In-Network Redundancy Generation for Opportunistic Speedup of Backup

Lluis Pamies-Juarez*, Anwitaman Datta†, Frédérique Oggier*

*Division of Mathematical Sciences
School of Physical and Mathematical Sciences
Nanyang Technological University, Singapore

†Division of Computer Science
School of Computer Engineering
Nanyang Technological University, Singapore

Abstract—Erasure coding is a storage-efficient alternative to replication for achieving reliable data backup in distributed storage systems. During the storage process, traditional erasure codes require a unique source node to create and upload all the redundant data to the different storage nodes. However, such a source node may have limited communication and computation capabilities, which constrain the storage process throughput. Moreover, the source node and the different storage nodes might not be able to send and receive data simultaneously – e.g., nodes might be busy in a datacenter setting, or simply be offline in a peer-to-peer setting – which can further threaten the efficacy of the overall storage process. In this paper we propose an “in-network” redundancy generation process that leverages on the self-repairing property of the novel SRC codes. This in-network redundancy generation allows storage nodes to generate new redundant data by exchanging partial information among themselves, improving the throughput of the storage process. The process is carried out asynchronously, utilizing spare bandwidth and computing resources from the storage nodes. We analytically show that the performance of this technique relies on an efficient usage of the spare node resources, and we derive a set of scheduling algorithms to maximize the same. We experimentally show that our algorithms can, depending on the environment characteristics, increase the throughput of the storage process significantly with respect to the classical naive storage approach.

Keywords—distributed storage; erasure codes; backup;

I. INTRODUCTION

There is a continued and rapid global growth in data storage needs. Archival and backup storage form a specific niche, of importance to both businesses and individuals. A recent market analysis from IDC[1] stated that the global revenue of the data archival business is expected to reach $6.5 billion in 2015. The necessity to cost-effectively scale-up data backup systems to meet this ever growing storage demand poses a challenge to storage systems designers.

When large volume of data is involved, deploying a networked distributed storage system becomes essential, since a single storage node cannot scale. Furthermore, distribution provides opportunities for fault tolerance and parallelized I/O. Examples of such distributed storage systems are readily found in datacenter environments, including distributed file systems such as GFS[1] or HDFS[2], distributed key-value stores like Dynamo[3] or Cassandra[4], as well as in ad-hoc end user resource based peer-to-peer (P2P) settings such as OceanStore[5] and friend-to-friend (F2F) storage systems[6,7], a special kind of peer-to-peer systems often considered particularly suitable for personal data backup.

An important design aspect in distributed storage systems is redundancy management. Data replication provides a simple way to achieve high fault-tolerance, while erasure codes such as Reed-Solomon codes[8] are more sophisticated alternatives, capable of significantly reducing the data storage footprint for different levels of fault-tolerance[9–12]. Various trade-offs in adopting erasure codes in storage systems, such as storage overhead & fault-tolerance, access frequency & decoding overheads and repair bandwidth & repair time after failures, have been studied in the literature, revealing in particular that erasure codes are particularly suitable for backup and archival storage, where data access is infrequent, and hence the effects of decoding are marginal. A relatively unexplored aspect of the usage of erasure codes in storage systems is the time required to insert new data.

When using replication, a source node aiming to store new data uploads one replica of this data to the first storage node, which can concurrently forward the same data to a second storage node, and so on. Doing so, the load for redundancy insertion can be shared, and the source node may not need to upload any redundant information itself. Replication thus naturally supports “in-network” generation of redundancy, that is, generation of new redundancy within the network, through data exchange among storage nodes, which in turn leads to fast insertion of data. In contrast, in erasure encoded systems, the source node is the one responsible for computing and uploading all the encoded redundant data to the corresponding storage nodes. The amount of data the source node uploads is then considerably larger, resulting in longer data insertion times. Insertion latency may further be exacerbated when the source node and the set of storage

1http://www.idc.com/getdoc.jsp?containerId=230762
nodes have additional (mismatched) temporal constraints on resources availability, in which case in-network redundancy generation can provide partial mitigation.

We elaborate the effect of temporal resource (un)availability issues with two distinct example scenarios, which we also use later in our experiments to determine how (much) in-network redundancy generation may help:

- In datacenters, storage nodes might be used for computation processes which require efficient access to local disks. Since backup processes consume large amounts of local disk I/O, system administrators might want to avoid backup transfers while nodes are executing I/O intensive tasks – e.g., Mapreduce tasks.
- In F2F backup applications, users exchange some of their spare disk resources with their friends in order to realize a collaborative data backup service. However, different users may be online at different times of the day.

In both cases the insertion of new redundancy by the source node is restricted to the periods when the availability windows of the source overlap that of the storage nodes.

Unlike replication, where in-network redundancy generation is achieved trivially, traditional erasure codes are not amenable. Our network coding inspired solution is based on a novel family of erasure codes called Self-Repairing Codes (SRC), recently proposed in order to improve data repair efficiency. A salient property of SRC codes is that the encoded data stored at each node can be easily regenerated by using information from a few other storage nodes. As we will show, this is the key property to achieve in-network redundancy generation. However, SRC codes have strict constraints on how storage nodes can combine their data to generate content for other nodes, which, along with the temporal availability constraints of nodes, complicates the design of efficient in-network redundancy generation.

The main contributions of this paper are as follows: (i) we introduce the concept of in-network redundancy generation for reducing data insertion latency in erasure code based storage systems, (ii) we define an analytical framework to explore valid transfer schedules, (iii) we show that determining optimal schedules is computationally intractable, and (iv) we propose a set of heuristic algorithms for efficient in-network redundancy generation.

We experimentally validate these ideas using real availability traces from friend-to-friend (F2F) and peer-to-peer (P2P) applications, and synthetic traces to explore datacenter-like scenarios and show that these algorithms substantially increase the throughput of the backup process.

II. RELATED WORK

The P2P research community has long studied the applicability of erasure codes in low availability environments with limited storage capacity [5], [11]. More recently, there has been a growing interest in applying erasure codes in datacenters to reduce storage costs [15]–[18].

While I/O and bandwidth are well recognized critical bottlenecks for the storage of huge amounts of data, existing literature does not explore how data insertion can be optimized in the context of erasure codes based storage. We instead identify and discuss some peripherally related works.

Decentralized erasure codes [19] explored in the context of sensor networks [20], [21] are arguably the closest related works. In such settings, the (disjoint) data generated by $k$ sensors is jointly and redundantly stored by $n$ storage sensors based on erasure coding principles, where the data is re-distributed among the storage (sensor) nodes using network coding [22], a popular mechanism deployed to improve the throughput utilization of a given network topology. This line of work did not explore the effect of temporal unavailability of nodes during the redundancy generation process, and also does not map readily to the scenario of one data source injecting data to other storage nodes, as studied in this work.

Benefits of network coding have further been exploited in [12] to restore encoded fragments (lost due to failures) in a distributed manner, while [13] achieves the same by instead designing customized Self-Repairing codes. This work leverages on Self-Repairing codes in order to carry out in-network redundancy generation for opportunistically speeding up the data insertion and backup process.

III. ERASURE CODES AND SELF-REPAIRING CODES

In this section, we provide some background on erasure codes as classically used for distributed storage, as well as on the newly introduced class of Self-Repairing Codes (SRC).

A. Erasure codes for distributed storage.

A classical $\langle n, k \rangle$ erasure code allows to redundantly encode an object of size $M$ into $n$ redundant fragments of size $M/k$, each to be stored in a different storage node. The data storage overhead (or redundancy factor) is then given by $n/k$, and the stored object can be reconstructed by downloading an amount of data equal to $M$, from $k$ or more different nodes out of $n$.

One of the main drawbacks of using classical erasure codes for storage is that redundant fragments can only be generated by applying coding operations on the original data. The generation of new redundancy is then restricted to nodes that possess the original object (or a copy), namely: the source node, storage nodes that previously reconstructed the original object, or possibly were storing a copy (as is the case in a hybrid model where a full copy of the object is kept, together with encoded fragments). When the original raw object is not available, repairing a single
node failure consequently entails downloading an amount of information equivalent to the size of the original object, causing a significant communication overhead.

In order to mitigate this communication overhead, a new family of erasure codes called Regenerating Codes was recently designed by adopting ideas from network coding. The main advantage of Regenerating Codes is that new redundant fragments can be generated by downloading an amount of data \( \beta \) from \( d \) other redundant fragments, where \( d \geq k \) and \( \beta \leq M/k \). Unlike in classical erasure codes, Regenerating Codes can thus repair missing fragments by downloading only an amount of data equals to \( d \beta \), where usually \( d \beta \ll M \). However, the biggest communication savings occur when \( d \geq k \) is a large value, in which case however, the probability to find \( d \) nodes available might be very scarce, limiting the practicality of such codes.

B. Homomorphic Self-Repairing Codes (HSRC).

Self-Repairing Codes (SRC) are new erasure codes designed to minimize the maintenance overhead by reducing the number of nodes \( d \) required to be contacted to recreate lost fragments. A specific family of SRC codes is Homomorphic Self-Repairing codes (HSRC), where two encoded fragments can be xorred for such regeneration, i.e. \( d = 2 \), as long as not more than half of the nodes have failed. This property makes HSRC codes very suitable for the in-network redundancy generation since partial redundant data stored in two different nodes can be used to generate data for a third node, without requiring the intervention of the source node. However, as we will show below, the pairs of nodes used for that purpose cannot be arbitrary chosen.

Let us recall briefly the construction of HSRC. We denote finite fields by \( \mathbb{F} \). The cardinality of \( \mathbb{F} \) is given by its index, that is, \( \mathbb{F}_2 \) is the binary field with two elements (the two bits 0 and 1), and \( \mathbb{F}_2^q \) is the finite field with \( q \) elements. If \( q = 2^m \), for some positive integer \( m \), we can fix a \( \mathbb{F}_2 \)-basis of \( \mathbb{F}_q \) and represent an element \( x \in \mathbb{F}_2^m \) using an \( m \)-dimensional vector \( x = (x_1, \ldots, x_m) \) where \( x_i \in \mathbb{F}_2 \), \( i = 1, \ldots, m \).

Let \( o \) be the object to be stored over a set of \( n \) nodes, which is represented as a data vector of size \( k \times m \) bits, i.e.:

\[
o = (o_1, \ldots, o_k), \quad o_i \in \mathbb{F}_2^m.
\]

Given these \( k \) original elements, the \( n \) redundant fragments are obtained by evaluating the polynomial

\[
p(X) = \sum_{i=1}^{k} o_i X^{2^{i-1}} \in \mathbb{F}_2^m[X]
\]

in \( n \) non-zero values \( o_1, \ldots, o_n \) of \( \mathbb{F}_2^m \), yielding the redundant vector \( r \) of size \( n \times m \) bits, i.e.:

\[
\mathbf{r} = (r_1, \ldots, r_n), \quad r_i = p(o_i) \in \mathbb{F}_2^m.
\]

In particular we need the code parameters \( \langle n, k \rangle \) to satisfy

\[
1 < k < n \leq 2^m - 1.
\]

IV. HSRC Redundancy Generation

The main important property of HSRC codes is its homomorphic property. From [13] we have that:

**Lemma 1**: Let \( a, b \in \mathbb{F}_2^m \) and let \( p(X) \) be the polynomial defined in (1), then \( p(a+b) = p(a) + p(b) \).

This implies that we can generate a redundant element \( r_k = p(o_k) \) from \( r_1 = p(o_1) \) and \( r_2 = p(o_2) \) if \( o_k = o_1 + o_2 \).

**Example 1**: Consider a \( \langle n = 7, k = 3 \rangle \) HSRC code and an object \( o = (o_1, o_2, o_3) \) of size \( 3 \times 4 \) bits, where \( o_i \in \mathbb{F}_2^4, \quad i = 1, 2, 3 \). We write \( o_1 = (o_{11}, o_{12}, o_{13}, o_{14}), \quad o_2 = (o_{21}, o_{22}, o_{23}, o_{24}), \quad o_3 = (o_{31}, o_{32}, o_{33}, o_{34}), \) from which we compute \( p(X) = \sum_{i=1}^{k} o_i X^{2^{i-1}} \). We evaluate \( p(X) \) in \( n = 7 \) values of \( \mathbb{F}_2^4 \), represented in vector form as \( o_1 = (1, 0, 0, 0), \quad o_2 = (0, 1, 0, 0), \quad o_3 = (1, 1, 0, 0), \quad o_4 = (0, 0, 1, 0), \quad o_5 = (1, 0, 1, 0), \quad o_6 = (0, 1, 1, 0), \quad o_7 = (1, 1, 1, 0), \) yielding:

\[
r_1 = p(o_1) = (o_{11} + o_{21} + o_{31}, o_{12} + o_{22} + o_{32}, o_{13} + o_{23} + o_{33}, o_{14} + o_{24} + o_{34}).
\]

\[
r_2 = p(o_2) = (o_{11} + o_{21} + o_{31} + o_{34}, o_{12} + o_{22} + o_{32} + o_{34}, o_{13} + o_{23} + o_{32} + o_{34}, o_{14} + o_{24} + o_{33} + o_{34}).
\]

\[
r_3 = p(o_3) = (o_{11} + o_{21} + o_{31} + o_{34}, o_{12} + o_{22} + o_{32} + o_{34}, o_{13} + o_{23} + o_{32} + o_{34}, o_{14} + o_{24} + o_{33} + o_{34}).
\]

\[
r_4 = p(o_4) = (o_{11} + o_{21} + o_{31} + o_{34}, o_{12} + o_{22} + o_{32} + o_{34}, o_{13} + o_{23} + o_{32} + o_{34}, o_{14} + o_{24} + o_{33} + o_{34}).
\]

\[
r_5 = p(o_5) = (o_{11} + o_{21} + o_{31} + o_{34}, o_{12} + o_{22} + o_{32} + o_{34}, o_{13} + o_{23} + o_{32} + o_{34}, o_{14} + o_{24} + o_{33} + o_{34}).
\]

We can check that

\[
p(o_7) = p(o_1) + p(o_6) = p(o_2) + p(o_3) = p(o_4) + p(o_5).
\]

Note that we have not used the vector \( o_k = (0, 0, 0, 1) \) here, which would have resulted in a longer code \( n > 7 \).

We now discuss how HSRC codes operate in two different scenarios: (i) when the source introduces data in the system, and (ii) during the in-network redundancy generation.

A. Source Redundancy Generation

The homomorphic property described in Lemma [11] has been introduced to repair node failures, though it can similarly serve to generate redundancy from the source. Recall from Lemma [11] that \( p(a + b) = p(a) + p(b) \), where both \( a, b \) can be seen as \( m \)-dimensional binary vectors, by fixing a \( \mathbb{F}_2 \)-basis of \( \mathbb{F}_2^m \). Let us denote this basis by \( \{b_1, \ldots, b_m\} \).

Thus \( a \) can be written as \( a = \sum_{i=1}^{m} a_ib_i \), \( a_i \in \mathbb{F}_2 \), and by virtue of the homomorphic property, we get that

\[
p(a) = p \left( \sum_{i=1}^{m} a_ib_i \right) = \sum_{i=1}^{m} a_ip(b_i).
\]
This means that the source only needs to compute \( p(b_1), \ldots, p(b_m) \) for a given basis \( \{b_1, \ldots, b_m\} \), after which all the other encoded fragments are obtained by xoring pairs of elements in \( \{p(b_1), \ldots, p(b_m)\} \). Thus, when using an \((n, k)\) HSRC code, the source computes \( k \leq m \) encoded fragments \( r_1 = p(\alpha_1), \ldots, r_k = p(\alpha_k) \), where \( \alpha_1, \ldots, \alpha_k \) are linearly independent, for example, \( \{\alpha_1, \ldots, \alpha_k\} \subset \{b_1, \ldots, b_m\} \), and then performs the corresponding xoring.

The source then injects the \( n \) encoded fragments in the network.

**Example 2:** In Example 1 we have \( k = 3 \leq m = 4 \), and a natural \( \mathbb{F}_2 \)-basis for \( \mathbb{F}_{2^4} \) is \( b_1 = (1, 0, 0, 0), b_2 = (0, 1, 0, 0), b_3 = (0, 0, 1, 0), b_4 = (0, 0, 0, 1) \). The source can generate redundancy by first computing \( r_1 = p(\alpha_1) = p(b_1), r_2 = p(\alpha_2) = p(b_2), r_4 = p(\alpha_4) = p(b_3) \), then \( r_3 = p(\alpha_3) = p(\alpha_1) + p(\alpha_2), r_5 = p(\alpha_5) = p(\alpha_1) + p(\alpha_4) \), \( r_6 = p(\alpha_6) = p(\alpha_2) + p(\alpha_4) \) and \( r_7 = p(\alpha_7) = p(\alpha_1) + p(\alpha_6) \). The \( n = 7 \) encoded fragments are then ready to be sent over the network. Further notice that the set \( B = \{p(\alpha_1), p(\alpha_2), p(\alpha_4)\} \) can be seen as a basis for the set of redundant fragments, since they are linearly independent, and can be combined to generate every redundant fragment.

**B. In-Network Redundancy Generation**

Let us now consider the case where the source might not inject the whole set of \( n \) encoded fragments, but only a subset \( \{r_i, i \in I \subset \{1, \ldots, n\}\} \) of the encoded fragments.

We use the triplet notation \((i, j) \vdash k\) to represent the possibility to generate the element \( r_k \) by xoring \( r_i \) and \( r_j \), \( r_k = r_i + r_j \). Note that due to the commutative property of the additive operator, triplets \((i, j) \vdash k\) and \((j, i) \vdash k\) can be indistinguishably used to denote the same redundancy generation process. We denote by \( C \) the set with all the feasible repair triplets from a set of \( n \) redundant elements. Finally, let us define the following two sets:

**Definition 1 (out-creation set):** Let \( O(i) \) be the set of all the possible \((i, j) \vdash k\) triplets where fragment \( r_i \) is used to generate some other fragment:

\[
O(i) = \{(i, j) \vdash k \mid j = 1, \ldots, n, j \neq i, \text{k s.t.} \ r_k = r_i + r_j\}.
\]

**Definition 2 (in-creation set):** Let \( I(k) \) be the set of all the possible \((i, j) \vdash k\) triplets that can be used to create \( r_k \):

\[
I(k) = \{(i, j) \vdash k \mid r_k = r_i + r_j; \ i, j = 1, \ldots, n\}.
\]

Finally, given a number of redundant elements \( n = 2^t - 1 \), for any positive integer \( t \leq m \), we have from \( \{13\} \) that:

\[
|O(i)| = n - 1 \quad (2)
\]

\[
|I(k)| = \frac{(n - 1)}{2}. \quad (3)
\]

**Example 3:** In Example 1 we have that

\[
O(1) = \{(1, 3) \vdash 2, (2, 3) \vdash 2, (1, 5) \vdash 4, (1, 4) \vdash 5, (1, 7) \vdash 6, (1, 6) \vdash 7\},
\]

\[
I(7) = \{(1, 6) \vdash 7, (2, 5) \vdash 7, (3, 4) \vdash 7\}.
\]

**C. HSRC Codes: Practical Implementation**

Previously we detailed how to encode a data vector \( o \) of size \( k \times m \) bits into a redundant vector \( r \) of size \( n \times m \) bits. We showed that HSRC codes allow this encoding by using data from the source node as well as by using data from other storage nodes. In this subsection we describe one method to practically implement HSRC codes to encode larger data objects of size \( M \), where \( M > k \times m \).

The first step to encode an object of size \( M \) is to split it into \( u = M / (k \times m) \) vectors\(^2\) of size \( k \times m \) bits. Let us represent the object to be encoded as \( o = (o_1, \ldots, o_M) \). After the splitting process, \( o = (\bar{o}_1, \ldots, \bar{o}_u) \), where \( o_i = (o_{ki(i-1)+1}, \ldots, o_{ki(i-1)+k}) \in \mathbb{F}_{2^m}, k = 1, \ldots, k \).

Each of these vectors \( \bar{o}_i \) is individually encoded using the polynomial \( \bar{p} \) to obtain an encoded vector \( \bar{r} = (\bar{r}_1, \ldots, \bar{r}_u) \). For each of the individual vectors \( \bar{r}_i \) the source can forward \( \bar{r}_i \) to the \( i \)-th node, and distribute \( o_i \) to the \( i \)-th node. After encoding each of the individual vectors we obtain the set of redundant vectors \( \bar{r} = (\bar{r}_1, \ldots, \bar{r}_i) \) (\( \bar{r}_i \) is encoded to obtain \( \bar{r}_i \)), where \( |\bar{r}_1| = |\bar{r}_2| = |\bar{r}_3| = 7 \). Finally, we obtain \( r_2 = (r_{1,2}, r_{2,2}, r_{3,2}) \), and similarly for all the fragments.

**Remark 1:** Note that this encoding technique allows stream encoding. As soon as the source node receives the first \( k \times m \) bits to store\(^3\) it can generate the vector \( \bar{o}_1 \), encode it to \( \bar{r}_1 \), and distribute \( \bar{r}_1, \ldots, \bar{r}_u \) to the \( n \) storage nodes. Similarly, when a storage node receives \( \bar{r}_i \) it can forward it to other nodes for in-network redundancy generation, for instance, when the source does not have adequate bandwidth to upload all the \( n \) redundant fragments.

**Remark 2:** To implement computationally efficient codes one can set \( m = 8 \), or \( m = 32 \), for which addition can simply be done by xoring system words, and for which efficient arithmetic libraries are available\(^{23}\).

In the rest of this paper we will assume that HSRC codes are implemented using the method described here. We will use the term redundant fragment to refer to each of the redundant elements \( r_1, \ldots, r_n \), i.e., each node stores one redundant fragment. And similarly we will use the

\(^2\)We assume that \( k \times m \mid M \). Otherwise the object can be zero-padded to guarantee it.

\(^3\)A gateway node in a datacenter receiving data from a web application end user would be treated as the source node in our model. In such scenarios, the source itself may not be in possession of the whole data in advance.
term redundant chunk to refer to each of the sub-elements \( F_{1,i}, \ldots, F_{u,i} \) stored in each node \( i \), i.e., each node \( i \) can store up to \( u \) redundant chunks.

V. SCHEDULING THE IN-NETWORK REDUNDANCY GENERATION

In-network redundancy generation has the potential to speedup the insertion of new data in distributed storage systems. However, the magnitude of actual benefit depends on two factors: (i) the availability pattern of the source and storage nodes, which determines the achievable throughput, and (ii) the specific schedule of data transfer among nodes subject to the constraints of resource availability, which determines the actual achieved throughput for data backup. In this section, we explore the scheduling problem, demonstrating that finding an optimal schedule is computationally very expensive even with a few simplifying assumptions, and accordingly motivate some heuristics instead.

Let \( s \) be a source node aiming to store a new data object to \( n \) different storage nodes, and let \( i, i = 1, \ldots, n \), represent each of these \( n \) storage nodes. We model our system using discrete time steps of duration \( \tau \), where at each time step nodes can be available or unavailable to send/receive redundant data. The binary variable \( a(i, t) \in \{0, 1\} \) denotes this availability for each node \( i \) for the corresponding time step \( t \). Using this binary variable we can define the maximum amount of data that node \( i \) can upload during time step \( t \) by

\[
u(i, t) = a(i, t) \cdot \omega_i \uparrow (t) \cdot \tau,
\]

where \( \omega_i \uparrow (t) \) is the upload capacity of node \( i \) during time step \( t \). Similarly, the amount of data each node can download during time step \( t \) is given by

\[
d(i, t) = a(i, t) \cdot \omega_i \downarrow (t) \cdot \tau,
\]

where \( \omega_i \downarrow (t) \) represents the download capacity of node \( i \) during time step \( t \).

Then, we define the in-network redundancy generation network as a weighted temporal directed graph \( G = (E(t), V(t)) \), \( t \geq 0 \), with the set of nodes \( V(t) \subset \{s, 1, 2, \ldots, n\} \), and the set of edges \( E(t) = \{(i, j) | i, j \in V(t)\} \). The amount of data that nodes send among themselves is a mapping \( f : E(t) \rightarrow \mathbb{R}^+ \), denoted by \( f(i, j, t) \), \( \forall (i, j) \in E(t), t \geq 0 \).

HSRC code characteristics constrain the mapping \( f \) since nodes can only send or receive data through valid redundancy creation triplets:

\[
f(i, k, t) > 0 \iff \exists c \in \mathcal{C} \text{ s.t. } c = (i, j)^\uparrow k.
\]

Furthermore, we assume (for algorithmic simplicity) that nodes send data through each of the redundancy generation triplets symmetrically:

\[
\forall c \in \mathcal{C}; c = (i, j)^\uparrow k.
\]

For ease of notation we will refer to the data sent through each of the redundancy generation triplets simply by \( R(c, t) \).

Similarly, because of the upload/download bandwidth constraints, the mapping \( f \) must also satisfy the following constraints:

- The amount of data the source uploads is constrained by its upload capacity:
  \[
  \sum_{i=1}^{n} f(s, i, t) \leq u(s, t); \forall i \in V(t).
  \]

- The amount of data storage nodes upload is also constrained by their upload capacity:
  \[
  \sum_{c \in O(i)} R(c, t) \leq u(i, t); \forall i \in V(t).
  \]

- The amount of data storage nodes download is restricted by their download capacity:
  \[
  f(s, i, t) + 2 \sum_{c \in I(i)} R(c, t) \leq d(i, t); \forall i \in V(t).
  \]

A Bandwidth-Valid In-Network Redundancy Generation Scheduling is any mapping \( f \) on \( G \) that satisfies the constraints defined in equations (4), (5), (6), (7) and (8).

A. Optimal Schedule

Let \( \theta(i, t) \) be the amount of data that node \( i \) had received at the end of time step \( t \). For sufficiently large enough files and small values of \( m \) (e.g. \( m = 8 \)), we can assume without loss of generality that \( \theta(i, t)/m \) corresponds to the index of the last redundant chunk received by node \( i \). Let \( \bar{M}(\bar{t}) \) denote the size of the largest possible file that a schedule \( f \) can store in \( \bar{t} \) time steps. Then, by definition of erasure codes, to consider that a file of size \( \bar{M}(\bar{t}) \) has been successfully stored after \( \bar{t} \) time steps, each node must receive an amount of data equal to \( \bar{M}(\bar{t})/k \). Using this fact we can define \( \bar{M}(\bar{t}) \) as:

\[
\bar{M}(\bar{t}) = \min \left( \theta(1, \bar{t}), \ldots, \theta(n, \bar{t}) \right) \times mk.
\]

For a given network \( G \) and a duration \( \bar{t} \), an in-network redundancy generation scheduling \( f \) is then optimal if it maximizes \( \bar{M}(\bar{t}) \).

Note that after \( \bar{t} \) time steps, the overall network traffic required by any schedule \( f \), namely \( T(f, \bar{t}) \), is equal to:

\[
T(f, \bar{t}) = \sum_{t=0}^{\bar{t}} \left( 2 \sum_{c \in \mathcal{C}} R(c, t) + \sum_{i=1}^{n} f(s, i, t) \right).
\]

Accordingly, we define an in-network redundancy generation schedule \( f \) to be an optimal minimum-traffic schedule if besides maximizing \( \bar{M}(\bar{t}) \), it also minimizes \( T(f, \bar{t}) \).

Remark 3: Note that to create the same amount of new redundancy the in-network redundancy generation requires twice the traffic required by the source redundancy generation.
will use Example 4, an in-network redundancy generation condition for the schedule to be actually valid. For that, we valid schedule is a necessary condition, it is not a sufficient B. Additional Scheduling Constraints

Currently to 3 different nodes. Although the 3 different schedules satisfy Figure 1. Example of 3 different in-network redundancy scheduling policies for a system where the source node can only upload data con- dant fragment in nodes due to the limited upload capacity of the source node it can scheduling policies, all depicted in Figure 1. We assume that operation the property time to nodes 1, 2 and 3 their first redundant chunk; at time step 1. However, it appears a circular dependency problem with triplets (1, 6)→7, (2, 7)→5 and (3, 5)→6. To show this dependency, imagine that we want to generate fragment r_{6,1} using non-source data. Note that r_{6,1} requires r_{5,1}, r_{5,1} requires r_{7,1}, which at the same time requires the fragment we aim to generate, r_{6,1}. Although it is a bandwidth-valid schedule, the circular dependency problem makes it an unfeasible schedule.

Finally, in the third case we see how the circular dependency problems can be avoided if the source sends uncorrelated fragments at each time step. It is easy to see from this example that a valid schedule needs to be not only bandwidth-valid, but also ensure that: (i) nodes do not receive duplicated data, and (ii) circular triplet dependencies are prevented.

C. Complexity Analysis

We show that finding an optimal schedule satisfying all the previous requirements is computationally very expensive, even under further simplifying assumptions:

Assumption 1: The amount of data that the source node s sends during each time step t to any storage node i, f(s, i, t), is a constant value and is not part of the optimization problem.

Assumption 2: Storage nodes can only receive redundant chunks sequentially. It means that node i will never receive chunk r_{j+1,i} before previously receiving chunk r_{j,i}.

It is easy to see that the simplified problem subject to these two assumptions corresponds to a specific instance of the generic case described above. The interesting property about this simplified version of the problem is that we can reduce the decision of choosing the optimal schedule f to an algorithm “SortedVector” which sorts C, as it is shown in Algorithm. It is also easy to see, how due to the iterative use of redundancy generation triplets, Algorithm avoids both the “duplicate data” and the “circular dependencies” problems. However, since |C| = n(n−1), it means that there are (n(n−1))! possible ways of sorting C, and thus, \( \tilde{e} \times (n(n-1)!) \) different scheduling possibilities. Thus, a brute force algorithm to determine the best schedule would have a \( O(n!) \) cost.

If we focus on a single time step t, then the scheduling problem can be restated as how to choose the best
permutation of $C$. We can represent this decision problem using a permutation tree as is depicted in Figure 2. The weight of the edges in this permutation tree correspond to the negative amount that choosing each edge contributes to $\hat{M}(t)$. Choosing the best scheduling algorithm is the same as finding the shortest path between vertices $v$ and $d$ in the permutation tree. The Bellman-Ford algorithm can find the shortest path with cost $O(|E| \cdot |V|)$ where $|E|$ and $|V|$ respectively represent the number of edges and vertices in the permutation tree. However, in our permutation tree the number of edges and vertices are both $(n(n-1))!$, which makes finding the optimal schedule for even the simplified problem computationally exorbitantly expensive, even for small number of nodes $n$. Hence, we consider the general problem described in $\mathcal{V}, \mathcal{A}$ to be also intractable.

VI. HEURISTIC SCHEDULING ALGORITHMS

In this section we investigate several heuristics for scheduling the in-network redundancy generation. We split the scheduling problem into two parts, following the strategy presented in Algorithm 1.

The heuristics do not require Assumption 1 thus allowing the source node to send different amounts of data to each storage node. We however still rely on Assumption 2, which allows us to model the decision problem with a sorting algorithm, as previously outlined in Algorithm 1. Thus, the overall scheduling problem is decomposed into the following two decisions: (i) How does the source node schedule its uploads? (ii) How are redundancy generation triplets sorted?

A. Scheduling traffic from the source

Recall that generating redundancy directly from the source node involves less bandwidth than doing it with in-network techniques (Remark 3). Thus, a good source traffic scheduling should aim at maximizing the source’s upload capacity utilization. Furthermore, the schedule must also try to ensure that the source injected data can be further used for the in-network redundancy generation.

Given a $(n,k)$ HSRC code, where $n = 2^k - 1$, any subset of $k$ linearly independent encoded fragments forms a basis, denoted by $B$ (see Example 2 for an illustration). Let $B$ be the set of all the possible bases $B$. Since each storage node stores one redundant fragment, we use $\hat{B}(t)$ to represent all the basis of $B$ whose corresponding storage nodes are available at a time step $t$ (and likewise, refer to each combination of such nodes as an available basis):

$\hat{B}(t) = \{ B \in B \mid a(i,t) = 1, \forall i \in B \}$.

From the set of available basis, $\hat{B}(t)$, the source node selects one basis $B$ and uploads some data to each node $i \in B$. The amount of data the source uploads to each node $i \in B$ is set to guarantee that at the end of time step $t$, all these nodes have received the same amount of data, $\theta(i,t) = \theta(j,t), \forall i,j \in B$. To satisfy this while maximizing the upload capacity$^4$ utilization of the source, the source needs to send to each node $i \in B$ an amount of data equal to

$$\frac{1}{|B|} \left( u(s,t) + \sum_{j \in B} \theta(j,t-1) - \theta(i,t) \right).$$

We considered the following policies for the source node to select a specific $B$ among all the available basis, $\hat{B}(t)$:

- Random: $B$ is randomly selected from $\hat{B}(t)$. Repeating this procedure for several time steps is expected to ensure that all nodes receive approximately the same amount of data from the source.

$^4$We assume that the upload capacity of the source is less than the download capacity of the basis nodes.
**Minimum Data**: The source selects the basis $B$ that on an average has received less redundant data. It means that $B$ is the basis that minimizes $\frac{1}{|B|} \sum_{i \in B} \theta(i, t)$. This policy tries to homogenize the amount of data all nodes receive.

**Maximum Data**: The source selects the basis $B$ that on an average has received more redundant data. It means that $B$ is the basis that maximizes $\frac{1}{|B|} \sum_{i \in B} \theta(i, t)$. This policy tries to have a basis of nodes with enough data to allow the in-network redundancy generation for the entire data object even when the source may not be available.

**No Basis**: The source does not consider any basis and instead uploads data to all the online nodes. The upload bandwidth of the source is also distributed to guarantee that, after time step $t$, all online nodes have received the same amount of data.

**B. Sorting the redundancy generation triplets**

We explore the following sorting heuristics to answer the second question:

- **Random**: Repair triplets are randomly sorted. This policy tries to uniformly distribute the utilization of network resources to maximize the amount of in-network generated data.

- **Minimum Data**: The list of available triplets are sorted in ascending order according to the amount of data $\theta(k, t)$ the destination node $k$ has received. This policy tries to prioritize the redundancy generation in those nodes that have received less redundant data.

- **Maximum Data**: Similarly to the Minimum Data policy, however, triplets are sorted in descending order. This policy tries to maximize the amount of data some specific subset of nodes receive, to allow them to sustain the redundancy generation process even when the source is not available.

- **Maximum Flow**: The triplets are sorted in descending order according to the amount of redundant data these nodes can help generate. Note that the amount of data a triplet $c$ can generate at each time step $t$, where $c = (i, j) \rightarrow k$, is given by:

$$\min(u(i, t), u(j, t), d(k, t), \theta(i, t) - \theta(k, t), \theta(j, t) - \theta(k, t))$$

This policy tries to maximize the amount of new redundancy generated per time step.

**VII. EXPERIMENTAL RESULTS**

He have proposed four different policies for the source traffic scheduling problem and four policies for the triplets sorting problem. However, due to space limitations we report for each case the two best policies (in terms of achieved throughput). At the source, the random and minimum data policies consistently outperform the others, and at the storage nodes, the maximum flow and minimum data sorting policies for the triplets likewise outperform the others. We will refer to each of the combinations as follows:

| Policy Name | Source Policy | In-Network Policy |
|-------------|---------------|------------------|
| RndFlw      | random        | maximum flow     |
| RndDta      | random        | minimum data    |
| MinFlw      | minimum data  | maximum flow     |
| MinDta      | minimum data  | minimum data    |

It is interesting to note that the minimum data policy obtains good storage throughput in both cases, which leads us to infer that in general, prioritizing redundancy generation in those nodes that have received less data is a good strategy to maximize the throughput of the backup process.

**A. Setting**

We considered a $\langle n = 7, k = 3 \rangle$-HSRC code, which is a code that can achieve a static data resiliency similar to a 3-way replication, but requiring only a redundancy factor of $7/3 \approx 2.33$. Using this erasure code we simulated various backup processes with different node (un)availability patterns for a fixed number of time steps $\bar{t}$. In all the simulated cases we consider three different metrics:

(i) The maximum amount of data that can be stored in $\bar{t}$ time steps, $\bar{M}(\bar{t})$.

(ii) The amount of data the source node uploads per unit of useful data backed up,

$$\frac{1}{\bar{M}(\bar{t})} \sum_{i=0}^{\bar{t}} \sum_{i=1}^{n} f(s, i, t).$$

(iii) The total traffic generated per unit of useful data stored, $T(f, \bar{t})/\bar{M}(\bar{t})$.

We evaluate the three metrics for a system using an in-network redundancy generation algorithm and we compare our results with a system using the naive erasure coding backup process, where the source uploads all the data directly to each storage node. Our results depict the savings and gains, in percentage, of using an in-network redundancy algorithm with respect to the naive approach.

Regarding the (un)availability patterns of nodes and their bandwidth constraints we consider two possible cases:

- A P2P-like environment where nodes have an upload bandwidth uniformly distributed between 20Kbps and 200Kbps, and an asymmetric download bandwidth equal to four times their upload bandwidth. Nodes in this category follow two different availability traces from real decentralized application: (i) traces from users of an instant messaging (IM) service [6], (ii) traces

\[\text{Node } k \text{ is the destination of a triplet } c, c = (i, j) \rightarrow k.\]
from P2P nodes in the aMule KAD DHT overlay [14]. In both cases we filter the nodes that on average stay online more than 4, 6 and 12 daily hours, obtaining different mean availability scenarios. Finally the time step duration is set to \( \tau = 1 \) hour and we obtain the results by averaging the results of 500 backup processes of \( \bar{t} = 120 \) time steps each (5 days).

- A datacenter-like scenario where nodes have a symmetric upload/download bandwidth equal to 1Gbps. These nodes have availability sessions that follow an exponential distribution with rate \( \lambda_{on} = (2h \times a)^{-1} \) and unavailability sessions that follow an exponential distribution with rate \( \lambda_{off} = (2h \times (1 - a))^{-1} \), where \( a \) is the average online availability. We simulate 4 average online availabilities, namely, 10\%, 30\%, 60\% and 100\%. In this case the time step duration is set to \( \tau = 5 \) minutes and we obtain the results by averaging the results of 500 backup processes of \( \bar{t} = 144 \) time steps each (12 hours).

\[ \begin{align*}
\text{Scheduling Policy} & >4h \\
& >6h \\
& >12h
\end{align*} \]

B. Results

In figures 3a, 3b, 3c and 3d, 3e, 3f we show results based on F2F and P2P scenarios (using IM and KAD traces respectively). Figures 3a and 3d show how the storage throughput increases with nodes being more available on an average. This is due to the constraint in eq. (5) requiring...
redundancy generation triplets to be symmetric, which requires the three involved nodes in each triplet to be available simultaneously. The higher the online availability, the higher the chances to find online three nodes from a triplet. Further, we observe that the \textit{RndFlw} policy achieves significantly better results in comparison to other policies.

As noted previously (in Remark 3), the total traffic required for in-network redundancy generation is twice that needed by the traditional process. Figures 3b and 3c confirm this observation. We additionally note that the increase in traffic is approximately the same or even less than the increase in storage throughput even for low availability (> 4h) scenarios. Thus the in-network redundancy generation scales well by achieving a better utilization of the available network resources than the classical storage process.

In the traditional approach, the source needs to upload \(7/3 \approx 2.33\) times the size of the actual data to be stored; \(4/7 \approx 57\%\) of this data is redundant. Figures 3c and 3f show the reduction of data upload at the source. In the best case (> 12h traces and \textit{RndFlw} policy) our approach reduces the source’s load by 40% (out of a possible 57%), yielding 40-60% increase in storage throughput (figures 3a and 3d).

Figures 3c, 3h, 3i show results for the datacenter-like scenario. When node availabilities are high, we know that significant throughput gains can be achieved (upto 140%). It is interesting to see how in the case of low node availability (10%) the total amount of data that can be stored with the in-network redundancy generation technique is less than using the traditional storage processes. This is an artefact of two shortcomings - one with our scheduling algorithm, and one with the synthetic trace we generated.

Finding three available nodes simultaneously is unlikely when overall availability is low. To solve this problem, we would need to look at more sophisticated in-network redundancy generation strategies not subjected to the symmetric constraint (defined in eq. 5), so that nodes can forward and store partially-generated data. However, the scheduling problem will be much more complicated, and is beyond the reach of this first work. Furthermore, in real traces, nodes will have correlation (e.g., based on batch jobs), which are missing in the synthetic traces, and such correlations can be leveraged in practice. Exploring both these aspects will be part of our future work.

VIII. Conclusions

In this work we propose and explore how storage nodes can collaborate among themselves to generate erasure encoded redundancy by leveraging novel erasure codes’ self-repairing property thus reducing a source node’s load, and improving overall throughput for data backup. We demonstrate that finding an optimal schedule is computationally prohibitive (even under simplifying assumptions), but experiments based on heuristics yield significant gain in storage throughput under diverse settings, proving their practicality.

ACKNOWLEDGMENTS

The work of L. Pamics-Juarez and F. Oggier for this paper is supported by the Singapore National Research Foundation under Research Grant NRF-CRP2-2007-03. A. Datta’s work in this paper has been funded in part by NTU/MoEs AcRF Tier-1 Grant number RG 29/09. We also thank M. Dell-Amico and P. Michiardi for providing the availability traces of the IM application.

REFERENCES

[1] S. Ghemawat, H. Gobioff, and S. Leung, “The google file system,” in Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 2003.

[2] D. Borthakur, “The apache hadoop distributed file system,” http://hadoop.apache.org/hdfs/, The Apache Software Foundation, 2011.

[3] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Voshall, and W. Vogels, “Dynamo: Amazons highly available key-value store,” in Proceedings of Symposium on Operating Systems Principles (SOSP), 2007.

[4] A. Lakshman and P. Malik, “The apache cassandra project,” http://cassandra.apache.org/, The Apache Software Foundation, 2011.

[5] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadhi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, “Oceanstore: An architecture for global-scale persistent storage,” in The 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2000.

[6] R. Sharma, A. Datta, M. Dell Amico, and P. Michiardi, “An empirical study of availability in friend-to-friend storage systems,” in Proceedings of the 11th Intl. Conference on Peer-to-Peer Computing (P2P), 2011.

[7] J. Li and F. Babek, “F2F: reliable storage in open networks,” in Proceedings of the 5th Intl. Workshop on Peer-To-Peer Systems (IPTPS), 2006.

[8] I. Reed and G. Solomon, “Polynomial codes over certain finite fields,” Journal of the Society for Industrial and Applied Mathematics, vol. 8, no. 2, pp. 300–304, 1960.

[9] R. Rodrigues and B. Liskov, “High availability in dhts: Erasure coding vs. replication,” in Proceedings of the 4th Intl. Workshop on Peer-To-Peer Systems (IPTPS), 2005.

[10] W. K. Lin, D. M. Chiu, and Y. B. Lee, “Erasure code replication revisited,” in Proceedings of the 4th Intl. Conference on Peer-to-Peer Computing (P2P), 2004.

[11] H. Weatherspoon and J. D. Kubiatowicz, “Erasure coding vs. replication: A quantitative comparison,” in Proceedings of the 1st Intl. Workshop on Peer-To-Peer Systems (IPTPS), 2002.

[12] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, “Network coding for distributed storage systems,” IEEE Transactions on Information Theory, vol. 56, no. 9, 2010.
[13] F. Oggier and A. Datta, “Self-repairing homomorphic codes for distributed storage systems,” in The 30th IEEE International Conference on Computer Communications (INFOCOM), 2011.

[14] M. Steiner, T. En-Najjary, and E. Biersack, “A global view of kad,” in Proceedings of the 7th ACM SIGCOMM conference on Internet Measurement (IMC), 2007.

[15] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan, “Does erasure coding have a role to play in my data center?” Microsoft Research, Tech. Rep. MSR-TR-2010-52, 2010.

[16] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, “Availability in globally distributed storage systems,” in Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI), 2010.

[17] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, “Diskreduce: Raid for data-intensive scalable computing,” in Proceedings of the 4th Annual Workshop on Petascale Data Storage (PDSW), 2009.

[18] S. Pawar, N. Noorshams, S. E. Rouayheb, and K. Ramchandran, “Dress codes for the storage cloud: Simple randomized constuctions.”

[19] A. Dimakis, V. Prabhakaran, and K. Ramchandran, “Decentralized erasure codes for distributed networked storage,” IEEE/ACM Transactions on Networking, vol. 14, 2006.

[20] ——, “Ubiquitous access to distributed data in large-scale sensor networks through decentralized erasure codes,” in Proceedings of the 4th Intl. Symposium on Information Processing in Sensor Networks (IPSIN), 2005.

[21] ——, “Distributed fountain codes for networked storage,” in Proceedings of the Intl. Conference on Acoustics, Speech and Signal Processing (ICASSP), 2006.

[22] R. Ahlswede, N. Cai, S. R. Li, and R. W. Yeung, “Network information flow,” IEEE Transactions on Information Theory.

[23] J. S. Plank, S. Simmerman, and C. D. Schuman, “Jerasure: A library in C/C++ facilitating erasure coding for storage applications,” University of Tennessee, Tech. Rep. CS-08-627, 2008.