Fed-DIC: Diagonally Interleaved Coding in a Federated Cloud Environment

Giannis Tzouros(✉) and Vana Kalogeraki

Department of Informatics, Athens University of Economics and Business, Athens, Greece
{tzouros,vana}@aueb.gr

Abstract. Coping with failures in modern distributed storage systems that handle massive volumes of heterogeneous and potentially rapidly changing data, has become a very important challenge. A common practice is to utilize fault tolerance methods like Replication and Erasure Coding for maximizing data availability. However, while erasure codes provide better fault tolerance compared to replication with a more affordable storage overhead, they frequently suffer from high reconstruction cost as they require to access all available nodes when a data block needs to be repaired, and also can repair up to a limited number of unavailable data blocks, depending on the number of the code's parity block capabilities. Furthermore, storing and placing the encoded data in the federated storage system also remains a challenge. In this paper we present Fed-DIC, a framework which combines Diagonally Interleaved Coding on client devices at the edge of the network with organized storage of encoded data in a federated cloud system comprised of multiple independent storage clusters. The erasure coding operations are performed on client devices at the edge while they interact with the federated cloud to store the encoded data. We describe how our solution integrates the functionality of federated clouds alongside erasure coding implemented on edge devices for maximizing data availability and we evaluate the working and benefits of our approach in terms of read access cost, data availability, storage overhead, load balancing and network bandwidth rate compared to popular Replication and Erasure Coding schemes.

1 Introduction

In recent years, the management and preservation of big data has become a vital challenge in distributed storage systems. Failures, unreliable nodes and components are inevitable and such failures can lead to permanent data loss and overall system slowdowns. To guarantee availability, distributed storage systems typically rely on two fault tolerance methods: (1) Replication, where multiple copies of the data are made, and (2) Erasure Coding, where data is stored in

© IFIP International Federation for Information Processing 2020
Published by Springer Nature Switzerland AG 2020
A. Remke and V. Schiavoni (Eds.): DAIS 2020, LNCS 12135, pp. 55–72, 2020.
https://doi.org/10.1007/978-3-030-50323-9_4

the form of smaller data blocks which are distributed across a set of different storage nodes.

Replication based algorithms as those utilized in Amazon Dynamo [1], Google File System (GFS) [2,3], Hadoop Distributed File System (HDFS) [4,5] are widely utilized. These can help tolerate a high permanent failure rate as they provide the simplest form of redundancy by creating replicas from which systems can retrieve the lost data blocks, but cannot easily cope with bursts of failures. Furthermore, replication introduces a massive storage overhead as the size of the created replicas is equal to the size of their original data e.g. 3-way replication occupies 3 times the volume of the original data block in order to provide fault tolerance.

On the other hand, Erasure Coding [6] can provide higher redundancy while also offering a significant improvement in storage overhead compared to replication. For example, a 3-way replication creates 3 replicas of a data block and causes a 3x storage overhead for providing fault tolerance, while an erasure code can provide the same services for half the storage overhead or even lower by creating smaller parity blocks that can retrieve lost data more efficiently than full-sized replicas. Thus, Erasure codes are more storage affordable than replication but their reliability is limited to the number of parity blocks for repairing erasures. For example, an erasure code that creates 3 parity chunks cannot fix a data block with 4 or more unavailable or lost chunks.

Yet the most critical challenge with erasure coding is that it suffers from high reconstruction cost as it needs to access multiple blocks stored across different sets of storage nodes or racks (groups of nodes inside a distributed system) in order to retrieve lost data [7], leading to high read access and network bandwidth latency. The majority of the distributed file systems deploy random block placement [8] and one block per rack policies [9,10] to achieve optimized reliability and load balancing for stored encoded data. However, storing data across multiple nodes and/or racks can lead to higher read and network access costs among nodes and racks during the repairing processes. For example, in the worst case, repairing a corrupted or unavailable block in a node may require traversing all nodes across different racks, causing a heavy amount of data traffic among nodes and racks. Also, in a typical cross-rack storage, the user does not have any control over the placement of the data blocks across different racks, limiting the ability of the system to tolerate a higher average failure rate.

To reduce the cost of accessing multiple nodes or racks, file systems can keep metadata records regarding the topology of the encoded data codewords (groups that contain original data blocks alongside their parity blocks) in private nodes. However, the placement of the metadata files among the system's nodes is also challenging. For example, storing a codeword in a small group of nodes while keeping metadata about the data blocks scattered throughout the public clouds instead of a specific storage node [11], will also require to traverse all nodes at worst in order to recover any failed data inside the codeword. This problem leads to high cross-node read and network access costs, despite the use of metadata.

In this paper we propose Fed-DIC (**Fed**erated cloud **D**iagonally **I**nterleaved **C**oding), a novel compression framework deployed on an edge-cloud infrastructure where client devices perform the coding operation and they interact with the federated cloud to store the encoded data. Fed-DIC's compression approach is based on diagonal interleaved erasure coding that offers improved data availability while reducing read access costs in a federated cloud environment. It employs a variation of diagonally interleaved codes on streaming data organized as a grid of input records. Specifically, the grid content is interleaved into groups that diagonally span across the grid, and then the interleaved groups of data are encoded using a simple Reed-Solomon (RS) erasure code. Next, our framework organizes the encoded data into batches based on the number of clusters in the federated cloud and places each batch to a different cluster in the cloud, while keeping a metadata index of the locations of each stored data stream. The benefit is that Fed-DIC will only access the cluster with the requested data records and retrieve the correspondent diagonals, enabling the system to efficiently extract the corresponding records.

Fed-DIC has multiple benefits: it maximizes the availability of the encoded data by ordering input data into smaller groups, based on diagonally interleaved coding, and encoding each group using the erasure coding technique. Furthermore, it supports efficient archival and balances the load by storing each version of the streaming data array in a rotational basis among the storage nodes, e.g. if we have an infrastructure with 3 file clusters, for the first version of the data array, the first batch of diagonals is stored on the first node cluster, the second batch on the second node cluster and the third batch on the third cluster. For the second version of the array, the first batch of diagonals is stored on the second cluster, the second batch on the third cluster and the third batch on the first cluster and so on. We present an approach how multiple storage usage can optimize read access costs while keeping data availability and low bandwidth cost for retrieving data by utilizing multiple storage clusters in the same cloud environment instead of storing data in a single cluster. We illustrate the effectiveness of our approach with an extended experimental evaluation in terms of read access cost, data availability, storage overhead, load balancing and network bandwidth rate compared to popular Replication and Erasure Coding schemes.

2 Background

In this section we provide some background material regarding the technologies that we utilize at Fed-DIC: the Federated Cloud environment, Erasure Coding and Diagonally Interleaved Coding.

2.1 Federated Storage Systems

Many large-scale distributed computing organizations that need to store and maintain continuous amounts of data deploy distributed storage systems, such as HDFS [4,5], GFS [2,3] (which were mentioned above), Ceph [12], Microsoft Azure

[13,14], Amazon S3 [15], Alluxio [16] etc., which comprise multiple nodes, often organized into groups called racks. Currently, most of these systems write and store large data as blocks of fixed size, which are distributed almost evenly among the system's nodes using random block placement or load balancing policies. In each system, one of the nodes operates as the master node e.g. the NameNode in HDFS, that keeps a record of the file directories and redirects client requests toward the storage API for opening, copying or deleting a file. However, these policies are limited as they depend on the size of the data stored in the systems as well as the policies followed by the specific storage nodes (*e.g.,* load balancing policies). Our framework assumes the deployment of multiple HDFS clusters within the federated cloud environment, each comprising a different master node and storage layer. The client edge device can communicate with each of the master nodes with a different interface in order to store different groups of data into separate HDFS clusters.

2.2 Erasure Codes

Distributed systems deploy erasure codes as a storage-efficient alternative to replication so as to guarantee fault tolerance and data availability for their stored data. Erasure codes are a form of Forward Error Correction (FEC) codes that can achieve fault tolerance in the communication between a sender and a receiver by adding redundant information in a message; this enables the detection and correction of errors without the need for re-transmission. For instance, a sender node encodes a file with erasure coding and generates a data codeword or a stripe containing original and redundant parity data. Next, the sender node sends sequentially the blocks of the encoded stripe to a receiver node. In its turn, the receiver node detects whether there is a sufficient number of available blocks in order to decode them into their original content. If no original blocks are received, the parity blocks can repair them up to a finite range.

The most commonly used erasure code algorithm is the Reed-Solomon (RS), a maximum distance separable code (MDS) which is expressed as a pair of parameters $(b,\ k)$ (RS$(b,\ k)$) where b is the number of input chunks on a data block and k is the number of parity chunks created by the erasure code. The parity chunks are generated by utilizing Cauchy or Vandermonde matrices over a $GF(2^m)$ Galois Field, where 2^m is the number of elements in the field and m is the word size of encoding. The code constructs a matrix of size $k \times d$ which contains values from the $GF(2^m)$ field that correspond to the dimensions of the matrix and represent the positions of the input chunks. Next, the RS code derives an inverse $k \times k$ submatrix from the previous matrix. The original matrix is multiplied by the inverse submatrix in order to convert the top square of the former into a $k \times k$ identity matrix which will keep the content of the original data chunks unaltered during the encoding and decoding processes. The result is a stripe of length $n = b + k$, that contains the b chunks of the original data and the k parity chunks generated by the code. RS is k-fault tolerant due to the fact that the original data can be recovered for up to k lost chunks. In other words, while replication needs to copy and store the original data $n + 1$ times,

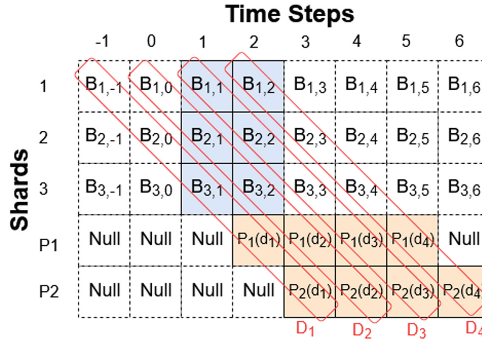


Fig. 1. A pictorial representation of diagonally interleaved coding for an input message with $(c, d, a) = (2, 5, 2)$. The data blocks are rearranged into diagonals and each diagonal is encoded into stripes ($D_1 ... D_4$) by the systematic code. B Symbols in time steps from -1 to 0 and from 3 to 6 are assumed to have zero, null or non-positive values, and they are not part of the input message.

erasure codes only require to store the data $\frac{n-k}{n}$ times, which costs considerably less compared to replication.

Reed-Solomon codes are also characterized by linearity [17]. In other words, they perform linear coding operations based on the Galois field arithmetic. More formally, given an $(b,\ k)$ code, let $B_1,\ B_2,\ ...,\ B_b$ be the b original data chunks and $P_1,\ P_2,\ ...,\ P_k$ be the k parity chunks. Each parity chunk P_j $(0 < j < k)$ can be expressed as $P_j = \sum_{i=1}^{b} (c_{ji} \cdot B_i)$, where $c_{ji} \subset GF(2^m)$ is a coding coefficient specified by the RS code for computing P_j.

This technique is limited as the redundancy provided by simple RS codes can repair up to k unavailable nodes. If there are more than k chunk erasures, the code will not be able to fully repair their original data. Our framework tries to deal with limited redundancy by deploying a more advanced erasure coding technique based on Reed-Solomon and Diagonally Interleaved Coding, the latter of which we describe in the next section.

2.3 Diagonally Interleaved Coding

Leong et al. have studied a burst erasure model in [18], where all erasure patterns with limited-length burst erasures are admissible so that they can construct an asymptotically optimal convolutional code that achieves maximum message size for all available patterns. This code involves stripes derived from one or more data messages interleaved in a diagonal order.

For a set of parameters (c, d, k), where c is the interval between input messages, d is the total number of symbols in the encoded message (original data and parity symbols) and k is the number of generated parity symbols, an input message is equally split into a vector of c columns and $d - k$ rows. Next, tables of blank or null symbols are placed around the message table that represent

non-existent messages before and after the input message. The symbols of the entire table are interleaved in diagonal pattern, forming well-defined diagonals containing at least one symbol from the input message. Finally, a systematic block code is used to create k parity symbols for every diagonal, thus constructing a convolutional code with $d-1$ diagonal stripes that can repair up to k lost symbols in each diagonal and span across d consecutive time steps. As a result, diagonally interleaved codes are able to handle an extended number of erasure bursts in one message and allow smaller erasures to be fixed without accessing massive amounts of data. In Fig. 1 we illustrate with an example how diagonally interleaved coding is applied for a single data block.

The process of splitting an input message into a vector can be applied only if the input data is organized in single data stripes. To optimize data availability, our framework uses a derived version of diagonally interleaved coding that takes as input data organized in a grid and interleaves all content into diagonals before encoding them with a Reed-Solomon code.

3 Challenges

In this section we present the challenges of existing schemes and how we propose to address them in our Fed-DIC framework.

High Read Access and Network Bandwidth Costs During Data Retrieval. One major challenge in typical cloud environments is the lack of user-oriented control in data distribution and storage. Most cloud systems store data blocks in randomly chosen nodes and nodes within racks in their clusters without balancing the load. For example, a system that uses an RS(b, k) to encode its streamed data, will distribute the $d = b + k$ chunks of the generated codeword to d different nodes in a random order. However, in cases of node failures, the system needs to retrieve data from other nodes within the rack or even across racks to retrieve parity data, leading to high read access costs and network overhead, which can considerably slow down the repair process.

Fed-DIC deals with this problem by uploading and distributing the encoded data to a federated cloud with multiple autonomous Hadoop clusters in the same network, each with a unique NameNode. To retrieve a particular data record, the framework keeps a metadata file containing the locations of the stored encoded data. The metadata file is created and can be accessed by the edge device in order to locate the requested data record and retrieve it faster with a significantly reduced read access latency, limited to the cluster where the specific data record is stored, without the need to traverse all nodes or maintain scattered metadata among nodes or clusters. Fed-DIC's topology in terms of the stored data among the clusters of the federated cloud, combined with the reduced storage size of the data chunks generated from its encoding process, provide significantly smaller read access costs and transfer bandwidth overhead for nodes in the cloud.

Limited Data Availability. Distributed systems deploy erasure coding methods to achieve higher redundancy than replication with more affordable storage cost. However, the availability provided by simple erasure codes such as Reed-Solomon codes for the encoded data is restricted to the number of parity chunks generated by the code. More specifically, a Reed-Solomon code that creates k parity data chunks from b original data chunks (RS(b, k)) can repair up to k failures between the original or parity data. If there are more than k unavailable or failed chunks in the stripe, the RS code will not be able to restore the data back to their original state.

To deal with this challenge, several advanced erasure codes have been presented, including alpha entanglement codes [19] and diagonally interleaved codes [18]. Fed-DIC uses a variation of diagonally interleaved coding on a group of streaming data containing input records from multiple sensor groups (columns) across multiple days (rows). The array data are interleaved diagonally and encoded with multiple parity chunks for each arranged diagonal pattern, achieving higher data availability and greater repairing range than conventional erasure coding methods.

Load Balancing Unreliability. Most large-scale distributed systems deploy load balancing policies for node distribution or utilizing one-node-per-rack [8–10] to balance the storage load across the cluster. However, most load balancing policies require the use of sophisticated techniques which may lead to load imbalances among nodes, especially when the number of the data chunks in a stripe exceeds the number of nodes that comprise a cluster.

Fed-DIC groups the encoded data diagonals into batches before they are stored to multiple node clusters in a non-random order. If the user decides to upload a data array and store it over the old one, the framework rotates the directions of the clusters in which the new batches will be stored in order to achieve good load balancing.

4 Design of the Fed-DIC Framework

To deal with the above problems of conventional erasure coding on federated clouds, we designed and developed Fed-DIC (**Fed**erated cloud **D**iagonally **I**nterleaved **C**oding), a framework that utilizes diagonal interleaving and erasure coding on streaming data records in a federated edge cloud environment. The goal of our framework is to reduce the read access cost and network overhead caused by accessing multiple nodes in a federated cloud while maximizing data availability for the data stored in the federated cloud environment. Fed-DIC also supports load balancing by storing multiple versions of the data records among clusters in a rotational order, while keeping storage availability, using the techniques we have developed and its API for distributing the data and balancing them across the clusters. In cases of high load in a cluster due to data congestion or unavailable nodes, Fed-DIC can reconfigure the number of batches and the content size of each batch in order to achieve load balancing by storing data to a smaller number of clusters with more nodes and larger storage space.

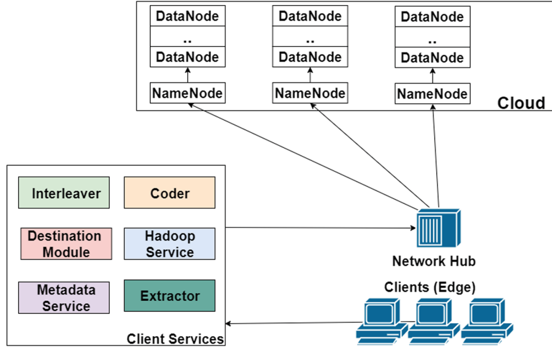


Fig. 2. The architecture of Fed-DIC, which comprises the client devices, where all operations are performed, the network hub, which connects the clients with the cloud and the federated cloud, which contains multiple independent storage clusters.

4.1 Framework Architecture

As illustrated in Fig. 2 Fed-DIC comprises three main components: the client side (edge devices), a federated cloud comprising multiple independent clusters where each cluster consists of multiple independent nodes, and a network hub that connects the two other components through the network. The client devices are operated by the user and provide six services: (1) The `Interleaver module` which re-orders the input data set into a grid and interleaves them into diagonal groups, (2) the `Coder module` which encodes all diagonal groups prior to the uploading process and decodes received diagonal stripes containing user-requested data, (3) the `Destination module` which splits the encoded stripes into batches and configures the order of destination clusters where the batches will be stored, (4) the `Hadoop Service` which communicates with the NameNodes of each cluster in order to upload the diagonal stripe batches, (5) the `Metadata service` which creates a metadata index file during the upload process and provides a query interface for the user during the retrieval process, and (6) the `Extractor module` which searches through a received diagonal stripe in order to extract the data record requested by the user and store it to a new file.

Our framework works as follows: A client takes as input a set of streaming data records and organises them into a grid of D columns and G rows. The data records in the grid are re-ordered into $C = D + G - 1$ diagonal groups, which are then encoded with Reed-Solomon, generating up to k parity chunks per diagonal using an 8-bit Galois Field. Next, Fed-DIC groups the diagonal stripes into H batches and stores each batch into a different cluster in the federated cloud. Simultaneously, the client creates a metadata file that contains information for each stored data record: The day the record was created, the group of sensors that generated the record and the diagonal stripe in which the record was interleaved. To retrieve selected data records, the client receives user-created request queries about data records and communicates with their correspondent clusters to download the stripes that include the records so as to extract their contents

in output files. To upload a new version of the already stored data while archiving the older versions, the cluster destinations are rotated in a stack order by setting the first cluster destination at the position of the last cluster destination in a circular pattern. In that way, Fed-DIC achieves not only the maintenance of multiple versions, but also load balancing throughout all clusters within the federated cloud. If Fed-DIC kept uploading newer versions into the same clusters each time, there could have been inconsistencies between the clusters. Especially, the first and last clusters in the cloud would have smaller data load than the other clusters.

4.2 Read Access Latency

The read access cost for a data query q from a group of Q queries, is given by the sum of the access time a client needs to traverse l lines in the metadata file to find the requested data, the latency needed to access any h clusters that contain the data ($h \leq H$) and the search delay caused by any missing d data chunks in a cluster. The probability p_i shows if a chunk is available for transferring. If $p_i = 0$, the chunk is missing. This is computed by the following formula:

$$T_q = l \cdot r_{md} + h \cdot r_h + \sum_{i=1}^{d} ((1 - p_i) \cdot t_m)$$

where r_{md} is the time a client needs to read a line from the metadata file, r_h is the time to access a cluster in the federated cloud and t_m is the search delay caused by missing data chunks in the cloud. The read access latency L_q for downloading and extracting a requested data query q is given by the access cost T_q which was computed previously, plus the time required to download all available d chunks in the diagonal stripe that includes the data using an internet connection of B bandwidth and the computation time T_q^{dec} a client needs to decode the diagonal stripe so as to extract the result. The formula for the overall query storage latency is given below:

$$L_q = T_q + \frac{\sum_{i=1}^{d} (p_i \cdot t_p)}{B} + T_q^{dec}$$

where t_p is the elapsed time for an available data chunk to be transferred from the federated cloud to a client device. Similarly, the total read access latency L_Q is the sum of the read access latency for all Q queries:

$$L_Q = \sum_{q=1}^{Q} (L_q)$$

The read access latency for erasure coding is computed in a similar way to L_q, with the only difference that the metadata access time is not taken into account.

4.3 Data Loss Percentage

When stored chunks are missing or unavailable in the federated cloud due to failures or nodes being disabled in the cloud's clusters, erasure codes try to

utilize any available parity chunks in order to reconstruct the damaged encoded file. However, if a decent amount of chunks are not available in a cluster, there may be permanent loss of the original data, due to the number of available data chunks being insufficient for use with erasure codes. The data loss percentage D_C of a fault tolerance method is measured by the fraction of the probability p_i of a data chunk c_i being available with the total number of data chunks in the entire cloud, subtracted from 1, as follows:

$$D_C = (1 - \frac{\sum_i^C (p_i \cdot c_i)}{C}) \cdot 100$$

4.4 Framework API

Fed-DIC provides an API with the following four operations:

Encode(). This operation interleaves the input data set into D diagonal data groups of varied length. Then, it merges data in each group into new data blocks so as to be encoded with a unique Reed-Solomon erasure code.

Store(). This operation groups the encoded diagonal data stripes into H batches containing an equal number of (D/H) codewords in each batch and communicates with all the NameNodes of the federated cloud in order to upload and store each batch in a different cluster, while keeping track of the data locations and information in a metadata file stored in the client devices. The metadata file can be shared and backed up in all clients in order to avoid any corruptions. If, for any reason, the cloud changes the location of its clusters, the clients need to update the metadata accordingly. However, a small non-significant access overhead may occur in the case that the client device that performs the Store() process becomes unavailable and the metadata have to be accessed from another client. Due to the integrity of our private client nodes, the probability of this situation is extremely rare, so it is not considered when measuring the read access latency.

Retrieve(). This method provides an interface to the user for entering multiple queries regarding a data record the user aims to retrieve. Once the user issues his queries, the method searches for each requested data record the diagonal stripe in which it is included and downloads it accessing immediately the corresponding storage cluster.

Decode(). Once the clients receive the diagonal stripes with the data requested by the user, this operation decodes any available chunks in a stripe into its original merged data block and extracts the requested result from the block before deleting it.

4.5 Uploading and Downloading Algorithms

We describe the two main algorithms implemented by our framework:

Storing Data to the Federated Cloud: A client takes as input the data records to be uploaded, these correspond to G sensor data groups over a time period R days, stored in .csv files. The client invokes the *Encode()* operation to organize the content into a grid with dimensions $G \times R$, where its elements are interleaved into C dynamic diagonal arrays of varied length (as shown in Fig. 1). Records are inserted into the grid according to the day and sensor group indicated on the record. Starting with the record of the last sensor group during the first day, the client forms a diagonal line from bottom right to top left and inserts any existing grid elements in the diagonal line, into a dynamic diagonal array. The diagonal arrays span through the entire grid with the last one containing only the record of the first sensor group during the last day. Next, in each diagonal array, the data in the elements are merged into a single data object and encoded using a (b, k) Reed-Solomon code. The encoding process splits each merged data object into equally sized b chunks and generates k parity chunks, creating a stripe of length $d = b + k$. Next, the client uses the operation *Store()* to group the diagonal stripes into H batches containing an equal number of (C/H) stripes in each batch and to upload the batches into the different clusters of the federated cloud by communicating with every NameNode within the cloud. Once the NameNode of a cluster receives the data, it distributes the chunks in random order to its nodes. During the storage process, the clients write and store metadata records about the stored data, their version, the date and sensor group as well as the number of the diagonal stripes they belong to. The metadata file helps the edge devices to access the stored data faster and more easily by reducing the access costs among the HDFS clusters. The distribution of the batches is performed in a sequential way. For example, in a federated cloud of F clusters, the first data batch is stored into the first cluster and so on until the last batch is stored in the F-th cluster. When the user wants to upload a new version of the data over the already stored versions, the clients swap the order of the cluster destinations by placing the first cluster destination right after the last cluster destination of the older version in a Last In, First Out (LIFO) order. In our example, for the second version of our data, the first batch will be uploaded into the F-th cluster, the second one to the first cluster and so on with the last cluster being uploaded to the (F-1)-th cluster. The way data records are stored in Fed-DIC enables us to traverse 1–2 clusters at most to recover any data segment. Whereas, conventional (b, k) Reed-Solomon would merge r_1 with every other record in the input into a single data block, split it into b original chunks and encode it using a Galois Field matrix to generate k parity chunks which are distributed to the cloud via Hadoop. Thus, even if a small part of data must be recovered, the data encoded with RS need to be restored in their entirety, which may require traversing all clusters in the cloud, incurring a heavy read access cost.

Retrieving Data from the Federated Cloud: The clients provide an interface to the user awaiting response queries. When the user issues a query, the clients gather all entered queries into a list array and use *Retrieve()* to search through the metadata file generated from the uploading process for the diagonal stripes where the query data are stored. For every entry in the query list, the client connects to the correspondent cluster to download the diagonal stripe with the requested data. If the edge device fails to download sufficient amount of chunks for restoring the stripe into its original data, it informs the user that the queried data from that diagonal stripe cannot be recovered. If it receives enough chunks from the stripe, it deploys *Decode()* to restore the diagonal stripe using RS(b, k) back to its original content. Finally, the clients search through the recovered data objects for the requested record entries and extract them as a result. When there are multiple concurrent requests from users, the clients schedule the requests to the hub in multiple rows according to the source cluster of the requested data and return the result for the oldest request each time.

5 Experimental Evaluation

In this section we evaluate Fed-DIC in terms of data loss, maximum transfer network rate and storage overhead, compared to the Replication and conventional Reed-Solomon Erasure Coding techniques. The client machines we used were desktop computers with an Intel i7-7700 4-core CPU at 3.5 GHz per core, with 16 GB RAM and a Western Digital WD10EZEX-08WN4A0 hard disk drive of 1 TB. The machines run Microsoft Windows 10 and are connected to the network using a Cisco RV320 Dual Gigabit WAN VPN Router with a data throughput of 100 Mbps and support of 20,000 concurrent connections. The router operates as our network hub and due to its specifications, the probability of a failure or bottleneck is extremely small. Although there are several ways to deal with such failures, this is outside the scope of our paper. For the experiments, we deploy via Oracle VirtualBox 4 clusters each comprising 4 nodes, 16 virtual machines (VMs) in total running Apache Hadoop 3.1.1 in Linux Lubuntu 16.04 for evaluating Fed-DIC against Replication and Reed-Solomon. For memory and disk allocation reasons, the VMs are running across 2 real desktop machines: Our client device and a second machine with the same hardware specifications as the first, which is connected to the same network. 8 VMs are running on each machine, connected to the same network as the client machines using a bridged adapter. Our setup is restricted to the equipment and network availability in our local computing and communication environment, however the algorithms we have developed can adapt well to accommodate larger clusters with thousands of nodes by modifying the number of batches in which the encoded data will be grouped as well as the content size in each batch. Also, we can set the batches to be stored in clusters with higher reliability within a large cloud. Our data set for the experiments is a collection of transport values obtained from SCATS sensors that are deployed in the Dublin Smart City [20]. This data set contains a huge amount of records with information regarding the specific sensor

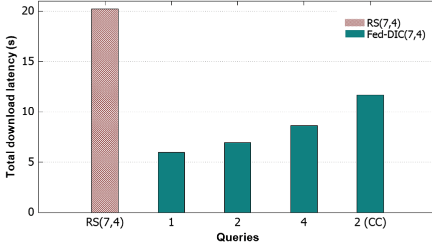


Fig. 3. Read access latency for Reed-Solomon and Fed-DIC (multiple queries)

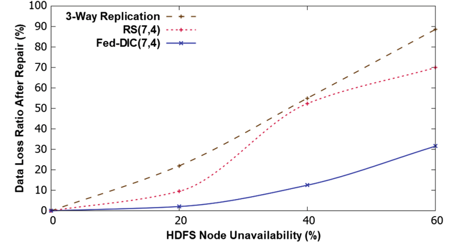
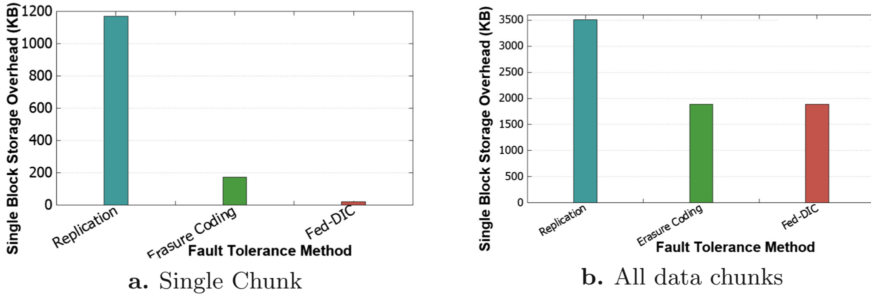


Fig. 4. Data Loss rate among Replication, Erasure Coding and Fed-DIC

that captured the snapshot and its capture date; the data needs to be stored and maintained in the cloud to be further analyzed by the human operators (i.e., to identify congested streets and entire geographical areas over time). Fed-DIC is responsible to store and recover this data to and from the cloud.

Our first experiment involves the total read latency of recovering data with Fed-DIC (7, 4) compared to Reed-Solomon (7, 4). For RS (7, 4) we merge the input files of the data grid used by Fed-DIC to a single .csv file. When the file is encoded to a stripe of 11 chunks (7 original and 4 parity), we distribute 3 chunks to each of the first 3 clusters, with the last 2 being stored in the last cluster. Note, that Reed-Solomon could retrieve the encoded file traversing only 3 clusters instead of going through all 4 clusters. In fact, Fed-DIC could also be easily configured (by appropriately setting the number of batches where diagonal stripes are grouped) to store and retrieve the data successfully utilizing only 3 clusters. However, in order to take advantage of the entire experimental environment (4-cluster cloud system with a total of 16 nodes) we utilize all 4 clusters for both techniques, to avoid load imbalances (data distributed in 3 clusters, while the 4th cluster is unused) and minimize the impact on the data loss percentage (in cases of failures). Due to the data chunks spanning across all 4 clusters, a simple decoding process with RS takes almost 20 s to complete, as seen in Fig. 3. This happens due to the clients having to access all 4 clusters in order to download all the chunks needed for recovering the stripe's original data. Even if we request a small portion of the encoded data, Reed-Solomon has no built-in features that allow us to retrieve a specific part of data, so it will still have to retrieve and decode the entire file content in order to give us an output. Our Fed-DIC technique on the other hand, reduces the total access latency by returning only requested parts of the stored data instead of the entire data content by accessing 1 to 2 clusters at most. For 1 to 4 queries for data inside the same cluster, Fed-DIC achieves at least 60% lower read access latency compared to RS. Even in the case that we request 2 data queries that are stored in two different clusters, Fed-DIC still reads the data in a shorter time compared to RS.



a. Single Chunk

b. All data chunks

Fig. 5. Storage overhead for Replication, Erasure Coding, and Fed-DIC

Our second experiment evaluates the reliability between 3-way Replication, RS (7, 4) and Fed-DIC (7, 4) in the data loss scenario. We performed 3 runs of experiments. As Fig. 4 indicates, due to its organized multi-cluster storage policies, Fed-DIC manages to achieve lower data loss rates than RS. Even when only up to 40% of the nodes are available in the federated cloud, Fed-DIC may be able to maintain a sufficient number of chunks in some diagonal stripes, which allows it to restore a portion the original data.

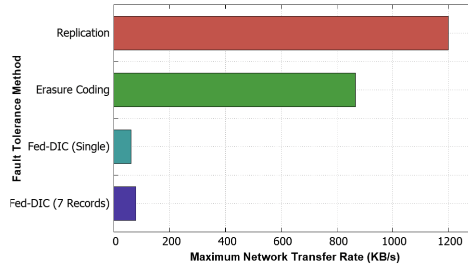
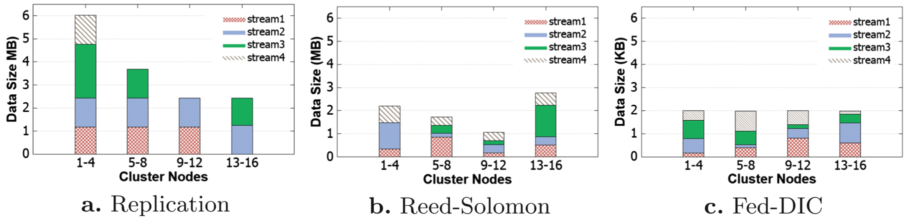


Fig. 6. Comparing Replication, Erasure Coding and Fed-DIC in terms of maximum network transfer rate (single record retrieval and 1 diagonal retrieval)

The next experiment we evaluate the storage overhead and the maximum network transfer rate between these fault tolerance methods. As Fig. 5a shows, Replication stores the entire data content inside the cluster without splitting it, causing a large storage overhead even for single blocks, compared to a chunk produced by simple Erasure Coding and Fed-DIC. In Fig. 5b we present the total storage overhead for all three methods. 3-Way replication occupies a massive portion of the storage with all 3 replicas combined, while all chunks generated by Erasure Coding and Fed-DIC produce lower overheads, with the latter occupying slightly less storage than erasure coding due to the varied sizes of the chunks. We also measured the rates during data transferring using performance monitoring programs included with Lubuntu OS. As seen in Fig. 6 due to the size of the

replicas, Replication severely burdens the network with a high transfer rate of 1.2 MBps, followed by Erasure Coding with a transfer rate of 900 KBps. Fed-DIC operates with smaller data transfers and thus provides smaller and less burdening network data rates when transferring one or multiple queried data records.

Finally, Fig. 7 shows the load balancing achieved in the three fault tolerance methods between 4 4-node clusters, while uploading 4 different data streams with similar sizes. Due to the random distribution of replicas and chunks in the HDFS cloud, Replication and client-side Reed-Solomon erasure codes are very inconsistent in terms of load balancing. Specifically, a majority of data may be stored to one cluster, while other clusters store less data, even though Erasure Coding seems more consistent than Replication. It is worth to note that we do not consider HDFS server-side erasure coding since it requires a code with higher parameters, which generates a number of chunks equal to the number of nodes in a single cluster. Meanwhile, Fed-DIC, using the rotational stack policy for cluster destinations described previously, it can store new streams in the federated cloud's clusters in a different order for every stream. Since our framework stores data of different size in each cluster in every uploading process, it can maintain an almost perfect load balance between H clusters for each H uploaded streams. For example, in Fig. 7 for every 4 streams uploaded in the cloud, Fed-DIC can achieve storage consistency and good load balancing between the 4 clusters.



a. Replication **b.** Reed-Solomon **c.** Fed-DIC

Fig. 7. Load balancing between 4 file clusters for all fault methods including Fed-DIC

6 Related Work

Several approaches over the last decade have been proposed for improving read access costs and the reliability of erasure coding in cloud storage environments. In particular, a method that drastically improves read access costs and data reconstruction in erasure coded storage systems is Deterministic Data Distribution, or D^3 for short [7]. D^3 maximizes the reliability of a storage system and reduces cross rack repair traffic by utilizing deterministic distribution of data blocks across the storage system. D^3 uses orthogonal arrays to define the data layout in which the data will be distributed across multiple racks, ignoring the

one block per rack placement, while balancing the load among nodes across the system's racks. This implementation works on single HDFS clusters with multiple racks but it does not seem to support federated clouds or other systems with independent clusters, unlike our approach with Fed-DIC. Even if we modify D^3 to support multiple clusters, the clusters need to contain a certain number of nodes in order to apply server-side erasure coding, whereas in Fed-DIC, erasure coding is performed by the client devices.

Simple erasure codes provide efficient fault tolerance but their reliability is restricted to the parameters set by the user. Advanced erasure coding techniques like Alpha entanglement codes by Estrada et al. [19], increase the reliability and the integrity of a system compared to normal Reed-Solomon codes by entangling old and new data blocks and creating robust, flexible meshes of interdependent data with multiple redundancy paths. Also in the Ring framework for key-value stores (KVS) [21], Taranov et al. introduce Stretched Reed-Solomon (SRS) codes which support a single key-to-node mapping for multiple resilience levels. These lead to higher and more expanded reliability compared to conventional Reed-Solomon codes. However, this work is only restricted to key-value stores and is not available to conventional databases for use. Also, unlike our work, the reliability ranges of SRS are limited only to the parameters of specific key-to-node mappings.

Hybris [11] by Dobre et al. is a hybrid cloud storage system that scatters data across multiple unreliable or inconsistent public clouds, and it stores and replicates metadata information within trusted private nodes. The metadata are related to the data scattered across the public clouds, providing easier access and strong consistency for the data, as well as improved system performance and storage costs compared to existing multi-cloud storage systems. In our case, Fed-DIC uses metadata containing information about the topology of data stored in a federated cloud so that the client can connect immediately to the cluster that contains a requested portion of the data, thus drastically reducing the read access cost in these systems compared to simple erasure codes.

7 Conclusion

In this paper, we presented Fed-DIC, our framework that integrates Diagonal Interleaved Coding with organized storage of the encoded data in a federated cloud environment. Our framework takes as input data organized in a grid, interleaves them into diagonal stripes that are encoded using a Reed-Solomon erasure code. The encoded diagonal stripes are grouped into batches which are stored to different clusters in the cloud. The user issues queries to retrieve portions of the data without the need for the clients to access every cluster in the cloud, thus reducing the access cost compared to other methods like Replication and simple Erasure Codes. Our experimental evaluations illustrate the benefits of our framework compared to other fault tolerance methods in terms of total read access latency, data loss percentage, maximum network transfer rate, storage overhead and load balancing. For future work, one direction we are following is to deploy

Fed-DIC in a federated environment with different hardware equipment where we plan to evaluate the working and benefits as well as the corresponding costs of our approach when different types of equipment are utilized.

Acknowledgment. This research has been supported by the Computer Systems and Communications Laboratory at AUEB. The authors would like to thank Dr. Davide Frey for shepherding the paper. This research has been supported by European Union's Horizon 2020 grant agreement No 734242.

References

1. DeCandia, G., et al.: Dynamo: Amazon's highly available key-value store. In: ACM SIGOPS Operating Systems Review, vol. 41, no. 6. ACM (2007)
2. Ghemawat, S., Gobioff, H., Leung., S.-T.: The Google file system (2003)
3. Wang, M., et al.: Formalizing Google file system. In: 2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing, pp. 190–191. IEEE (2014)
4. Shvachko, K., et al.: The hadoop distributed file system. In: MSST, vol. 10, pp. 1–10 (2010)
5. Karun, A.K., Chitharanjan, K.: A review on Hadoop—HDFS infrastructure extensions. In: 2013 IEEE Conference on Information & Communication Technologies, pp. 132–137. IEEE (2013)
6. "Erasure coding vs. Replication: A quantitative comparison"
7. Li, Z., et al.: D^3: deterministic data distribution for Ecient data reconstruction in erasure-coded distributed storage systems. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 545–556. IEEE (2019)
8. Ambade, S.V., Deshpande, P.: Hadoop block placement policy for different file formats. Int. J. Comput. Eng. Technol. (IJCET) **5**(12), 249–256 (2014)
9. Sathiamoorthy, M., et al.: Xoring elephants: novel erasure codes for big data. arXiv preprint arXiv:1301.3791 (2013)
10. Muralidhar, S., et al.: f4: Facebook's warm {BLOB} storage system. In: 11th $\{USENIX\}$ Symposium on Operating Systems Design and Implementation (fOSDIg 2014), pp. 383–398 (2014)
11. Dobre, D., Viotti, P., Vukolić, M.: Hybris: robust hybrid cloud storage. In: Proceedings of the ACM Symposium on Cloud Computing, pp. 1–14. ACM (2014)
12. Weil, S.A., et al.: Ceph: a scalable, high-performance distributed file system. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, pp. 307–320 (2006)
13. Chappell, D., et al.: Introducing Windows Azure. In: Microsoft, Dec (2009)
14. Calder, B., et al.: Windows Azure storage: a highly available cloud storage service with strong consistency. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 143–157 (2011)
15. Amazon, E.: Amazon web services, November 2012. http://aws.amazon.com/es/ec2/
16. Chang, X., Zha, L.: 'The performance analysis of cache architecture based on Alluxio over virtualized infrastructure. In: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 515–519. IEEE (2018)
17. MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error-Correcting Codes, vol. 16. Elsevier (1977)

18. Leong, D., Qureshi, A., Ho, T.: On coding for real-time streaming under packet erasures. In: 2013 IEEE International Symposium on Information Theory, pp. 1012–1016. IEEE (2013)
19. Estrada-Galinanes, V., et al.: Alpha entanglement codes: practical erasure codes to archive data in unreliable environments. In: 2018 48th Annual IEEE/IFIP DSN, pp. 183–194. IEEE (2018)
20. SCATS. https://data.smartdublin.ie/dataset/traffic-volumes/resource/4d45af2f-5ec1-4728-820a-a0fe350ad1dd
21. Taranov, K., Alonso, G., Hoee, T.: Fast and strongly-consistent peritem resilience in Key-Value Stores. In: EuroSys, pp. 39–1 (2018)