Computer Modeling in Engineering & Sciences

Tech Science Press

DOI: 10.32604/cmes.2022.021795

check for updates

ARTICLE

Health Data Availability Protection: Delta-XOR-Relay Data Update in Erasure-Coded Cloud Storage Systems

Yifei Xiao and Shijie Zhou*

School of Information and Software Engineering, University of Electronic Science and Technology of China, Chengdu, 610054, China

*Corresponding Author: Shijie Zhou. Email: sjzhou@uestc.edu.cn

Received: 05 February 2022 Accepted: 19 May 2022

ABSTRACT

To achieve the high availability of health data in erasure-coded cloud storage systems, the data update performance in erasure coding should be continuously optimized. However, the data update performance is often bottlenecked by the constrained cross-rack bandwidth. Various techniques have been proposed in the literature to improve network bandwidth efficiency, including delta transmission, relay, and batch update. These techniques were largely proposed individually previously, and in this work, we seek to use them jointly. To mitigate the cross-rack update traffic, we propose DXR-DU which builds on four valuable techniques: (i) delta transmission, (ii) XOR-based data update, (iii) relay, and (iv) batch update. Meanwhile, we offer two selective update approaches: 1) data-delta-based update, and 2) parity-delta-based update. The proposed DXR-DU is evaluated via trace-driven local testbed experiments. Comprehensive experiments show that DXR-DU can significantly improve data update throughput while mitigating the cross-rack update traffic.

KEYWORDS

Data availability; health data; data update; cloud storage; IoT

1 Introduction

With growing of the ageing population and the related rise in chronic illness (e.g., diabetes [1] or Parkinson's disease [2]), the Internet of Things (IoT) has been widely identified as a potential solution to alleviate the pressures on healthcare systems [3]. For instance, Health Care Assistants (HCA) [4] (e.g., Remote Patients Monitoring) are generating a huge amount of data (called "health data" for brevity) in real time using IoT medical sensors and ambient sensors (Fig. 1). These massive amounts of health data are usually stored in Cloud Storage Systems (CSS) to enable applying different analytical techniques to extract the medical knowledge, such as detecting patients' health status, innovating methods for the diagnosis of different diseases, and how to treat them [5]. For example, medical images are usually used to assist the healthcare provider to predict diseases and make clinical decisions accurately [6]. However, with the explosively increasing of health data in CSS, the conventional storage techniques which poses many needs and challenges [3], such as

1. The need to develop infrastructures that are capable of processing data in parallel.
2. The need to provide safe data transmission [7,8] and data storage for the huge amount of unstructured data sets.
3. The need to provide a fault-tolerant mechanism with high availability.

In this paper, we focus on the third one: provide a fault-tolerant mechanism with high availability.

Erasure codes (EC) are a leading technology to achieve strong fault-tolerance in CSS [9]. Roughly speaking, as all the files in CSS are usually split into fixed-size data blocks, EC encode these data blocks to generate a small number of redundant blocks (also called parity blocks), such that a subset of data and parity blocks still suffices to recover the original data blocks. Compared to conventional replication (e.g., 3-replication), EC can assuredly maintain the same degree of fault tolerance with much less storage overhead and hence is preferable in practical storage systems. For example, the erasure-coded Quantcast File System saves 50% of storage space over the original HDFS, which uses 3-replication. Besides, EC have been widely used in CSS, such as Microsoft Azure [10], Google Cloud [11], Facebook Cluster [12] and Alibaba Cloud [13].

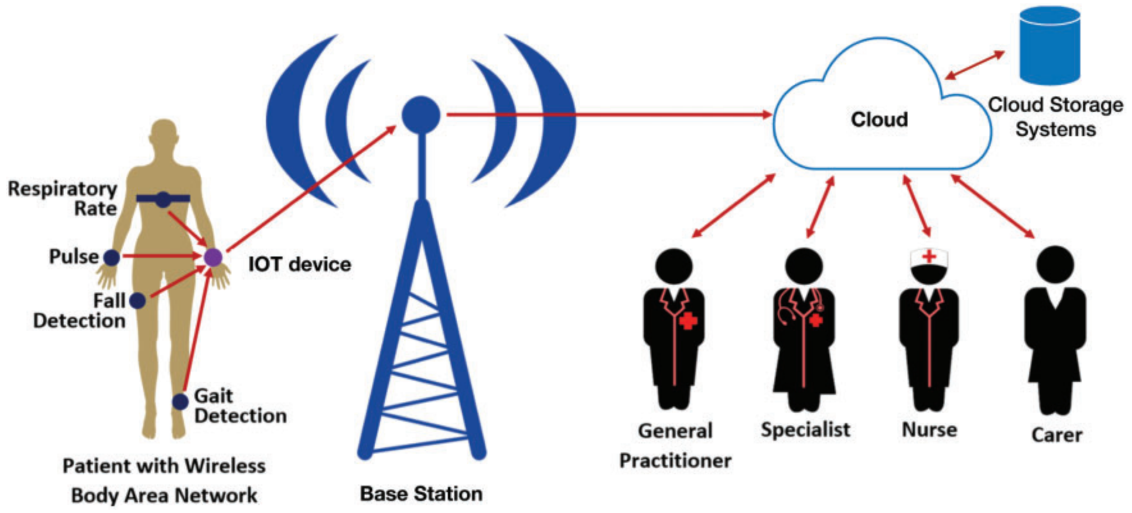


Figure 1: This figure shows a typical architecture of Remote Patients Monitoring [5], where the wearable sensors can measure the patients' vital signs-respiratory rate, pulse, and body temperature. In addition, some special-purpose sensors can be used for fall detection, gait detection, etc. These massive amounts of health data collected by IoT wearable devices are required to be safely transferred and stored in CSS [7,8]. Thus, it can allow the authorized relevant parties (such as caretakers or doctors) to safely access these sensitive and privacy data [14,15] and apply different analytical techniques

However, EC bring two new problems, namely *data repair (DR)* and *data update (DU)*. In DU, since each parity block is a linear combination of multiple data blocks, once the data block is updated, the relevant parity blocks must also be updated to achieve data consistency. Otherwise, it may cause permanent data loss (especially for the precious health data) in the face of node failures. Obviously, the health data in CSS is "hot data", which means it will be frequently generated or updated by various IoT devices. Thus, it will cause considerable network traffic for DU, especially for the cross-rack traffic, which is often oversubscribed and much more scarce than the inner-rack bandwidth [16]. To provide a fault-tolerant mechanism with high availability, it is necessary to provide an efficient and reliable DU scheme to solve the problem of data transmission in DU, especially for the cross-rack data transmission.

In order to alleviate the impact of network traffic, many works concentrate on network tier, as shown in Fig. 2. We re-examine and group them into two classes: ① improve bandwidth utilization (e.g., PUM-P, PDN-P [17], and T-Update [18]) and ② reduce network traffic (e.g., XORInc [19] and CAU [16]). Specifically, to improve bandwidth utilization, PUM-P [17] used a dedicated node called Update Manager (UM) to collect the update info and the old parity value of the relevant parity nodes for DU. T-Update [18] found that the traditional data transmission path is a star structure, which is detrimental to fully use the network bandwidth. What is worse, it is easy to cause a single-point bottleneck. Hence, T-Update modified the transmission path to a tree structure, which is great to leverage network traffic to other unused links and increase the network parallelism. To reduce network traffic, XORInc [19] offloads computation operations onto the programming network devices (i.e., modern switches with XOR computation capability and sufficient buffers). Thus, it can help data nodes forward the delta info to the relevant parity nodes. In order to mitigate the cross-rack traffic, Shen et al. [16] proposed CAU, which grouped the storage nodes into racks, and offered two optional update methods (data-delta-based update and parity-delta-based update) based on *batch update* and *relay*. However, despite the fruitful achievements of these great works, we found there is still massive room for network optimization, especially for the XOR-based DU.

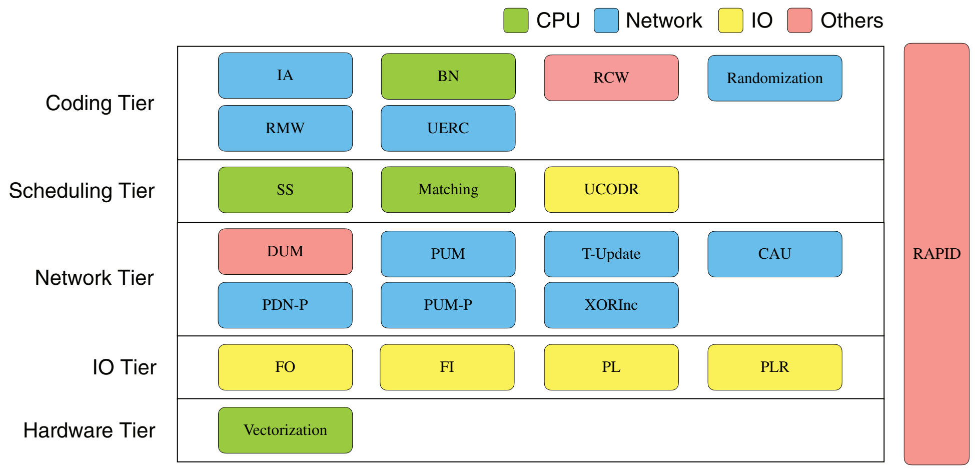


Figure 2: The different schemes are divided into five tiers (*Coding Tier, Scheduling Tier, Network Tier, IO Tier, and Hardware Tier*). The main goals (CPU, network, IO, and others) of these schemes are with different colors [20]

By carefully summarizing the previous works, we found four valuable techniques for network optimization: *delta transmission, XOR, relay* and *batch update*. The delta transmission means that we only transmit the delta info, since the DU size is generally smaller than the whole block size [21]. XOR means our scheme is based on XOR, as XOR-based DU can lead better throughput than RS-based DU. Relay means we exploit the relay nodes to forward data, which can fully use the unused links to mitigate the update traffic. In a word, we propose a simple and efficient mechanism Delta-XOR-Relay DU (DXR-DU) by using them jointly. To summarize, our work mainly makes the following contributions:

- We summarized the previous works on network optimization and found four valuable techniques: *delta transmission, relay, XOR* and *batch update*.
- Based on the four techniques, we proposed a novel data update scheme called DXR-DU, which can significantly improve throughput for DU. In other words, it can help CSS to build a fault-tolerant mechanism with high data availability.
- We implemented the DXR-DU prototype in Go programming language and analyzed that it can achieve the optimal cross-rack data update.
- We conducted numerous local testbed experiments based on our prototype. [1]Experiments on a local testbed show that DXR-DU can significantly reduce the cross-rack traffic and improve the update throughput.

2 Background

2.1 Cloud Storage Systems

It is well-known that modern DC deploy thousands of storage nodes in one or multiple geographic regions to provide large-scale storage services. These storage nodes are grouped into racks and further interconnected via the network core-an abstraction of aggregation switches and core routers [22]. Fig. 3 shows a typical CSS with three racks and each rack comprises four nodes.

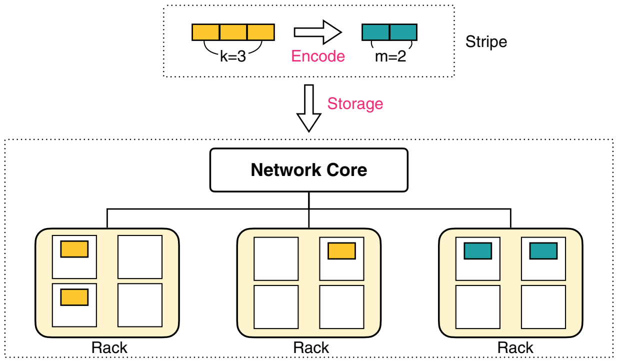


Figure 3: A typical RS (5, 3) CSS, here $n = 5$, $k = 3$, $m = 2$

2.2 Erasure Codes and RS Codes

A leading technique to achieve strong fault-tolerance in CSS is to utilize EC. As stated above, EC use the original data to generate more encoded data, thus they allow a fixed number of component failures in the overall system. EC are usually configured by two parameters: the number of data symbols k to be encoded, and the number of coded symbols n to be produced [9]. The data symbols and the coded symbols are usually assumed to be in finite field $GF(2^w)$ in computer systems.

RS codes [23] are a well-known erasure code construction and have been widely deployed in production [24–26]. RS codes are usually referred to as RS (n, k). For instance, Fig. 3 depicts a typical CSS with RS (5, 3), which encodes $k = 3$ data blocks into $m = 2$ parity blocks. These $n = k+m$ blocks group into a *stripe*, scattering in different nodes.

[1]The source code of DXR-DU is available for download at: http://git@gitee.com:xyf1989/cau.git.

2.3 Data Update

It is known that EC can be divided into two classes: RS-based codes and XOR-based codes [13]. Accordingly, we can classify DU into two types: *RS-based DU* and *XOR-based DU*.

2.3.1 RS-Based DU

Fig. 4 shows the typical encoding process of RS (5, 3), where the leftmost matrix (called *generator matrix*) encodes the data blocks (d_0, d_1, d_2) into a *codeword* $(d_0, d_1, d_2, p_0, p_1)$. After encoding, the data blocks (d_0, d_1, d_2) will be sent to the corresponding data nodes and the parity blocks (p_0, p_1) will be sent to the corresponding parity nodes. From Fig. 4 we can infer that, in a (n, k) RS-based CSS, each parity block could be represented by a linear combination of the k data blocks with the following equation:

$$p_i = \sum_{j=0}^{k-1} \alpha_{i,j} d_j, i \in [0, m-1] \tag{1}$$

where $m = n - k$ and all elements are numbers in GF(2^w) for some value of w. Suppose that d_h is updated to $d_h'(0 \le h \le k-1)$, Eq. (1) can be called for DU.

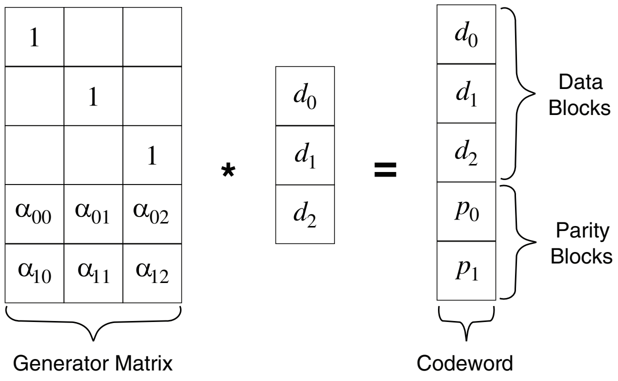


Figure 4: The encoding process of RS (5, 3). The leftmost matrix is called *generator matrix*, which encodes data blocks $(d_0; d_1; d_2)$ into codeword $(d_0; d_1; d_2; p_0; p_1)$

RS-delta-based: On the other hand, we can simply utilize the delta info $(d_h' - d_h)$ to renew the parity block with the following equation:

$$p_i' = \sum_{j \in [0,k-1], j \neq h} \alpha_{i,j} d_j + \alpha_{i,h} d_h' = p_i + \alpha_{i,h}(d_h' - d_h), i \in [0, m-1] \tag{2}$$

where p_i denotes the old value. In this way, we can simply transfer the delta of d_h (also called $\Delta d_h = d_h' - d_h$) to the parity node i.

2.3.2 XOR-Based DU

No matter Eq. (1) or Eq. (2) is selected for DU, a considerable number of multiplications are generated, which will significantly impede the performance of DU. To end this, as shown in Fig. 5, XOR-based encoding is proposed via *Binary Distribution Matrix (BDM)*, where each element e in GF(2^w) can be denoted by a matrix M(e) of $w \times w$ or a vector V(e) of $1 \times w$, thus, the generator matrix of size $k \times m$ can be converted to a new generator matrix of size $wk \times wm$ in GF(2) [9]. In this light, we can use the smaller element (w bits) to encode. According to Fig. 5, the parity blocks can be computed by the following equations:

$$p_{0,0} = d_{0,0} \oplus d_{1,0} \oplus d_{2,0} \oplus d_{2,2} \quad (3)$$

$$p_{0,1} = d_{0,1} \oplus d_{1,1} \oplus d_{2,0} \quad (4)$$

$$p_{0,2} = d_{0,2} \oplus d_{1,2} \oplus d_{2,1} \quad (5)$$

$$p_{1,0} = d_{0,0} \oplus d_{1,0} \oplus d_{1,2} \oplus d_{2,0} \quad (6)$$

$$p_{1,1} = d_{0,1} \oplus d_{1,0} \oplus d_{2,1} \quad (7)$$

$$p_{1,2} = d_{0,2} \oplus d_{1,1} \oplus d_{2,2} \quad (8)$$

where the matrix multiplications are now converted to XORs of data bits corresponding to the ones in BDM. Zhou et al. [9] proved it is more efficient to use XOR operation to encode instead of directly using RS-based encoding. In other words, XOR-based DU can significantly reduce the computation overhead than RS-based DU.

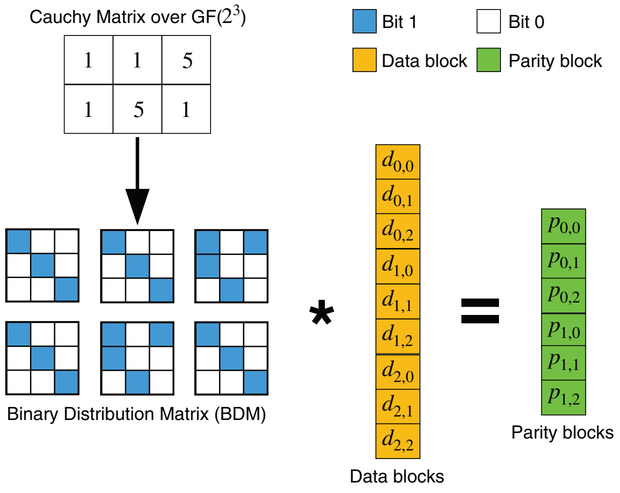


Figure 5: Encoding with BDM: the Cauchy matrix is converted to BDM, where the blue block denotes bit 1 and the white block denotes bit 0, identically, $k = 3$, $m = 2$, $w = 3$

2.3.3 Parity Update in CSS

As mentioned earlier, nodes are grouped into racks. We assume data racks are dedicated to data nodes and parity racks are dedicated to parity nodes. Without loss of generality, suppose that there are U_d blocks denoted by $(d_0, d_1, \ldots, d_{U_d-1})$ that are updated to $(d'_0, d'_1, \ldots, d'_{U_d-1})$ in the data rack R_d, based on Eq. (2), we can calculate the parity block p'_i ($i \in [0, m-1]$) with the following equation:

$$p'_i = \sum_{j=0}^{U_d-1} \alpha_{i,j} \Delta d_j + p_i = \Delta p_i + p_i \quad (9)$$

where Δp_i is the delta of parity block i. Suppose that parity block i ($i \in [0, m-1]$) are located in the parity rack R_p, and there are U_p parity blocks to update. As illustrated in Fig. 6, there are two options to renew the parity block: ① data-delta-based update, and ② parity-delta-based update [16].

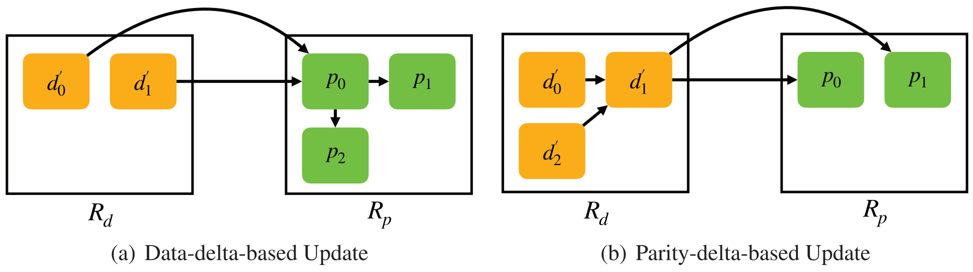


Figure 6: Examples of the data-delta-based update and parity-delta-based update: (a) $Ud = 2$ and $Up = 3$; (b) $Ud = 3$ and $Up = 2$

Data-delta-based update: which updates the parity blocks of a rack in batch via transmitting data delta blocks directly [22]. As shown in Fig. 6a, the number of data updates in R_d is less than the number of parity updates in R_p (i.e., $U_d < U_p$). Thus, we separately send the delta info (Δd_0, Δd_1) to the relay node P_0, when P_0 receives all the deltas, it calculates and forwards the new values for P_1 and P_2 via Eq. (9).

Parity-delta-based update: as shown in Fig. 6b, $U_d > U_p$, to mitigate the cross-rack traffic, parity-delta-based update is selected, where we select a data node as the relay node to collect the deltas in the same rack. Similarly, the relay node is responsible for regenerating the parity blocks via Eq. (9) and transferring the deltas to the relevant parity nodes.

3 Delta-XOR-Relay Data Update

In this section, we elaborate the design overview of Delta-XOR-Relay Data Update (DXR-DU).

3.1 Design Overview

Our study of previous works on network optimization found four valuable techniques for network optimization: *delta transmission, XOR, relay*, and *batch update*.

Recall that the existing two classes of network optimization: ① improve bandwidth utilization, and ② reduce network traffic. We found that the key technique to improve bandwidth utilization is using *relay*. For example, PUM-P [17] used a dedicated node called update manager (UM) as a relay node to compute the deltas of the relevant parity blocks, while PDN-P discarded it. CAU [16] selected a data node or a parity node as a relay node, and RackCU [22] selected a data rack or a parity rack as a relay rack. It sounds like the triangle principle: If the sum (network overhead) of the two sides (using relay node) is greater than the third side (directly sending data), it is unnecessary to use the relay; Otherwise, we should use the relay to fully exploit the unused links. Besides, the relay can be used for updating one block (e.g., T-Update [18]) or a group of blocks (e.g., CAU [16] and RackCU [22]), and the latter should consider *node grouping*. For example, CAU groups nodes into racks and selects a relay node for each rack.

To reduce network traffic, we found two key factors: *delta transmission* and *batch update*. The block size in CSS normally ranges from 1 MB to 64 MB [16,24]. But it is unnecessary to update the whole block, since DU is small (60% of them are less than 4 KB [21]). Thus, the better way is to transfer the delta of the updated block. Another key point is *batch update*. For instance, CAU proved the batch update is powerful for saving network traffic via setting the threshold at 100 (i.e., when 100 data requests arrive). However, the batch update has the disadvantage that will slightly sacrifice the system reliability. Fortunately, we can utilize the interim replication to maintain the system reliability and data availability at the same level as the baseline EC approach [16].

As mentioned above, the fourth valuable technique is XOR. The experimental results in Section 4 reinforce our determination to use XOR. In the next section, we will discuss how to use *delta, relay* and *batch update* jointly based on XOR-based DU.

3.2 Transmission Path

As far as we known, the transmission path is either a star structure (e.g., the baseline method) or a tree structure (e.g., T-Update, CAU, XORInc and RackCU). As mentioned earlier, the conventional star-structured path can easily cause single point bottleneck or even single point failure. Obviously, the tree-structured path is better. To build a tree-structured path, T-Update relies on the network distance (i.e., the hops) between nodes. While CAU groups nodes based on racks, and selects a relay node for each rack. Comparely, we believe CAU is more simple and easy for implementation. Besides, T-Update builds a tree only for one block, while CAU builds a tree for a group of related blocks. For example, in parity-delta-based update (Fig. 6b), CAU collects the deltas of a rack and directly transfers the merging result of parity block i ($\Delta p_i = \sum_{j=0}^{U_d-1} \alpha_{i,j} \Delta d_j$) to related parity nodes. We argue that transferring the parity deltas is better than transferring the data deltas one by one. Therefore, we build the transmission path based on CAU.

3.2.1 Delta Transmission

It has long been recognized that transferring the data block in delta style will substantially save network load than transferring the whole data block. However, few works indeed transfer the delta in implementation. Although it is just a implementation issue, we proved that it is significant performance differentiator in evaluation. Thus, in this section, we elaborate our way to transfer the delta.

Block merging for batch update: It is well-known that a dirty (updated) data block may be modified in different places within a batch time (as shown in Fig. 7), which means the delta info (the gray parts) is scattered. To end this, we employ a very straightforward way: we label the leftmost offset as *rangeL* and the rightmost offset as *rangeR*. Thus, we only transfer the [*rangeL, rangeR*] of the whole block. As mentioned earlier, DU is small (most updates are less than 4 KB), even though we pack these small and scattered blocks into a large piece, the optimization space is still huge.

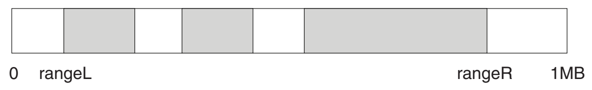


Figure 7: An example of a dirty block, which size is 1 MB. We label the leftmost offset as *rangeL* and the rightmost offset as *rangeR*

Delta Alignment: As mentioned above, we label the delta of an updated block as [*rangeL, rangeR*], but the ranges of distinct data blocks within a stripe are probably different, which prevents us from calculating the parity blocks. Therefore, before renewing the parity node in a relay node, we have to perform the delta alignment.

A typical example is depicted in Fig. 8, where there are four updated blocks within a stripe, and the ranges are $D_0[28, 1557]$, $D_1[1024, 8096]$, $D_2[356, 768]$, $D_3[4096, 14400]$, respectively. To compute the new value of the parity node, we use a dedicated node called *central controller* to align these four deltas based on the maximum range. Thus, D_0 can receive three deltas with identical range, and easily renew the parity node.

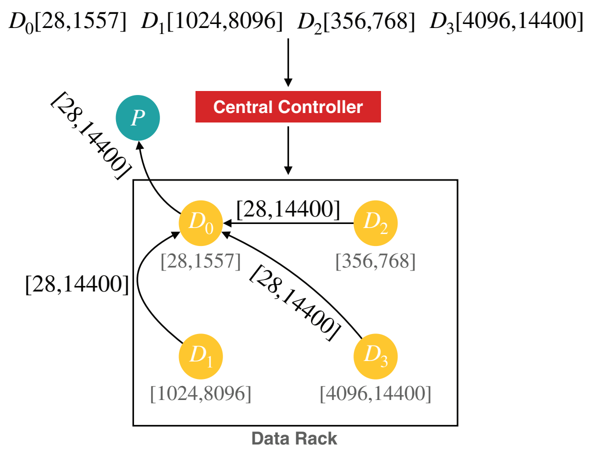


Figure 8: Delta alignment for batch update in DXR-DU

3.3 Design of DXR-DU

As mentioned above, we design DXR-DU based on the four valuable techniques: delta transmission, XOR, batch update and relay. As shown in Fig. 9, we first build the transmission path based on CAU, which offers two selective methods (data-delta-based update and parity-delta-based update). When the number of the updated data blocks in data rack is smaller than the number of updated parity nodes in parity rack (i.e., $U_d < U_p$), as shown in Fig. 9a, data-delta-based update is selected, which means we choose a parity node as the relay node to collect deltas and compute the new values for parity nodes. On the contrary, parity-delta-based update is selected, where we choose a data node as the relay node to collect the deltas and compute the parity deltas for the parity nodes.

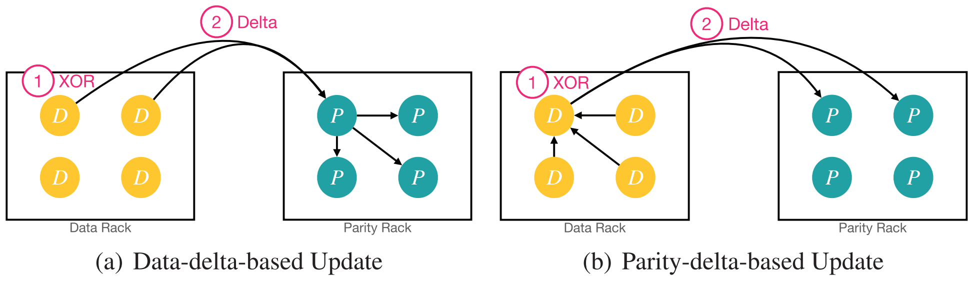


Figure 9: The design of DXR-DU is based on CAU, which offers two update methods: data-delta-based update and parity-delta-based update. But we have two extra techniques: ① XOR and ② delta transmission. In coding tier, we choose XOR-based DU. Meanwhile, we utilize the delta transmission to send data

Based on CAU, we have two extra techniques: ① We choose XOR-based DU to improve the update throughput in coding tier, unlike CAU, which relies on RS-based DU. ② We employ the delta transmission to mitigate the network traffic, especially for the cross-rack traffic.

Algorithm details: Algorithm 1 elaborates the main procedure to schedule the update requests in a batch. We first collect the data blocks (getting from user requests) in a batch, and perform block merging (Line 1). Then, we group these blocks into stripes (Line 2). For each stripe, according to the number of the updated data blocks in a data rack, we handle the data blocks of a stripe in this rack: 1) as mentioned above, if $U_d > U_p$, we use parity-xor-based update (Line 6). 2) Otherwise, we use data-xor-based update (Line 7).

3.4 Parity Update in XOR-Based DU

Similar to RS-based DU, in this section, we discuss the optimal cross-rack parity update in XOR-based DU. For ease of presentation, we take an example of a CSS, where there are $k = 4$ data nodes and $m = 4$ parity nodes. We set $w = 3$ and only show the blocks in one stripe (as shown in Fig. 10). Besides, we suppose the parity update equations are as follows:

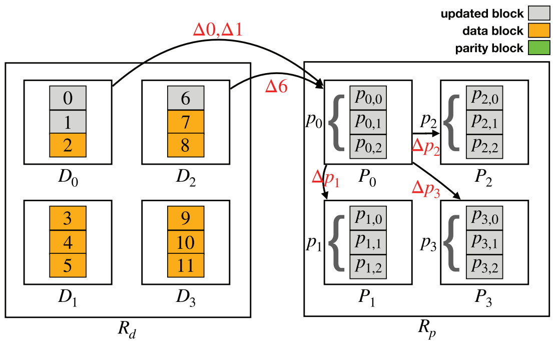
$$p_0 = d_0 \oplus d_2 \oplus d_3 \oplus d_4 \oplus d_5 \oplus d_8 \oplus d_{10} \tag{10}$$

$$p_1 = d_0 \oplus d_1 \oplus d_3 \oplus d_4 \oplus d_5 \oplus d_6 \oplus d_9 \oplus d_{11} \tag{11}$$

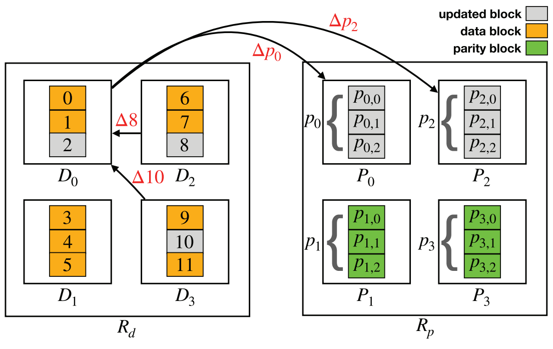
$$p_2 = d_0 \oplus d_1 \oplus d_3 \oplus d_5 \oplus d_7 \oplus d_8 \tag{12}$$

$$p_3 = d_0 \oplus d_1 \oplus d_3 \oplus d_5 \oplus d_7 \oplus d_{11} \tag{13}$$

$$p_i = (p_{i,0}, p_{i,1}, p_{i,2}), i \in [0, 3] \tag{14}$$



(a) Data-delta-based Update



(b) Parity-delta-based Update

Figure 10: Examples of the data-delta-based update and parity-delta-based update: (a) $Ud = 3$ and $Up = 2$; (b) $Ud = 2$ and $Up = 3$

As we focus on network optimization, it is unnecessary to know the exact equations of $p_{i,j}$, we just need to make sure that all parity could receive what they want, thus we label $p_i = (p_{i,0}, p_{i,1}, p_{i,2}), i \in [0, 3]$ as the all the data of node P_i needed. The key question is: how to update p_i to minimize the cross-rack network load? For example, as shown in Fig. 10a, if d_0, d_1, d_6 are changed in batch, according to Eq. (10) to Eq. (14), we need to update all parity blocks (p_0, p_1, p_2, p_3).

In this case, where the number of updated data blocks in the data rack is smaller than the number of updated parity nodes in the parity rack ($U_d < U_p$). Unlike the data-delta-based update in CAU, where there is only one updated block belonging to a node in a stripe. Here we should consider multiple updated blocks ($\leq w$) in a data node. To save cross-rack traffic, we should consider whether to transfer the xor result of multiple updated blocks or not. But we found we can not do that, because 1) every delta info may have different update range, 2) if we transmit the xor result ($d_0 \oplus d_1$) of d_0 and d_1 to P_0, while P_0 only needs d_0 (Eq. (1)), we can not extract d_0 from the xor result without transferring d_1. Therefore, we can only transfer them one by one (i.e., Δ0, Δ1, Δ6).

In data-delta-based update, where the number of updated data blocks (denoted by U_d) is smaller than the number of updated nodes in the parity rack (denoted by U_p), we should select a relay node for the parity rack. A small question is: how to select the relay node in the parity rack? We tested 2 options: 1) Random select, and 2) consider load balance, which means every round we select a different relay node for forwarding data. However, we test it and found it is unnecessary to do that. Thus, we choose the frist option.

On the other hand, if $U_d > U_p$, in parity-delta-based update, where we will select a relay node for the data rack, similar to *parity-delta-based update*, we will utilize the relay node to compute and transfer the deltas of the corresponding parity node. As illustrated in Fig. 10b, there are three updated data blocks (d_2, d_8, d_{10}) and 2 parity node to be updated (P_0, P_2), namely $U_d = 3 > U_p = 2$, thus we randomly select D_0 as the relay node to compute and forward the deltas of P_0 and P_2.

Algorithm 1: DXR-DU Algorithm

```
Data:
the list of request blocks with $(blockID, rangeL, rangeR)$ in a batch $blocks$ ;
the number of racks $numOfRacks$;
the index of the parity rack $ParityRackIndex$ ;
Result:
the total spending time $T$ ;
the total cross-rack traffic $Traffic$ ;
// group blocks into stripe, then get the list of stripe
1  $reqs = GetDistinctReqs(blocks)$ ;
2  $stripes = turnReqsToStripes(reqs)$;
   // traverse the list of stripe
3  for $stripe$ range $stripes$ do
4      for $i = 0$ to $numOfRacks$ do
5          if $i \neq ParityRackIndex$ then
6              if $getRackUpdateNums(i, stripe) > getParityUpdateNums(stripe)$ then // the number
                   of updated data blocks is more than the number of updated parity
                   blocks
7              $parityUpdate(i, stripe)$ ;
8              else $dataUpdate(i, stripe)$ ;
9          end
10     end
11 end
```

In a nutshell, *data-delta-based update* and *parity-delta-based update* are two methods to minimize the cross-rack network traffic. Compare to CAU, we proved it has advantages on update time, throughput and cross-rack traffic in experiments.

4 Performance Evaluation

In this section, we conduct extensive performance evaluation via local testbed experiments between the proposed approach DXR-DU and two well-known Counterparts: PDN-P and CAU. We summarize our major findings below: compared to the state-of-the-art schemes, ① DXR-DU saves more than 44.9% of cross-rack traffic in most cases (Section 4.3); ② DXR-DU increases 53.6% of update throughput (Section 4.4).

4.1 Preliminaries

Traces: We assess the update performance via trace-driven evaluation. We utilize MSR Cambridge Traces (MSR) [16], which record the I/O patterns from 13 core servers of a data center. Every trace consists of successive read/write requests, each of which records the request type (read or write), the start position of the requested data, and the request size, etc. According to the ranking results of the average update size of MSR Cambridge Traces in [16], we select 4 traces with dramatically distinct update sizes (sorted from small size to big size): *rsrch_2, hm_0, hm_1, proj_0*.

Counterparts: We compare DXR-DU to another 2 state-of-the-art schemes: (i) PDN-P [17] and (ii) cross-rack-aware update (CAU) [16]. We summarize these two schemes as below:

PDN-P: When a data block is updated, PDN-P directly send the delta to the relevant parity nodes, which means it builds a star-structured transmission path for each update.

CAU: As shown in Fig. 6, CAU updates parity blocks simply through the selective parity update: 1) if the updated data blocks of a data rack are more than the parity blocks of a parity rack, CAU updates the parity blocks via transferring parity delta blocks; Otherwise, it updates them through transferring the data delta blocks.

4.2 Implementation

Since open source implementations for PDN-P and CAU are not available, we design and implement the prototype of DXR-DU and its two counterparts (PDN-P and CAU) with Go programming language on Ubuntu 18.04. These schemes rely on Cauchy RS code implementations. Hence, we utilize the *reedsolomon* library which is the Go version of Jerasure library 2.0.

The system architecture of our prototype is illustrated in Fig. 11, where we choose RS(12, 4) (deployed in Windows Azure Storage [27]), where there exists $k = 8$ data nodes and $m = 4$ parity nodes. We utilize Linux tool *tc* [28] to group them into three racks via vitual Top-of-Rack switches (ToR) and set the cross-rack/inner-rack at 40/200 Mbps. Such a configuration can tolerate any four nodes failure as well as any single rack failure. Besides, we have another node called metadata server, which is used for metadata management. The metadata server also includes two components: the *client*, which is to generate user requests, and the *central controller*, which is responsible for sending commands to the storage nodes and receiving ACKs from them. In addition, the *agent* in storage nodes is responsible for performing the tasks (e.g., computing and forwarding data) according to the received commands. When the task is finished, it returns an ACK to the command sender. All the nodes are virtual machines (VM) which are generated from 3 Huawei H12M-03 servers via Proxmox VE [29]. Each VM is equipped with a dual core CPU, 2 GB memory and 32 GB disk.

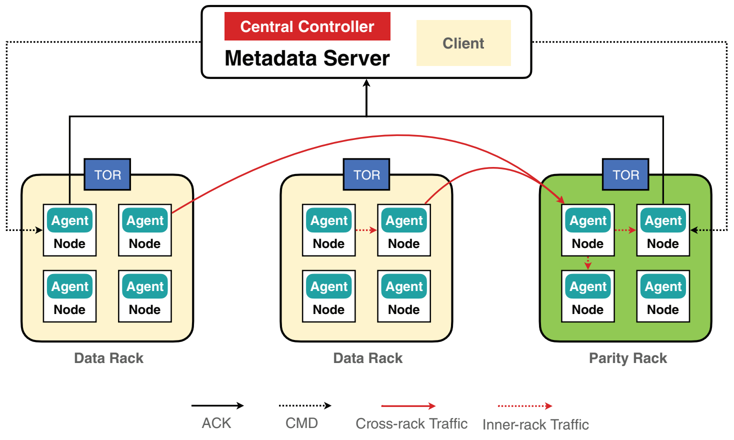


Figure 11: The figure depicts the system architecture of our prototype, where there are 3 racks and 13 nodes. The metadata server includes a central controller and a client. The client is to generate user requests and the central controller is to send commands to the storage nodes and receive ACKs from them. The agent in storage nodes is responsible for performing the tasks according to the received commands

4.3 *Cross-Rack Traffic*

As mentioned above, we believe that our proposed scheme has advantages on cross-rack traffic, thus we first keep eyes on the amount of induced cross-rack traffic.

Experiment A.1 (Impact of update size): We first study the impact of the update size by selected four traces with distinct update sizes: *rsrch_2, hm_0, hm_1, proj_0*. We configure the block size as 1 MB. Table 1 shows the cross-rack traffic for each update. Compared to PDN-P and CAU, DXR-DU reduces the cross-rack traffic by up to 98.0% and 71.6%, respectively. The result is actually out of our expectations but still consistent with the fact that DU is small.

Table 1: Comparison on cross-rack traffic with distinct update sizes

Schemes	Traces					
	rsrch_2	hm_0	hm_1	proj_0	Average	Improvement
PDN-P	4	4	4	4	4	98%
CAU	0.155	0.395	0.325	0.280	0.289	71.6%
DXR-DU	**0.060**	**0.030**	**0.171**	**0.065**	**0.082**	-

Experiment A.2 (Impact of block size): To assess the impact of block size, we set the block size as 0.25/1/4 MB, respectively. Table 3 exhibits that DXR-DU keeps efficiency on saving the cross-rack traffic with different block sizes. DXR-DU can reduce 98.4%, 68.2% of the cross-rack update traffic on average compared to PDN-P and CAU, respectively. The rationale is that DXR-DU utilizes delta transmission.

In a nutshell, with the help of delta transmission, DXR-DU can significantly mitigate the cross-rack update traffic by up to 44.9%–99.1%.

4.4 *Throughput*

As the health data is "hot data", it is significant to achieve an excellent DU throughput to maintain the data availability at a high level. In this paper, to compare various schemes fair, we assess the DU throughput of the schemes by changing the update size and block size.

Experiment B.1 (Impact of update size): We first evaluate the update time of a single block on average by changing th update size: *rsrch_2, hm_0, hm_1, proj_0*. Similarly, the default block size is 1MB. Table 2 shows the result that DXR-DU keeps efficiency on DU throughput. For DXR-DU, the update time of single block on average only needs 0.021 s. Compared to PDN-P and CAU, DXR-DU can save update time of single block on average by up to 89.9% and 53.6%, respectively.

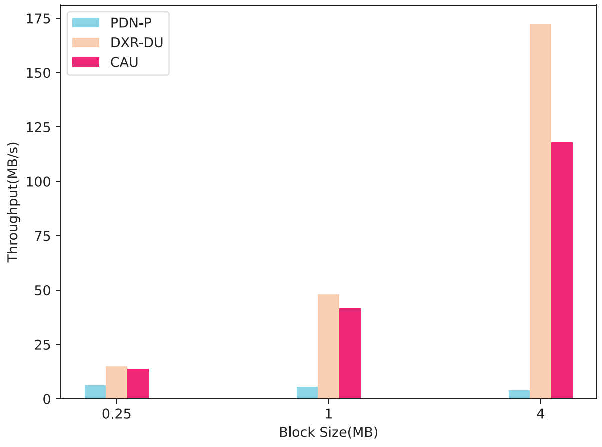
Experiment B.2 (Impact of block size): We further assess the DU throughput under different block sizes (0.25/1/4 MB), from Fig. 12 we observe that DXR-DU improves the update throughput by up to 13.8% and 88.8% when compared to PDP-P and CAU, respectively. Unsurprisingly, as DXR-DU wins the game in the comparison on cross-rack traffic, it also has significant advantages on throughput.

Table 2: Comparison on update time for single block with distinct update sizes

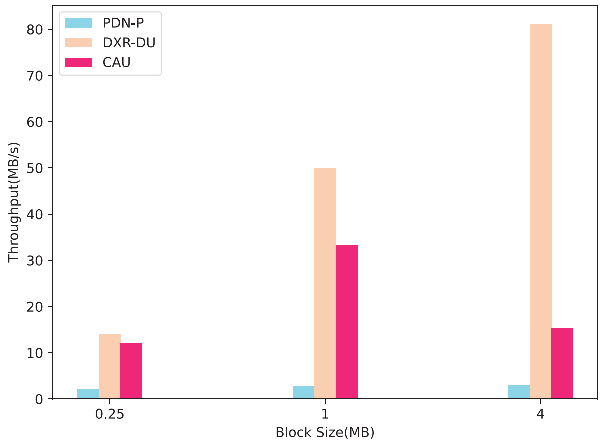
Schemes	Traces					
	rsrch_2	hm_0	hm_1	proj_0	Average	Improvement
PDN-P	0.183	0.361	0.133	0.159	0.209	89.9%
CAU	0.024	0.100	0.028	0.029	0.045	53.6%
DXR-DU	**0.021**	**0.021**	**0.023**	**0.020**	**0.021**	**-**

Table 3: Comparison on the traces with different block sizes

Scheme	rsrch_2			hm_0		
	0.25 MB	1 MB	4 MB	0.25 MB	1 MB	4 MB
PDN-P	1.00	4.00	16.00	1.00	4.00	16.00
CAU	0.06	0.16	0.44	0.12	0.40	1.54
DXR-DU	**0.02**	**0.06**	**0.18**	**0.01**	**0.03**	**0.15**
Scheme	hm_1			proj_0		
	0.25 MB	1 MB	4 MB	0.25 MB	1 MB	4 MB
PDN-P	1.00	4.00	16.00	1.00	4.00	16.00
CAU	0.14	0.33	0.66	0.08	0.28	0.98
DXR-DU	**0.05**	**0.17**	**0.40**	**0.01**	**0.06**	**0.24**

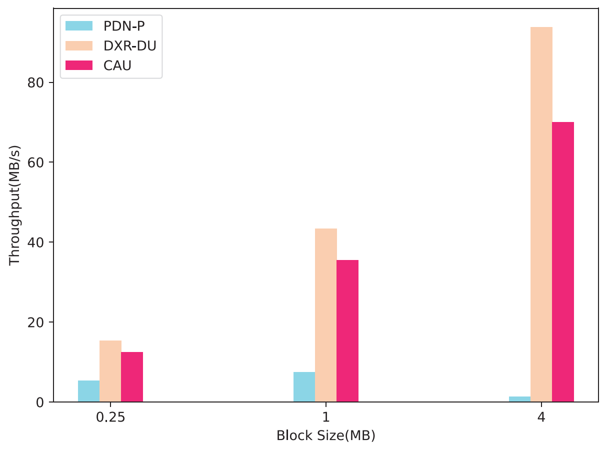


(a) wdev_3

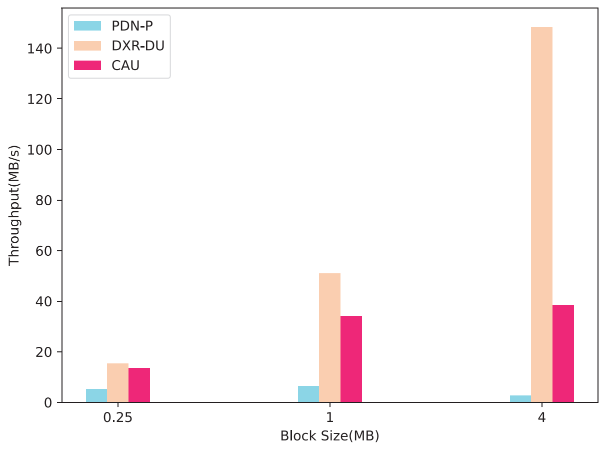


(b) hm_0

Figure 12: (Continued)



(c) hm_1



(d) proj_0

Figure 12: Comparison on the traces with different block sizes

According to our experiments, DXR-DU can reduce 44.9%–99.1% of the cross-rack update traffic on average compared to PDN-P and CAU in most cases. Meanwhile, it can dramatically improve the DU throughput by up to 53.6% when compared to CAU.

5 Conclusion

To achieve the high availability of health and medical big data in erasure-coded cloud storage systems, the data update performance in erasure coding should be continuously optimized. We perform DU performance optimization via mitigating the update traffic, especially the cross-rack traffic. Thus, we propose a rack-aware update scheme called Delta-XOR-Relay Data Update (DXR-DU) based on four valuable techniques: delta transmission, XOR, relay, and batch update. Our proposed scheme offers two selective update options: (i) data-delta-based update, when the number of updated data blocks in data rack is less than the number of parity blocks to update in parity rack, we select a parity node as a relay node for collecting the data deltas and renewing the parity blocks, and (ii) parity-delta-based update for the opposite case, where we select a relay node for each data rack to collect the local data deltas and send the parity deltas to the relevant parity nodes. Experiments on a local testbed show that DXR-DU can significantly reduce the cross-rack update traffic and improve the update throughput.

Funding Statement: We thank the anonymous reviewers for their insightful feedback. We also appreciate Wenhan Zhan for his sincere help. This work is supported by Major Special Project of Sichuan Science and Technology Department (2020YFG0460), Central University Project of China (ZYGX2020ZB020, ZYGX2020ZB019).

Conflicts of Interest: We declare that we have no conflicts of interest to report regarding the present study.

References

1. Chang, S. H., Chiang, R. D., Wu, S. J., Chang, W. T. (2016). A context-aware, interactive m-health system for diabetics. *IT Professional, 18(3),* 14–22. DOI 10.1109/MITP.2016.48.

2. Pasluosta, C. F., Gassner, H., Winkler, J., Klucken, J., Eskofier, B. M. (2015). An emerging era in the management of Parkinson's disease: Wearable technologies and the Internet of Things. *IEEE Journal of Biomedical and Health Informatics, 19(6),* 1873–1881. DOI 10.1109/JBHI.2015.2461555.

3. Hassan, M. K., El Desouky, A. I., Elghamrawy, S. M., Sarhan, A. M. (2019). Big data challenges and opportunities in healthcare informatics and smart hospitals. In: *Security in smart cities: models, applications, and challenges*, pp. 3–26. Cham: Springer.

4. Granato, D., Santos, J. S., Escher, G. B., Ferreira, B. L., Maggio, R. M. (2018). Use of principal component analysis (PCA) and hierarchical cluster analysis (HCA) for multivariate association between bioactive compounds and functional properties in foods: A critical perspective. *Trends in Food Science & Technology, 72,* 83–90. DOI 10.1016/j.tifs.2017.12.006.

5. Baker, S. B., Xiang, W., Atkinson, I. (2017). Internet of things for smart healthcare: Technologies, challenges, and opportunities. *IEEE Access, 5,* 26521–26544. DOI 10.1109/ACCESS.2017.2775180.

6. Liu, H., Guo, Q., Wang, G. L., Gupta, B. B., Zhang, C. M. (2020). Medical image resolution enhancement for healthcare using nonlocal self-similarity and low-rank prior. *Multimedia Tools and Applications, 78(7),* 9033–9050.

7. Nguyen, G. N., Viet, N. H. L., Elhoseny, M., Shankar, K., Gupta, B. et al. (2021). Secure blockchain enabled cyber-physical systems in healthcare using deep belief network with resnet model. *Journal of Parallel and Distributed Computing, 153,* 150–160. DOI 10.1016/j.jpdc.2021.03.011.

8. Kaur, M., Singh, D., Kumar, V., Gupta, B., Abd El-Latif, A. A. (2021). Secure and energy efficient-based e-health care framework for green internet of things. *IEEE Transactions on Green Communications and Networking, 5(3),* 1223–1231. DOI 10.1109/TGCN.2021.3081616.

9. Zhou, T., Tian, C. (2020). Fast erasure coding for data storage: A comprehensive study of the acceleration techniques. *ACM Transactions on Storage, 16(1),* 1–24. DOI 10.1145/3375554.

10. Copeland, M., Soh, J., Puca, A., Manning, M., Gollob, D. (2015). *Microsoft azure*. New York, NY, USA: Apress.

11. Verma, C., Pandey, R. (2018). Comparative analysis of GFS and HDFS: Technology and architectural landscape. *2018 10th International Conference on Computational Intelligence and Communication Networks (CICN)*, Denmark, IEEE.

12. Muralidhar, S., Lloyd, W., Roy, S., Hill, C., Lin, E. et al. (2014). f4: Facebook's warm {BLOB} storage system. *11th USENIX Symposium on Operating Systems Design and Implementation*, Broomfield, CO.

13. Xie, X., Wu, C., Gu, J., Qiu, H., Li, J. et al. (2019). AZ-code: An efficient availability zone level erasure code to provide high fault tolerance in cloud storage systems. *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, Santa Clara, CA.

14. Xiong, H., Yang, M., Yao, T., Chen, J., Kumari, S. (2021). Efficient unbounded fully attribute hiding inner product encryption in cloud-aided WBANs. *IEEE Systems Journal,* 1–9. DOI 10.1109/JSYST.2021.3125455.

15. Xiong, H., Chen, J., Mei, Q., Zhao, Y. (2020). Conditional privacy-preserving authentication protocol with dynamic membership updating for vanets. *IEEE Transactions on Dependable and Secure Computing, 19(3),* 2089–2104.

16. Shen, Z., Lee, P. P. C. (2018). Cross-rack-aware updates in erasure-coded data centers. *ICPP 2018: Proceedings of the 47th International Conference on Parallel Processing*, pp. 1–10. DOI 10.1145/3225058.3225065.

17. Zhang, F., Huang, J., Xie, C. (2012). Two efficient partial-updating schemes for erasure-coded storage clusters. *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*, Xiamen, China, IEEE.

18. Pei, X., Wang, Y., Ma, X., Xu, F. (2016). T-Update: A tree-structured update scheme with top-down transmission in erasure-coded systems. *35th Annual IEEE International Conference on Computer Communications*, San Francisco, CA, IEEE.

19. Wang, F., Tang, Y., Xie, Y., Tang, X. (2019). Xorinc: Optimizing data repair and update for erasure-coded systems with xor-based in-network computation. *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, Santa Clara, CA, IEEE.
20. Xiao, Y., Zhou, S., Zhong, L. (2020). Erasure coding-oriented data update for cloud storage: A survey. *IEEE Access, 8,* 227982–227998. DOI 10.1109/Access.6287639.
21. Chan, J. C., Ding, Q., Lee, P. P., Chan, H. H. (2014). Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. *12th USENIX Conference on File and Storage Technologies*, Santa Clara, CA.
22. Gong, G., Shen, Z., Wu, S., Li, X., Lee, P. P. (2021). Optimal rack-coordinated updates in erasure-coded data centers. *IEEE Conference on Computer Communications*, *Virtual Conference,* IEEE.
23. Reed, I. S., Solomon, G. (1960). Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics, 8(2),* 300–304. DOI 10.1137/0108018.
24. Apache hadoop 3.0.0 (2017). https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html.
25. Chen, H., Zhang, H., Dong, M., Wang, Z., Xia, Y. et al. (2017). Efficient and available in-memory kv-store with hybrid erasure coding and replication. *ACM Transactions on Storage, 13(3),* 1–30.
26. Vajha, M., Ramkumar, V., Puranik, B., Kini, G., Lobo, E. et al. (2018). Clay codes: Moulding MDS codes to yield an MSR code. *16th USENIX Conference on File and Storage Technologies (FAST 18)*, Oakland, CA, USENIX Association. https://www.usenix.org/conference/fast18/presentation/vajha.
27. Huang, C., Simitci, H., Xu, Y., Ogus, A. W., Calder, B. et al. (2012). Erasure coding in windows azure storage. *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA, USENIX. https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang.
28. Linux traffic control (2001). https://man7.org/linux/man-pages/man8/tc.8.html.
29. Proxmox (2021). https://www.proxmox.com/en/proxmox-ve.