all non-leaf vertices, where in each iteration a vertex is added to one diagonal (line 9 or 11). The intermediate contents of the diagonals (tracked by the sets $g_1, \dots, g_k$) are given in Fig. 13 at iteration $i = 2$ and in Fig. 14 at iteration $i = 3$. Since every tree vertex gets assigned to one diagonal, the output of

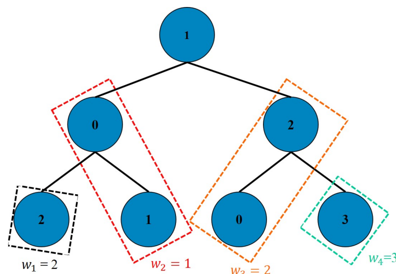


Fig. 13: Sample run of Algorithm 3 after iteration $i = 2$. The diagonals chosen to extend upwards are the lower-weight ones between the diagonals of two sibling vertices.

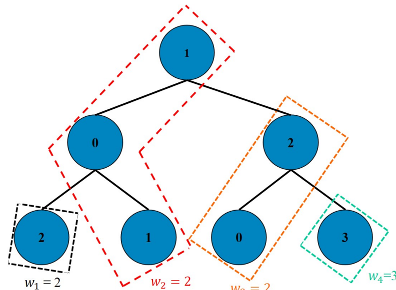


Fig. 14: Sample run of Algorithm 3 after iteration $i = 3$.

the algorithm is a diagonal cover. Next we show that the output diagonal cover is in fact a principal diagonal cover.

Theorem 18. *For any Treeplication-multiset input, Algorithm 3 returns a diagonal cover that is an l-principal diagonal cover for every l.*

Proof: To prove that the algorithm outputs a principal diagonal cover, we need to show that the probability in (19) is maximized among all possible covers. Since k, l and n are constants that do not depend on the cover, maximizing the average probability of (19) is equivalent to minimizing

$$\sum_{j=1}^{k} \binom{n-w_j}{l-w_j}. \tag{20}$$

Given a diagonal cover, for each non-leaf vertex we distinguish between its child that is assigned to the same diagonal (mate child), and its child that is in a different diagonal (non-mate child). In any cover every non-leaf vertex has one mate and one non-mate child. Algorithm 3 guarantees the property that for any non-leaf vertex, the sum of vertex weights below it in its diagonal is less than or equal to the weight of the diagonal of its non-mate child. Now we assume by contradiction that a different algorithm outputs a cover with lower sum (20) (and, hence, higher probability (19)). In that output we have at least one vertex in which this property is not met. For this vertex, denote by q_1 the sum of vertex weights below it in its diagonal, and by q_2 the weight of the diagonal of its non-mate child. By the contradiction assumption we have $q_1 > q_2$. We now show that, moving this vertex (and all vertices above it in its diagonal) to the diagonal of its non-mate child will result in a lower sum (20). Denote by x the total weight moved in that operation. To prove that, we need to show that

$$\binom{n-q_2}{l-q_2} + \binom{n-q_1-x}{l-q_1-x} \geq \binom{n-q_1}{l-q_1} + \binom{n-q_2-x}{l-q_2-x}, \tag{21}$$

where the inequality is between terms of (20) in which the two outputs differ (all other terms are equal and cancel out). To see that (21) is true we use elementary combinatorial identities on Pascal's triangle as follows. Define $\pi_r(j) \triangleq \binom{r+j-1}{r}$, where $\pi_r(j)$ is called the j-th element in the r-th diagonal of Pascal's triangle. A well-known identity for Pascal's triangle is $\pi_r(j) = \sum_{y=0}^{j} \pi_{r-1}(y)$. With that notation, we can write

$$\binom{n-q_2}{l-q_2} - \binom{n-q_1}{l-q_1} = \sum_{y=l-q_1+2}^{l-q_2+1} \pi_{n-l-1}(y), \tag{22}$$

$$\binom{n-q_2-x}{l-q_2-x} - \binom{n-q_1-x}{l-q_1-x} = \sum_{y=l-q_1-x+2}^{l-q_2-x+1} \pi_{n-l-1}(y). \tag{23}$$

Another well known property of Pascal's triangle is that elements of its diagonals increase with the argument j. This implies

$$\sum_{j=l-q_1+2}^{l-q_2+1} \pi_{n-l-1}(j) \geq \sum_{j=l-q_1-x+2}^{l-q_2-x+1} \pi_{n-l-1}(j), \tag{24}$$

because the two sums have the same number of summands and the left-hand side is shifted to larger arguments. From (24) and (22),(23) we conclude (21), which contradicts the existence of a higher-probability diagonal cover. ■

Since the diagonal cover in Fig. 11 is the output of Algorithm 3, the principal 3-health of the multiset is now proved to be 0.935.

A. Empirical tree-health distribution

To illustrate the tree-health measure proposed in this section, we show the distribution of principal l-healths of 1000 multisets drawn from the optimal non-uniform selection distribution of Section IV. We take $k = 32$ and multiset size $n = 3k = 96$, and plot the distribution of the principal l-health, for $l = 32$ (after running Algorithm 3 on each multiset). The results are plotted in the histogram of Fig. 15. It can be seen that multisets from the same selection distribution have significant health variability. In practice, if each multiset represents a different data unit stored in the system, we will seek to invest storage resources in increasing the multiset sizes of the data units with the lowest principal l-health. In the next section, we discuss methods to improve the multiset health in a decentralized way.

VII. Distributed Code Augmentation

When a data unit in the system has low tree health as defined in the previous section, a natural corrective measure is to augment it by adding more code fragments. In this section, we address the problem of deciding *which* code fragments to

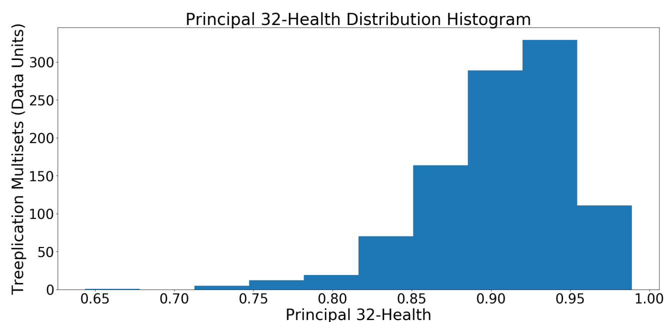


Fig. 15: Distribution of the principal 32-health across 1000 Treeplication multisets. $k = 32, n = 3k$.

add for this data unit. To fit within decentralized distributed systems, we seek solutions that make those decisions without knowing the full current state of the Treeplication multiset of the data unit. Suppose $\mathcal{N}$ is the set of nodes storing code fragments of a particular data unit. In the sequel, augmenting is done by a node not previously in $\mathcal{N}$ that generates and stores a code fragment while only having access to a subset $\mathcal{N}_A \subset \mathcal{N}$ of nodes, which we call the *accessible nodes*.

Definition 15. *Given a subset of accessible nodes of a data unit, we define* ***distributed augmentation*** *as the operation of adding a code fragment using information from accessible nodes.*

Note that the distributed augmentation operation is divided into first deciding which code fragment to add, and then communicating code fragments from accessible nodes to generate this code fragment. Restricting the set of accessible nodes to be small improves the efficiency in a distributed system, because fewer accessible nodes mean fewer resources used by the node performing augmentation.

A simple example of distributed augmentation for uncoded replication is augmenting by adding a data fragment chosen from the data fragments in the accessible nodes. For Treeplication, we next propose a scheme that uses the tree structure of the code to augment data units using a small accessible node set $\mathcal{N}_A$.

A. Augmenting Treeplication with small accessible node sets

We now specify a procedure for distributed augmentation where the accessible nodes are those that hold fragments of two sibling vertices and their parent.

Treeplication Augmentation 1.

1) *Pick a node z in $\mathcal{N}$*
2) *Identify the vertex σ stored in z*
3) *Take $\mathcal{N}_A$ to be all nodes storing either σ, its parent ρ, or its sibling σ'.*
4) *If only σ is present in $\mathcal{N}_A$, augment by replicating it. Else:*
5) *Choose the vertex from σ,σ' with lower weight and augment by replicating it (if exists), or generating it using ρ (if not).*

Remarks: 1) In step 5, we always choose to augment one of the two siblings σ,σ', but need the parent ρ in cases where the sibling σ' has zero weight. 2) In case of equal-weight vertices in step 5, we break ties arbitrarily.

In a real system, a reasonable way to choose z from $\mathcal{N}$ in step 1 of Treeplication Augmentation 1 is uniformly. The size of the accessible node set $\mathcal{N}_A$ equals the number of appearances of σ,ρ,σ' in the multiset, which depends on the chosen z. The intuition to augment the lower-weight (weaker) sibling is that it is better than the stronger sibling in improving the probability that the subtree remains decodable locally, and better than the parent in improving the survival probability when the fragment ρ can be recovered from elsewhere in the full tree.

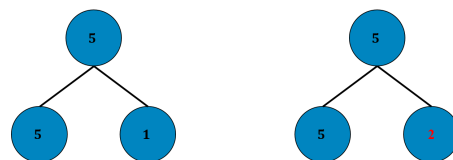
Later in the section, we compare Treeplication Augmentation 1 to the following simpler alternative.

Augmentation by replication.

1) *Pick a node z in $\mathcal{N}$*
2) *Identify the vertex σ stored in z*
3) *Replicate σ*

In Augmentation by replication, we simply copy the code fragment from the node we picked in line 1 to the new node entering $\mathcal{N}$. This procedure can apply to both Treeplication and standard replication, where in the latter σ is always a data fragment.

An example of Treeplication Augmentation 1 is presented in Fig. 16 for a $k = 2$ tree and a sample Treeplication multiset. Because in Fig. 16a the right leaf has lower weight than the left leaf, the former is chosen to be augmented (Fig. 16b). Before augmentation, the probability that the multiset remains decodable (survives) after loss of 9 nodes is 0.64, and increasing to 0.91 after augmenting by one code fragment. If we chose the other (stronger) leaf for augmentation, the survival probability would only go up to 0.86.



(a) Before augmentation. (b) After augmentation.

Fig. 16: Treeplication Augmentation 1 for a $k = 2$ tree and a sample Treeplication multiset.

We propose Treeplication Augmentation 1 mainly as an example how with a small accessible set $\mathcal{N}_A$ we can improve survivability over Augmentation by replication. There are many possible generalizations and enhancements of Treeplication Augmentation 1 that can further improve performance. For example, it is possible to consider bigger subtrees in the augmentation choice, and solve interesting optimization problems to maximize global survival probability given local subtree information.

B. Empirical study: augmentation in node birth-death processes

To study and compare augmentation schemes in distributed systems, we next define a dynamic system setup where augmentation affects the long term survival of data units. To that end, we use a discrete-time *birth-death process* to model the dynamics of the data-unit's node set. At each time instant, an event of either birth (addition of a node) or death (removal of a node) occurs in the data unit. If birth is drawn, adding a node invokes an augmentation operation. If death is drawn, a randomly selected node is chosen to be removed from $\mathcal{N}$ of that data unit. We restrict ourselves to balanced processes, where birth and death occur each with probability 0.5. In general, birth-death processes may have time instants where neither birth nor death occurs, but for the purpose of comparing augmentation schemes these time instants are not interesting. We use the term *generation* to define the state of the data unit following the node removal in death instants and augmentation in birth instants. We define the *data loss* event for the data unit as the first generation where the data unit becomes non-decodable. Since data-loss is irreversible, the process terminates immediately after. In our results the number of generations a data unit *survives* is the number of process instants before the data-loss event.

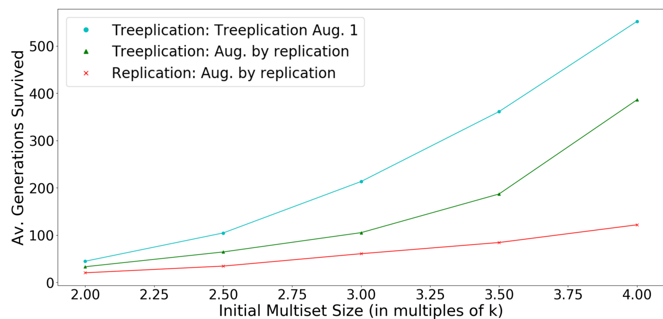


Fig. 17: Augmentation in a birth-death process. From bottom to top: replication with Augmentation by replication, Treeplication with Augmentation by replication, and Treeplication with Treeplication Augmentation 1. $k = 32$.

In Fig. 17, we compare the average number of generations survived in three different setups: 1) replication with Augmentation by replication, 2) Treeplication with Augmentation by replication, and 3) Treeplication with Treeplication Augmentation 1. For each setup we simulated the birth-death process, and plotted the average number of generations survived across 3000 runs. In each run we randomly drew the 0-th generation of the data unit: for Treeplication using the optimal non-uniform distribution (from Section IV), and for replication uniformly from the data fragments. Fig. 17 demonstrates that Treeplication fares better than replication also in the dynamic regime, and more interestingly, that Treeplication Augmentation 1 improves significantly over Augmentation by replication.

VIII. Conclusion

We have shown that Treeplication codes combine the strength of erasure codes in recoverability with replication-like access performance. The tree structure of Treeplication allows deriving exact recursive expressions toward the analysis and optimization of the code under random-multiset models. It is an interesting open problem to generalize the code structure beyond a binary tree while still maintaining the algorithmic and analysis capabilities we demonstrated for Treeplication.

IX. Acknowledgement

We thank the associate editor and the anonymous referees for comments that improved the presentation of this paper. This work was supported in part by the Israel Science Foundation, and in part by the US-Israel Binational Science Foundation.

References

[1] M. Blaum, P. Farrell, and H. van Tilborg, "Array Codes," *Handbook of Coding Theory, V.S. Pless and W.C. Huffman*, pp. 1855–1909, 1998.
[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[3] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Trans. Inf. Theory*, vol. 56, no. 9, pp. 4539–4551, 2010.
[4] J. Edmonds and M. Luby, "Erasure codes with a hierarchical bundle structure," *IEEE Trans. on Information Theory*, early access.
[5] M. Elyasi and S. Mohajer, "Determinant coding: A novel framework for exact-repair regenerating codes," *IEEE Trans. on Information Theory*, vol. 62, no. 12, pp. 6683–6697, 2016.
[6] A. Folsom, Y. Homma, J. H. Ryu and B. Tong, "On a general class of non-squashing partitions," *Discrete Mathematics*, vol. 339, no. 5, pp. 1482–1506, 2016.
[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," Proceedings ACM Symposium on Operating Systems Principles (SOSP), 2003.
[8] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin, "On the locality of codeword symbols," *IEEE Trans. on Information Theory*, vol. 58, no. 11, pp. 6925–6934, 2012.
[9] R. Graham, D. Knuth, and O. Patashnik, "Concrete Mathematics: A Foundation for Computer Science," (Second Edition). Reading, Massachusetts: Addison Wesley, pp. 257–266, 1994.
[10] M. Hirschhorn, "A different view of m-ary partitions," *Australian J. of Combinatorics*, vol. 30, pp. 193–196, 2004.
[11] K. Mahler, "On a special functional equation," *J. Lond. Math. Soc.*, vol. 15, pp. 115–123, 1940.
[12] M. Mitzenmacher and E. Upfal, "Probability and Computing,". London, Cambridge University Press, 2005.
[13] F. Oggier and A. Datta, "Self-repairing homomorphic codes for distributed storage systems," *Proceedings IEEE INFOCOM*, 2011.
[14] L. Pamies-Juarez, H. D. L. Hollmann, and F. E. Oggier, "Locally repairable codes with multiple repair alternatives," *Proceedings IEEE International Symposium on Information Theory*, 2013.
[15] D. S. Papailiopoulos and A. G. Dimakis, "Locally repairable codes," *IEEE Trans. on Information Theory*, vol. 60, no. 10, pp. 5843–5855, 2014.
[16] N. Prakash, G. M. Kamath, V. Lalitha, and P. V. Kumar, "Optimal linear codes with a local-error-correction property," *Proceedings IEEE International Symposium on Information Theory*, 2012.
[17] K.V. Rashmi, N.B. Shah, and P.V. Kumar, "Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix Construction," *IEEE Trans. on Information Theory*, vol. 57, no.8, pp. 5227–5239, 2011.
[18] I.S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *SIAM J. Appl. Math*, vol. 8, no. 2, pp. 300–304, 1960.
[19] N. Silberstein, A.S. Rawat, O.O. Koyluoglu, and S. Vishwanath, "Optimal locally repairable codes via rank-metric codes," *Proceedings IEEE International Symposium on Information Theory*, 2013.
[20] N. J. A. Sloane, and J. A. Sellers, "On non-squashing partitions," *Discrete Mathematics*, vol. 294, no. 3, pp. 259–274, 2005.

[21] I. Tamo and A. Barg, “A family of optimal locally recoverable codes,” *IEEE Trans. on Information Theory*, vol. 60, no. 8, pp. 4661–4676, 2014.

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

[23] C. Tian, B. Sasidharan, V. Aggarwal, V. A. Vaishampayan, and P. V. Kumar, “Layered exact-repair regenerating codes via embedded error correction and block designs,” *IEEE Trans. on Information Theory*, vol. 61, no. 4, pp. 1933–1947, 2015.

[24] M. Ye and A. Barg, “Explicit constructions of high-rate MDS array codes with optimal repair bandwidth,” *IEEE Trans. on Information Theory*, vol. 63, no. 4, pp. 2001–2014, 2017.