Applying Erasure Codes for Fault Tolerance in Cloud-RAID

Maxim Schnjakin, Tobias Metzke, Christoph Meinel
Hasso Plattner Institute
Potsdam University, Germany
Prof.Dr-Helmert-Str. 2-3, 14482 Potsdam, Germany
{maxim.schnjakin, office-meinel}@hpi.uni-potsdam.de
tobias.metzke@student.hpi.uni-potsdam.de

***Abstract*—Public cloud storage services enable organizations to manage data with low operational expenses. However, the benefits come along with challenges and open issues such as security and reliability. In our work, we presented a system that improves availability, confidentiality and reliability of data stored in the cloud. To achieve this objective, we encrypt user's data and make use of erasure codes to stripe data across cloud storage providers.**

In this paper we focus on the need to identify an algorithm for encoding and reassembling the data from the clouds. Erasure codes have been introduces more than three decades ago. Due to new technology trends and powerful hardware, new codes as well as improvements on classic codes have been developed recently. Therefore, we provide an overview of the current state of erasure codes. Further, we introduce the relevant codes in detail and compare them on the basis of identified criteria that are relevant to their application in a cloud context. Furthermore, we take a look at the current open source libraries, that support the discussed algorithms. The comparative study will help us to identity the best algorithm for our Cloud-RAID system.

I. Introduction

The usage of computing resources as pay-as-you-go model enables service users to convert fixed IT cost into a variable cost based on actual consumption. Therefore, numerous researchers argue for the benefits of cloud computing focusing on the economic value [1], [2].

However, despite of the non-contentious financial advantages cloud computing raises questions about privacy, security and reliability. Among available cloud offerings, storage services reveal an increasing level of market competition. According to iSuppli [3] global cloud storage revenue is set to rise to $5 billion in 2013, up from $1.6 billion in 2009. One reason is the ever increasing amount of data which is supposed to outpace the growth of storage capacity.

For a customer (service) to depend solely on one cloud storage provider has its limitations and risks. In general, vendors do not provide far reaching security guarantees regarding the data retention [4]. Placement of data in the cloud removes the physical control that a data owner has over data. So there is a risk that a service provider might share corporate data with a marketing company or use the data in a way the client never intended. Further, customers of a particular provider might experience vendor lock-in. In the context of cloud computing, it is a risk for a customer to become dependent on a provider for its services.

In our previous work [5], [6] and [7] we presented an approach that deals with the mentioned problems by separating data into unrecognizable slices, that are distributed to different providers. It is important to note, that only a subset of the providers needs to be available in order to reconstruct the original data.

To achieve this goal, we aim to utilize erasure codes. In order to find the best algorithm for our application, we conduct a comparative analysis on relevant algorithms. Erasure codes have been around since the 1960s already and originate in information theory rather than engineering [8]. In general, the algorithms have not been widely applied up to some years ago. This is mainly due to the fact that the first algorithms have been based on Galois field operations which have not been well supported by CPUs and therefore were not applicable for large data sets. However, recent hardware and software development [9] enabled the support for fast calculations of required operations and paved the way for the usage of erasure codes.

In recent years, the number of new erasure coding techniques has been rising fast. New codes have been created out of the need for simpler erasure codes based on CPU-supported operations in earlier years. Furthermore, new erasure codes adapted to the characteristics of new technology trends like *Multicore and Multinode Systems, Big Data, and Cloud Computing*. While some codes focus more on better encoding and decoding performance compared to the first erasure codes, others aim to reduce the repair traffic. A comprehensive overview of the existing erasure codes and their characteristics is needed. This paper provides an overview of the status quo of relevant erasure codes and helps us by the identification of an appropriate algorithm for our system.

The contributions of this paper comprise:

- An overview of *new* and *classic* erasure codes. In general, erasure algorithms can be distinguished by the intention to reduce the so-called storage overhead and by the intention to reduce the repair traffic. This paper

provides an introduction to a number of selected codes.
- A direct comparison of relevant erasure algorithms. We have identified several criteria to enable a direct comparison between listed codes (with regard to the requirements of our system).
- An overview of available erasure implementations. Several frameworks for erasure coding exist nowadays; this paper introduces those that are widely used and freely available while also highlighting the implemented erasure codes.

The remainder of this paper is structured as follows: in sections III and IV we introduce the general architecture of our application and describe the creteria needed for the comparison of the presented algorithms. Section V afterwards provides an overview of the selected erasure codes and describes their characteristics. Then, in Section VI we present a list of open source erasure code frameworks. Finally, in Section VIII we conclude with a discussion of the implications of the conducted survey to the Cloud-RAID system.

II. Related Work

Erasure codes have been evaluated on a variety of metrics and criteria, such as the CPU impact of encoding and decoding [10], [11]. The coding and decoding performance of different erasure codes can vary significantly. Concerning the performance of the codes, the overview presented in this paper does not provide any formulas, experiment results or calculations. Providing generic formulas is rather complex, since the performance of the codes is highly dependent on the specific configurations of the codes, the libraries used for the experiments, and the specific scenarios for encoding and decoding (e.g. the number of failed chunks for decoding). Furthermore, several performance evaluations have been already conducted [12], [13], [14] for all the codes presented in this paper.

However, in our specific use case the performance of individual algorithms plays a role that is certainly important, but not decisive: in our tests presented in [6] and [5] we observed, that the average performance overhead caused by data encoding is less than 2% of the amount of time for data transfer to a cloud provider. With this, encoding is dominated by the transmission times and can be neglected. Here, the storage overhead and the I/O traffic are more important, as the values are associated with costs. There has been some work lowering I/O costs in erasure-based systems. As stated in chapter V, Rotated RD codes have been designed to lower I/O costs on recovery. However, in this paper we do not intend to improve a specific algorithm, but to undertake a comparative analysis on relevant algorithms.

III. Motivation

The ground of our approach is to find a balance between benefiting from the cloud's nature of pay-per-use and ensuring the availability of the company's data. As mentioned above, the basic idea is not to depend on solely one storage provider but to spread the data across multiple providers using redundancy to tolerate possible failures. The approach is similar to a service-oriented version of RAID. While RAID manages sector redundancy dynamically across harddrives, our approach manages file distribution across cloud storage providers. We carry the principle of the RAID-technology to cloud infrastructure. In order to achieve our goal we foster the usage of erasure coding techniques. The system has a number of core components that contain the logic and management layers required to encapsulate the functionality of different storage providers. Our architecture (see Figure 1) includes the following main components:

- **User Interface Module.** The interface presents the user a cohesive view on his data and available features. Here users can manage their data and specify requirements regarding the data retention (quality of service parameters).
- **Resource Management Module.** This system component is responsible for intelligent deployment of data based on users' requirements. The component is supported by:
  - a registry and matching service: assigns storage repositories based on users requirements (for example physical location of the service, costs and performance expectations). Monitors the performance of participating providers and ensures that they are meeting the agreed SLAs
  - a resource management service: takes operational decisions regarding the content storage
  - a task scheduler service: has the ability to schedule the launch of operations at peak-off hours or after specified time intervals.
- **Data Management Module.** This component handles data management on behalf of the resource management module and is mainly supported by:
  - a data encoding service: this component is responsible for striping and encoding of user's content
  - a data distribution service: spreads the encoded data packages across multiple providers. Since each storage service is only accessible through a unique API, the service utilizes storage "service-connectors", which provide an abstraction layer for the communication to storage repositories
  - a security service: manages the security functionality based on a user's requirements (encryption, secret key management).

With this, we consider a distributed multiple cloud storage setting from user's perspective, such as we stripe data over multiple clouds. Interested readers will find more background information in our previous work [5],[6]. In the following, we will analyze existing erasure codes to

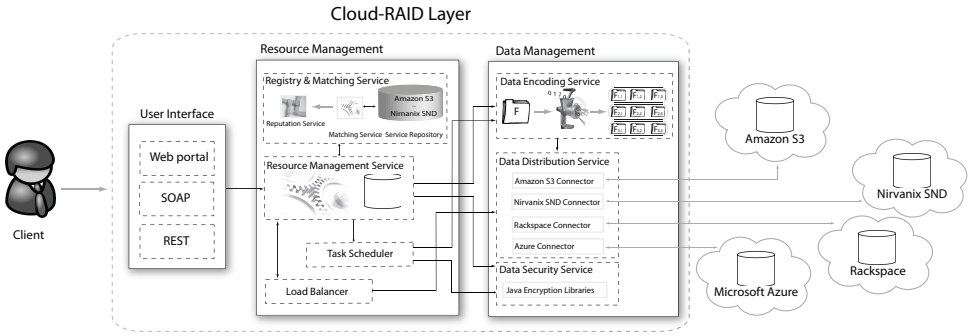


Figure 1. A general architecture of the Cloud-RAID system.

decide on an appropriate algorithm to encode and repair the outbound data.

IV. Comparison Criteria

Existing erasure codes follow different intentions. While some strive to increase the encoding/decoding performance, others focus on the reduction of recovery costs. In order to provide a comprehensive overview of the existing codes, a list of objective criteria is needed. Based on our specific use case, we have identified the following criteria for a direct comparison between different algorithms:

- **The degree of fault tolerance.** In general, the number of possible coding chunks represents the fault tolerance capability of an erasure algorithm. In our work, we use the parameter m to define the total number of chunks (clouds) that can be lost without losing data. If the value of the attribute is not limited or predefined by an erasure code, it is marked as ∞.
- **The storage overhead.** Due to the nature of erasure codes, encoding is associated with a certain data overhead. This overhead is caused by redundant m packages, which have to be created for data recovery. The storage overhead can be expressed as follows: $\frac{coded\ data}{original\ data}$.
- **Data recovery preferences.** When it comes to data recovery, some algorithms define access patterns to reduce recovery I/O and bandwidth. The according recovery plan determines which data objects have to be accessed for efficient data reconstruction. However, in our use case, the selection of required chunks is based on the performance capabilities of clouds (e.g. response time, throughput etc.). Therefore, we favor algorithms which do no distinguish between individual (encoded) data packages when reassembling the data.
- **The repair traffic.** In compliance with [15] we define the repair traffic as the amount of outbound data being read from clouds in the event of a single-cloud failure. With this, the repair traffic represents the number of accesses required to reconstruct the original data. We denote the value as the number of *data fragments* that need to be read. In general, we are interested in fetching the minimum number of chunks for data recovery. This is due to the fact, that in the pay-per-use cloud models it is not economical to read all data packages from all clouds to recover original data.
- **The recovery read access mode.** The recovery of data is done through accessing specific blocks within encoded data packages (chunks). Some algorithms require read access to entire chunks or use all encoded chunks when reassembling the data. Other codes require *partial read accesses* to encoded data. Indeed, partial reads are not allowed in clouds. The parameter denotes if partial reads are required by an individual algorithm or not.

V. Erasure Codes

This section will introduce several erasure codes and their characteristics based on the criteria defined in section IV. Will will start by introducing the basics of erasure codes before presenting five specific codes in sections V-B to V-G in detail. This list is based on the work of Plank et al. in [12] extended by two recent erasure codes.

A. Basic Erasure Coding

Given a file object, erasure coding divides original data into k equally sized chunks that hold $data$ and m chunks that hold $coding$ information. The coding chunks are calculated from the data chunks using an erasure algorithm. Each chunk (data and coding) comprises a number of *words* of length w. All chunks are thereby equivalently structured, meaning that the word size is equally fixed for all data and coding chunks.

Encoding then is an operation that combines all data elements that are horizontally aligned (share the same offset in the data chunks, often referred to as a *stripe*) and encodes them into a coding element (fragment) of the same size with the same offset on a coding chunk.

The operations used for encoding can differ between bitwise exclusive-or (XOR) and Galois field arithmetic

$(GF(2^w))$. While XOR operations can be executed fast on CPUs, $GF(2^w)$ multiplications traditionally perform worse. However, new libraries like the one provided by Plank et al. [9] offer support for fast computations of these operations with modern CPUs as well.

In summary, erasure algorithms map a data object broken into k equal-size original chunks onto a larger set on n chunks of the same size in such a way, that the original chunks can be recovered from any $n - k = m$ chunks. Erasure codes that only need *any* arbitrary k out of n chunks for the recovery of original data are called *maximum distance separable* codes (MDS codes) [16]. Similar to Plank et al. [12], we are only interested in MDS codes. Furthermore, we focus only on *horizontal codes* where chunks hold either original data or parity data. The so called *vertical codes* are not considered in our work, as they allow chunks to hold both kinds of data. This means vertical codes operate on all encoded chunks to restore the original data, whereas horizontal codes require the minimum number of k arbitrary chunks to achieve the same goal.

B. Reed-Solomon

Reed-Solomon codes [8] (RS codes) have the longest history among erasure codes. They originate in information theory and operate on w-bit words. The used word size w is only limited by the total number of chunks n which is not allowed to be greater than 2^w+1. Except for this restriction, the word size can be freely determined by a user. Typically, the chosen size corresponds to machine word boundaries (8, 16, 32, 64) for good performance and $w = 8$ for best performance.

Further, the total number of encoded chunks can be defined arbitrarily. Each RS configuration is denoted as (k,m) and increases the storage costs by a factor of $\frac{m}{k}$. Note that m=1 represents a simple replication, and RAID level 6 can be described by (k=4, m=2). In the example, the algorithm calculates n=6 chunks, any k=4 chunks of which are sufficient to reconstruct the object, resulting in a total overhead of $\frac{2}{4} = 50\%$.

RS codes treat each word of length w as a number between 0 and 2^w-1 and operate on these numbers with Galois field arithmetic $(GF(2^w))$. The algebraic structure defines addition, multiplication and division on words and produces a closed and well-behaved system [16]. Addition in $GF(2^w)$ is equivalent to XOR operation, multiplication however is far more complex. It is usually implemented with multiplication tables or discrete logarithm tables and considered expensive in terms of computation time. However, as mentioned in Section V-A, new libraries enable the fast computation of these operations nowadays.

Encoding with RS codes requires the multiplication of a *Generator matrix* G^T (derived from a *Vandermonde matrix*) with individual data words of a single *stripe*. The operation creates a *codeword* which comprises k data elements and m coding elements. The latter are stored on dedicated coding chunks. The procedure has to be executed for each stripe. Figure 2 illustrates the encoding process of a single stripe.

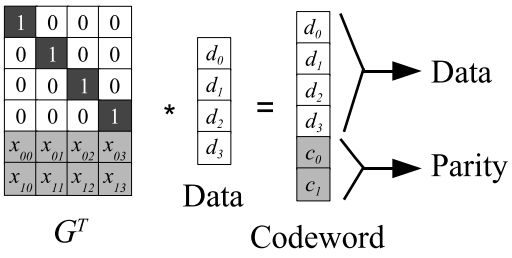


Figure 2. Reed-Solomon Encoding with k=4 and m=2.

However, data recovery is comparable to the encoding process and is realised by a stripe-by-stripe recalculation of existing chunks. This requires the elimination of those strips in the G^T matrix that correspond to the lost data chunk. Then, the resulting matrix has to be inverted and multiplied by k arbitrarily chosen data fragments that have not been lost.

The amount of data needed for the recovery of a single chunk always equals $k \times chunk\ size$, since k out of n chunks are needed to reconstuct the data. In terms of data elements that have to be read, RS codes always need to access $k \times \lceil \frac{chunk\ size}{word\ size} \rceil$ elements.

The use of a Vandermonde matrix for the construction of G^T ensures that the inversion of the latter is always possible. The inversion, however, is a rather expensive operation in terms of computation time.

Overall, the Reed-Solomon codes are the most flexible erasure codes in terms of word size and coding chunks, with a rather weak performance compared to other codes introduced in the following sections. However, they describe the most generic family of codes that can be used for a variety of applications and have been employed in many fields such as deep-space communication, consumer electronics (e.g. CDs, DVDs, Blu-ray discs), in data transmission technologies (e.g. DSL) and in RAID-6 storage systems.

C. Cauchy-Reed-Solomon

Cauchy Reed-Solomon (CRS) codes [17] differ from RS algorithms in two ways: i) The G^T matrix is created by the usage of Cauchy matrices instead of Vandermonde matrices, and ii) they employ XOR operations instead of expensive $GF(2^w)$ multiplications. Similar to RS codes, the total number of coding chunks (defined by the parameter m) can be chosen freely. The produced storage overhead is equivalent to RS codes.

To eliminate multiplications, CRS codes transform the G^T matrix from a $n \times k$ matrix of w-bit words to a $wn \times wk$ matrix of bits. This new matrix is then multiplied not by single data words of w-bit size but *strips* of w data

fragments. For performance reasons, the size of a fragment should be a multiple of the machine's word size. Each of the k data chunks now consists of several strips, each strip containing w fragments. The w is not constrained to the machine's word size and can be defined freely (as long as $n \leq 2^w$). Encoding and decoding still happens stripe-by-stripe - whereby each stripe consists now not of single data words but strips. Note, the modification of the matrix is also possible for Vandermonde matrices.

Encoding needs only XOR operations in order to compute coding data. A coding fragment is created as the XORed result of all data fragments that have a one-bit in the coding chunk's line of G^T. Figure 3 illustrates how the last coding fragment is created by XORing all data fragments that have a one bit in the last line of G^T.

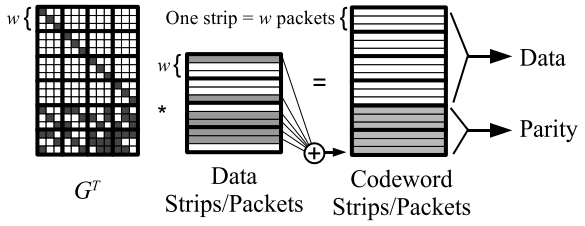


Figure 3. Cauchy Reed-Solomon Encoding with n=4 and m=2.

The recovery of lost data with CRS codes is handled analogously to RS codes: all rows of G^T corresponding to failed chunks have to be deleted, then the matrix needs to be inverted and multiplied by k existing data chunks. With this, the recovery process requires the same data fragments as RS codes. The total number of read operations on fragment level can be smaller due to possibly larger CRS fragment sizes (in comparison to RS word sizes).

Despite the potential performance gain through the elimination of $GF(2^w)$ multiplications, the performance of CRS codes is highly dependent on the number of ones in G^T. The more one-bits the generator matrix consists of, the more data fragments have to be XORed, which results in a weaker encoding performance. Research on the creation of optimal matrices for CRS codes [18] yielded matrices with fewer ones than the original Cauchy matrices. Some libraries (e.g the *Jerasure* library, see Section VI) optimize these matrices even further to achieve better performance for en-/decoding.

D. EVEN/ODD and RDP

EVEN/ODD [19] and RDP[20] are special-purpose RAID-6 erasure codes. Therefore, the number of coding chunks always equals to $m = 2$. The coding chunks are called parity chunks *P* and *Q*. The storage overhead of both codes is only dependent on the number of data chunks, since m is fixed. Therefore, the caused storage overhead can be calculated as follows: $\frac{2}{k}$.

Analogously to CRS codes, EVEN/ODD and RDP operate on strips with w packets. While EVEN/ODD constraints w so that $n + 1 \leq w$, RDP expects $n \leq w$. Both algorithms define $w + 1$ to be a prime number.

The P-drive is calculated by XORing all strips of one stripe in order to create a parity strip. The Q-drive, however, is constructed differently with specific parity equations for both codes (see Figure 4). While RDP reuses the P-drive to calculate the Q-drive, EVEN/ODD employs a special pattern with an intermediate result.

When restoring lost data, EVEN/ODD and RDP work with one parity drive (restoring one failed data chunk) or a combination of parity drives and special patterns similar to the ones shown in Figure 4. The amount of data needed for the recovery is comparable to CRS, since RDP and EVEN/ODD work with similar strip and fragment definitions. Further, the algorithms belong to MDS codes family, which means that the recovery process requires any arbitrary k out of $k + 2$ encoded data chunks.

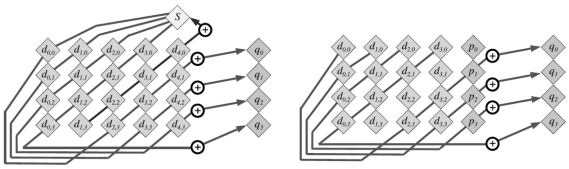


Figure 4. The Q-drive calculation patterns of EVEN/ODD (left) and RDP (right).

According to [19] the total number of operations needed to encode and decode data is provably lower for RDP and EVEN/ODD compared to standard RS and CRS codes. However, in contrast to RS and CRS codes EVEN/ODD and RDP are under patent protection, which means that the implementation of the algorithms is not available to the public (see Section VI).

E. Minimal Density

In case of RAID-6 specifications, the used G^T bit-matrix (see CRS codes) is quite constrained. The number of coding chunks is fixed to $m = 2$, which results in a $w(k + 2) \times wk$ matrix. The first wn rows of this matrix form a fixed identity matrix and represent the original data in the resulting codeword when encoding this data (see Figure 3).

The next w rows comprise the k identity matrices that build the P-drive and are therefore fixed either. The composition of the last w rows of G^T is the only flexible part for a RAID-6 specification. In case of $k \leq w$, the minimal number of ones in these rows is given by $kw + k - 1$ [21]. Minimal Density (MD) codes are using matrices that achieve this lower bound for RAID-6 configurations.

MD codes can therefore be seen as an optimized code family of CRS codes for RAID-6 specifications. Thus, MD codes share basic characteristics (e.g. storage overhead) with other RAID-6 codes like EVEN/ODD and RDP.

Three different constructions of MD codes exist, depending on the size of the parameter w:

- **Blaum-Roth** codes for $w + 1$ being a prime number [21],
- **Liberation** codes for w being a prime number [22] and
- **Liber8tion** codes for $w = 8$ [23].

As Plank et al. [12] state, MD codes encode with $k - 1 + \frac{k-1}{2w}$ XOR operations per coding word. Thus, the algorithms perform better with a higher value of the parameter w. For decoding, MD codes require *Code-Specific Hybrid Reconstruction* in order to achieve a near-optimal performance. Overall, as shown in [12], these code perform better than CRS codes for RAID-6 specifications.

F. Rotated Reed-Solomon

Rotated RS codes [13] have been designed to reduce bandwidth and I/O needed for the recovery of lost data. In particular, these codes optimize the performance of degraded reads in single chunk failure scenarios. Since 99.75% of recovery scenarios deal with single chunk failures, as Schroeder et al. [24] discovered, Khan et al. [13] optimized their codes for this particular use case. The authors build on classic RS codes and modify the way of encoding data to allow faster recovery by reusing and therefore accessing less data fragments. Beyond that the algiorithms share all characteristics (namely storage overhead, performance, and configurability) with classic RS codes.

However, Rotated RS codes are not generally MDS codes. In [13] Khan et al. present several constraints that are necessary for these codes to belong to the family of MDS codes. These limitations comprise $m \in \{2, 3\}$, $k \leq 36$ and $w \in \{4, 8, 16\}$. Research on general MDS constructions of Rotated RS codes is still ongoing.

As depicted in Figure 5, encoding with rotated RS codes modifies classic RS coding in two ways: i) single-row stripes of w-bit sized data words are transformed into r multi-row stripes of bit-words (r and w may be different), and ii) XORing the encoded data fragments (symbols) is not done within one row but across adjacent rows.

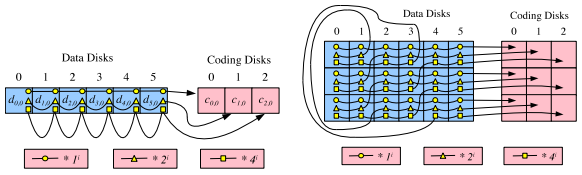


Figure 5. Encoding of classic RS (left) and Rotated RS (right) codes with $n = 6$ and $m = 3$

Except these two modifications, the encoding operation shares all characteristics with classic RS encoding (e.g. performance, storage overhead). As shown in Figure 6, the decoding operation benefits from the rotated encoding in terms of read accesses. With this, the recovery of single chunk failures gains in performance. The number of data symbols that have to be accessed is reduced by a particular pattern of XORing between adjacent rows. In the provided example, the behavior leads to three different recovery paths through the original data for each data bit. The procedure would correspond to one path in classic RS codes (repeated three times).

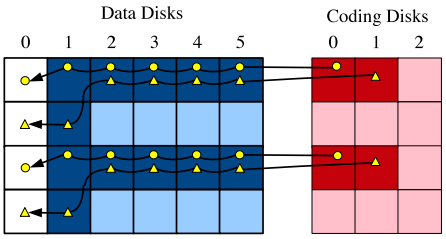


Figure 6. Recovery of first data chunk with Rotated RS codes with $n = 6$ and $m = 3$

In [13] Khan et al. state that the reconstruction of a whole chunk, when r is even, requires exactly $\frac{r}{2}(k + \lceil \frac{k}{m} \rceil)$ data fragments to be read, in comparison to rk data fragments that RS codes need to access. In the example provided in Figure 6, Rotated RS codes need to read 16 fragmetns, while RS codes would need to access 24. Note, the Rotated RS codes are the only codes presented so far that perform partial reads on encoded chunks to improve the performance of the algorithm. The improvement is based on the ability to access chunks more specifically data objects on bit-level without reading the remaining bits of the proper data word. However, this might not be applicable in a context of cloud storage, where partial reads (or writes) are not allowed.

Concerning the number of reads, Rotated RS access seven chunks in the recovery scenario presented in Figure 6 and therefore perform more accesses than classic RS codes. Furthermore, encoded chunks can not be selected arbitrary but provided by the code. This means, the individual chunks can not be considered to be equal.

G. Local Reconstruction

Local Reconstruction Codes (LRC)[14] have been created to reduce the bandwidth in I/O for data recovery processes. The algorithms are build on classic RS codes and introduce *local parities* that serve as additional coding chunks for specific data groups. These parities reduce the average number of data fragments which have to be accessed in the event of a recovery.

Encoding with LRC differs from RS encoding in three ways: i) the original data chunks are divided into a specified number of l groups, ii) coding chunks are divided into *global* and *local* parities, and iii) data chunks are encoded onto all global and one local parity. An exemplary configuration of LRC(6,2,2) is shown in Figure 7.

However, LRC is not MDS in general. Huang et al. [14] prove, that LRC is nonetheless *Maximally Recoverable* (MR). In comparison to MDS, MR code compensates *up*

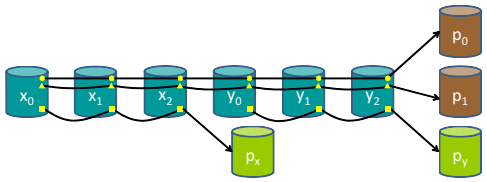
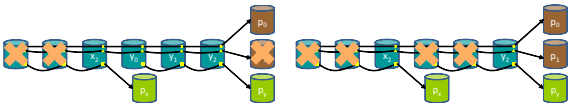


Figure 7. A LRC(6,2,2) configuration with six data nodes, divided into two groups and encoded onto two global parities (p_0 and p_1). Each group is additionaly encoded onto one local parity (p_x and p_y). Thus, this algorithm procudes a total of four coding chunks.

to m chunk losses. More specifically, all those chunks that are *information-theoretically* decodable, but not an *arbitrary* set of m chunks. The recovery of a single (failed) chunk per data group, requires reading of all survived chunks. In case of LRC(6,2,2), a single data chunk can be recovered by reading three other chunks. In comparison to six chunks needed by classic RS codes, which is an improvement of 50%. In general, single chunk failures require $\frac{n}{l}$ data chunks to be read.

Figure 8 illustrates the loss of three and four chunks, both example scenarios are *information-theoretically* decodable and can be repaired by LRC. In fact, the LRC code is able to decode all three failure cases as well as 86% of all four failure scenarios [14]. However, the chunks can not be considered to be equal, since local parity chunks are more important to specific data chunks than to others.



(a) 3 Failures (b) 4 Failures

Figure 8. Recovery of first data chunk with Rotated RS codes with $n = 6$ and $m = 3$

To provide the same level of fault tolerance LRC produces more storage overhead than RS codes. The LRC code creates a storage overhead of $\frac{n+l}{k}$. In our example, the LRC(6,2,2) code corresponds to an RS(9,6) code with $k = 6$ and $m = 3$ (resulting in $n = 9$). The storage overhead results in 1.67-times of the original data size for LRC(6,2,2). In a direct comparison to LRC RS(9,6) code consumes only 1.5-times of original space. However, in contrast to the RS code, the LRC code is able to compensate all three failure scenarios (like RS) as well as 86% of all four failure scenarios. Hence, the algorithm trades storage capacity for reconstruction costs.

VI. Erasure Libraries

Standard RS codes and CRS codes have been around for more than 30 years. Several libraries and frameworks support erasure codes with different optimizations and adjustments. In this section we provide an overview of some popular libraries for erasure coding.

A. Open Source Libraries

Besides proprietary implementations of erasure codes, several open source implementations for different erasure codes are freely available [25], [26], [27], [28]. Most of these projects intent to provide developers with free high quality tools:

Luby: The ICSI lab in Berkeley developed CRS coding in the 1990's [17]. In 1997, the original authors of CRS coding released a C version of their code. It is available from the ICSI's website [25]. All specifications of n, m, w and packet sizes are supported. The used matrices are the original constructions [17] which do not feature a minimized number of ones.

Zfec: The Zfec [26] library enables RS erasure coding since its creation in 2007. The widely used library operates on Vandermonde matrices with $w = 8$. The current version 1.4.24 was published in April 2012. The library is programmable, portable and actively supported by the authors. It includes command-line tools and APIs in C, Python and Haskell.

Jerasure: The C library Jerasure was released in 2007. It supports a wide range of erasure codes, including RS coding, CRS coding, general Generator matrix and bit-matrix coding, and Minimal Density RAID-6 coding [27]. RS coding supports Vandermonde as well as Cauchy matrices, w can be set to 8, 16 or 32. For RAID-6 RS coding, the library features a specific performance optimization. It employs a multiplication by two rather than implementing a general multiplication in $GF(2^w)$ for the calculation of the Q-drive. CRS is implemented with provably optimal encoding matrices for RAID-6 and optimized matrices for higher values of the parameter m. Jerasure supports also the three Minimal Density RAID-6 codes. The implemented Code-Specific Hybrid Reconstruction optimization improves the performance of the bit-matrix codes (especially for decoding). The latest release of the library appeared in August 2008 [27].

Cleversafe: Cleversafe's dispersed storage system was released in an open source version in May 2008 [28]. It is written entirely in Java and supports all API functionality of the proprietary version. The latter is one of the first commercial systems to implement availability beyond RAID-6. The erasure coding part is based on Luby's original CRS implementation [25] with $w = 8$.

This list provides only an excerpt of all available open source libraries for erasure codes. New libraries appear constantly and could be viable alternatives to the mentioned implementations. Furthermore, erasure coding is also

included in frameworks like the Hadoop Distributed File System (HDFS)[1] that is publicly available and extendible.

B. Erasure Codes Coverage

As stated above, there is a wide range of proprietary and open source libraries for erasure codes. However, the libraries implement different algrithms and are optimized for different use cases. Proof of concept frameworks as for example *Luby* implements only CRS codes in a basic manner, whereas current and actively supported libraries like *Jerasure* and *zfec* support a higher number of codes and enable individual configurations.

Table I provides an overview of the codes presented in Section V and the libraries that implement them. Codes implemented by *other* libraries are supported by at least one open source application that is publicly available, including the libraries introduced before.

	Jerasure	zfec	Luby	other
RS	✓	✓	–	✓
CRS	✓	–	✓	✓
EVEN/ODD & RDP	–	–	–	–
Minimal Density	✓	–	–	✓
Rotated RS	–	–	–	✓
Locally Repairable	–	–	–	✓

Table I
OVERVIEW OF ERASURE CODES AND THE LIBRARIES IMPLEMENTING THEM.

The overview clearly shows that classic RS codes are well supported by various libraries. According to [12] zfec outperforms all presented libraries when it comes to classic RS coding. CRS codes are by all open source libraries. However, Jerasure operates on optimized matrices for CRS codes and is one of the most sophisticated libraries for CRS coding. Besides, Jerasure is one of the few implementations for Minimal Density codes. Rotated RS codes as well as LRC are not supported by major libraries so far.

VII. COMPARISON

This section summarizes the characteristics of the presented erasure codes and enables a direct comparison among the examined algorithms. The basis for the comparison is given by the criteria described in Section IV.

Concerning the flexibility of the codes, Table II clearly shows that the RAID-6 specific EVEN/ODD, RDP and Minimal Density codes are more constrained when it comes to the number of coding chunks and therefore the level of fault-tolerance. The storage overhead produced by these codes equals to the more flexible codes (e.g. RS, CRS) for the same level of fault tolerance. Nevertheless, they can only tolerate a failure of two arbitrarily chunks at best.

[1] see http://hadoop.apache.org/docs/stable/file_system_shell.html

The RAID-6 as well as the RS and CRS codes are more flexible than Rotated RS and LRC codes concerning the choice of chunks needed for data recovery. While LRC and Rotated RS somehow predefine the particular chunks for the data recovery to reduce bandwidth and I/Os, all other codes work with arbitrarily selected chunks and consider all chunks to be equal.

To reduce time and cost, the recovery process should consicer as few unique data chunks as possible. However, RS codes have not been designed for efficiency in recovery scenarios but rather for an improvement of the availability of data at low storage overhead. Therefore, the recovery performance of the algorithms is rather weak compared to almost all other codes presented in this paper. The number of fragments needed to be accessed for the recovery of a single chunk can be calculated with the following equation: $n * \lceil \frac{chunk\ size}{word\ size} \rceil$. Other algorithms focus on the reduction of the total number of chunks that have to be accessed. For example, the recovery of a single chunk with LRC requires a total of $\frac{n}{l} * \lceil \frac{chunk\ size}{word\ size} \rceil$ read accesses. Rotated RS codes reduce the number of reads by reusing the accessed fragments and require only $\frac{r}{2}(n + \lceil \frac{n}{m} \rceil) * \lceil \frac{chunk\ size}{word\ size} \rceil$ reads for the recovery of a single chunk (if r is even). The total number of reads with CRS and the RAID-6 codes depends on the number and the type (data or parity) of failed chunks. In general, they use less fragments than RS codes but more than Rotated RS and LRC algorithms.

Specific to Rotated RS codes is the access pattern used for data recovery: these codes rely on the ability to access data (partially) on bit-level to improve encoding performance. As stated in Section V-F, this is not necessarily desirable.

	coding chunks	storage overhead	equal chunks	partial read	open source
RS	∞	$n \setminus k$	✓	–	✓
CRS	∞	$n \setminus k$	✓	–	✓
EVEN/ODD & RDP	2	$n \setminus k$	✓	–	–
Minimal Density	2	$n \setminus k$	✓	–	✓
Rotated RS	∞	$n \setminus k$	–	✓	✓
Local Reconstruction	∞	$n + l \setminus k$	–	–	✓

Table II
OVERVIEW OF ERASURE CODES AND THEIR CHARACTERISTICS.

However, the major findings can be summarized as follows: i) RAID-6 specific codes perform best in encoding and decoding in RAID-6 scenarios, ii) CRS codes perform better in encoding and decoding than RS codes in most cases, iii) Rotated RS codes use significantly less fragments for data recovery than most presented codes for $m \geq 3$, and iv) LRC can achieve up to 50% savings in terms of read accesses and traffic consumption for recovery compared to RS codes. Despite all encoding and decoding experiments, Huang et al.[14] state that in modern data-centered applications that operate on Exabytes of data, the speed of encoding and

decoding can somehow be neglected compared to the time and cost of the transmission of data itself. These statements are consistent with the results of our experiments presented in [6] and [5]

VIII. Conclusion

In this paper, we conducted a survey on the existing erasure codes and libraries to identify the best algorithm for the Cloud-RAID application. To this end we identified determining criteria with reference to our use case.

EVEN/ODD and RDP codes can not be applied in our implementation. On the one hand, there is no publicly available implementation of the algorithms, as the codes are under patent protection. On the other hand, we want the number of coding chunks to be freely selectable. This is due to fact, that Cloud-RAID assigns the coding parameters based on user's requirements on the availability of hosted data. With this constraint, Minimal Density codes are not applicable for our specific use case. We also can not benefit from the advantages of Rotated Reed-Solomon codes, as cloud storage services do not allow partial reads on data objects in general.

To assert the feasibility of the remaining algorithms we have to consider the cost structure of cloud storage services. Vendors differ in pricing scheme and performance characteristics. Some providers charge a flat monthly fee, others negotiate contracts with individual clients. However, in general pricing depends on the amount of data stored and bandwidth consumed in transfers. Higher consumption results in increased costs. Therefore we prefer algorithms causing only minimal overhead in terms of storage and traffic I/O. Hence, Cauchy Reed-Solomon (CRS) and classic Reed-Solomon (RS) codes seem to be the optimal choice, as they cause less storage overhead than Local Reconstruction Codes (LRC). Due to the better encoding and decoding performance of CRS in comparison to RS, we decided to use CRS codes in our application. However, in case of rather read- than than write-oriented usage scenarios, LRC might be the superior choice to CRS codes as they are able to reduce the volume of transferred data by 50%. Therefore, the implementation of LRC also be considered within the scope of future work and enhancement of Cloud-RAID.

References

[1] Nicholas Carr. *The Big Switch*. Norton, 2008.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010. ISSN 0001-0782. URL http://doi.acm.org/10.1145/1721654.1721672.

[3] Jeffrey Burt. Future for cloud computing looks good, report says. online, 2009. URL http://www.eweek.com/c/a/Cloud-Computing/Future-for-Cloud-Computing-Looks-Good-Report-850563/.

[4] Ponemon Institute. Security of cloud computing providers study. online, April, 2011. URL http://www.ca.com/~/media/Files/IndustryResearch/security-of-cloud-computing-providers-final-april-2pdf.

[5] Maxim Schnjakin, Dimitri Korsch, Martin Schoenberg, and Christoph Meinel. Implementation of a secure and reliable storage above the untrusted clouds. *Proceedings of 8th International Conference on Computer Science and Education (ICCSE 2013)*, 2013.

[6] Maxim Schnjakin and Christoph Meinel. Implementation of cloud-raid: A secure and reliable storage above the clouds. *Proceedings of 8th International Conference on Grid and Pervasive Computing (GPC 2013)*, 2013.

[7] Maxim Schnjakin and Christoph Meinel. Evaluation of cloud-raid: A secure and reliable storage above the clouds. *Proceedings of the 22th International Conference on Computer Communications and Networks (ICCCN 2013)*, 2013.

[8] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial & Applied Mathematics*, 8(2):300–304, 1960.

[9] JS Plank, KM Greenan, EL Miller, and WB Houston. GF-Complete: A comprehensive open source library for Galois Field arithmetic. Technical report, Technical Report UT-CS-13-703, University of Tennessee, 2013.

[10] Blaum M., J. Brady, J. Bruck, and Jai Menon. Evenodd: an efficient scheme for tolerating double disk failures in raid architectures. *Computers, IEEE Transactions on*, 44(2):192–202, 1995. ISSN 0018-9340.

[11] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *Proccedings of the 7th conference on File and storage technologies*, FAST '09, pages 253–265. USENIX Association, Berkeley, CA, USA, 2009. URL http://dl.acm.org/citation.cfm?id=1525908.1525927.

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

[13] Osama Khan, Randal Burns, James Plank, William Pierce, and Cheng Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proc. of USENIX FAST*, 2012.

[14] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. Erasure coding in windows azure storage. In *USENIX ATC*, 2012.

[15] Hu Yuchong, Henry CH Chen, Patrick PC Lee, and Yang Tang. Nccloud: Applying network coding for the storage repair in a cloud-of-clouds. In *USENIX FAST*, 2012.

[16] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The Theory of Error-correcting Codes: Part 2*, volume 16. Elsevier, 1977.

[17] Johannes Bloemer, Malik Kalfane, Richard Karp, Marek Karpinski, Michael Luby, and David Zuckerman. An XOR-based erasure-resilient coding scheme. 1995.

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

[19] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: an optimal scheme for tolerating double disk failures in RAID architectures. In *Proceedings of the 21st annual international symposium on Computer architecture*, ISCA '94, pages 245–254. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994. ISBN 0-8186-5510-0. URL `http://dx.doi.org/10.1145/191995.192033`.

[20] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 1–14, 2004.

[21] Mario Blaum and Ron M Roth. On lowest density MDS codes. *Information Theory, IEEE Transactions on*, 45(1):46–59, 1999.

[22] James S. Plank. The RAID-6 Liberation codes. In *In FAST-2008: 6th Usenix Conference on File and Storage Technologies*, pages 97–110, 2008.

[23] James S Plank. A new minimum density RAID-6 code with a word size of eight. In *Network Computing and Applications, 2008. NCA'08. Seventh IEEE International Symposium on*, pages 85–92. IEEE, 2008.

[24] Bianca Schroeder and Garth A Gibson. Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean to you? In *FAST*, volume 7, page 1, 2007.

[25] M. Luby. Code for Cauchy Reed-Solomon coding. Website, 1997. Open Source Code Distribution, available at http://www.icsi.berkeley.edu/~luby/cauchy.tar.uu.

[26] Zooko Wilcox-O'Hearn. zfec 1.4.24. Website, 2012. Open Source Code Distribution, available at https://pypi.python.org/pypi/zfec.

[27] James S Plank, Scott Simmerman, and Catherine D Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-Version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.

[28] Cleversafe Inc. Cleversafe Dispersed Storage. Website, 2008. Open Source Code Distribution, available at http://www.cleversafe.org/downloads.