Received May 31, 2021, accepted June 18, 2021, date of publication June 29, 2021, date of current version July 7, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3093450

QLOC: Quorums With Local Reconstruction Codes

ANWITAMAN DATTA [1], ADAMAS AQSA FAHREZA [1], AND FRÉDÉRIQUE OGGIER [2]
[1]School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798
[2]Division of Mathematical Sciences, Nanyang Technological University, Singapore 639798

Corresponding author: Anwitaman Datta (anwitaman@ntu.edu.sg)

The work of Anwitaman Datta and Adamas Aqsa Fahreza was supported by the Ministry of Education (MoE), Singapore, through the Academic Research Fund Tier 1 for the project titled 'StorEdge: Data store along a cloud-to-thing continuum with integrity and availability' under Project 2018-T1-002-076.

This work did not involve human subjects or animals in its research.

ABSTRACT In this paper we study the problem of consistency in distributed storage systems relying on erasure coding for storage efficient fault-tolerance. We propose QLOC - a flexible framework for supporting the storage of warm data, i.e., data which, while not being very frequently in use, nevertheless continues to be accessed for reads or writes regularly. QLOC builds upon (1) a generic family of local reconstruction codes with guarantees in terms of fault-tolerance, efficient recovery from failures and degraded mode operations, and can be instantiated with parameters customized to requirements such as storage overhead and reliability dictated by user needs and operational environments, and (2) quorum-based consistency mechanisms with support for read-modify-write operations without any underlying atomic primitives, providing deployment choices trading-off fault-tolerance, consistency and concurrency requirements. We carry out a theoretical analysis of the code properties, and experimentally benchmark the performance of the consistency enforcement mechanisms, demonstrating the practicality of the proposed approach.

INDEX TERMS Consistency, local reconstruction erasure codes, quorum system, read-modify-write.

I. INTRODUCTION

Erasure codes have become an integral part of the storage stack for storing humongous volumes of data. For example, Windows Azure Storage [1], Facebook's f4 [2], Google File System Colossus [3], Baidu's Atlas [4], Cloudera's Apache HDFS [5] all adopted erasure codes in their architecture. This embrace of erasure coding based redundancy is owing to the significant savings of raw storage for a targeted level of fault tolerance, as compared to a replication based system. This is achieved by taking several data blocks and computing linear combination of these blocks to generate the redundancy (we call them parity blocks or symbols), such that, as long as a suitable (determined by the design of the code) subset of the original data and parity blocks are available, the missing original data blocks can be recreated. The choice of these linear combinations also impacts how the system works when in a **degraded state**, i.e., when some storage nodes become unavailable but the number of node failures is within the threshold of fault-tolerance:

(1a) Degraded reads: If a given data object itself is not available, then one needs to recompute the data item by accessing other data items and parity blocks. This process of 'degraded reads' incurs computation costs, network usage and storage I/O operations which can be significantly higher than in a replicated storage system. In the latter, as long as a copy of the data is available, it can be directly read. Write operations that rely on read operations naturally have a higher cost too.

(1b) Repair costs: The system may determine that one of the original data blocks, or a parity block, needs to be replenished for long term durability, e.g., because the corresponding storage node is deemed to have permanently failed. Such a repair process is similarly expensive, in comparison to a replication based system which needs to access and transfer only one block of data, with no added computational costs.

Erasure codes for distributed storage typically mitigate (1a-b) by distinguishing "local" from "global" parities: local parities are computed from a few data blocks, they are easier to read in degraded mode or repair than global parities, which involve many or all data blocks, and are mostly useful when it comes to simultaneously repair several failures. A concrete example of such erasure codes is the class of Local Reconstruction Codes (LRC) used in Windows's Azure system [1] and its precursors, Pyramid codes [6]. This paper considers storage systems using an erasure code with both local and

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Marozzo.

semi-global parities, fitting the framework of local reconstruction codes.

While the problems of repair and operations in a degraded state have effective mitigations, which have been deployed in commercial systems, management of **mutable content** with erasure coding is in relative nascence, and needs mechanisms compatible for erasure coded data, which are not immediately inherited from techniques available for replicated data:

(2a) Update detection and propagation: For replicated data, hash digest of the data can be used to detect differences among replicas, and likewise reconciliation of the versions can be carried out by just transferring and replacing content with the differences (a popular technique is the Merkle tree [7] based anti-entropy algorithm). These techniques do not immediately extend to erasure coded data, where individual parity blocks carry information about multiple data blocks.

(2b) Consistency management: If multiple processes concurrently read/modify a given data object, they may be exposed to different values for the same data. When there are multiple copies of the same data (replication), the issue of handling consistency gets even more complicated. Multiple techniques (e.g., based on locks and quorums [8], primary/secondary replicas) for handling consistency for replicated data exist. They cannot be applied as is for coding based redundancy.

Given challenges such as (2a-b), real-world large-scale deployments often relegate erasure codes for "cold" storage, that is the storage of non-mutating content, e.g., for archival and append-only data, or for storing data with support for limited update semantics (e.g., Facebook's f4 [2] supports deletions). While it makes sense to prefer replication for "hot" data that is frequently accessed and manipulated, we argue that the volume of "warm" data (infrequently accessed but manipulated, typically "older" data), continuously grows over time and thus would benefit from an erasure coding based storage with support for mutations and consistency guarantees.

These considerations determine **the objectives of this work** — design techniques to support the storage of mutable data by providing consistency guarantees while also leveraging the benefits of storage efficiency and state-of-the-art solutions for repair and degraded operations as afforded by local reconstruction codes.

A. SYSTEM ARCHITECTURE & SYSTEM MODEL

We assume a back-end storage system (shown in Figure 1) which dichotomizes the replication-based storage of the frequently accessed hot data from the erasure code-based storage of the infrequently accessed warm data. This dichotomy, and use of replication versus erasure coding across hot and warm data respectively has been deemed practical for data-center scale storage systems, exemplified, for instance, by Facebook's Haystack [9] and f4 [2] systems. Data meant for archival, which is infrequently read and never edited (but would typically be expected to support data deletion) after

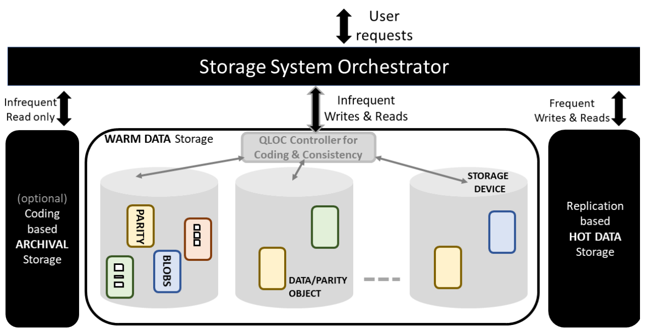


FIGURE 1. **A modular storage architecture for storing the continuum of hot, warm and (optionally) cold data: Our focus is the warm data store, which uses local reconstruction coding based redundancy for storage efficient fault tolerance, and guarantees consistency using quorums.**

creation may optionally also be stored in a third storage subsystem.

We furthermore assume a storage system orchestrator tasked to determine the category to which a specific data object ought to belong and accordingly coordinate the migration of a given data item among hot, warm and cold (archival) storage subsystems, and route access requests accordingly.

Our work is agnostic to the mechanism of how the orchestrator determines which data should be placed in a particular subsystem, or what kind of storage medium - hard-disks or SSDs - may be suitable to meet specific performance requirements within the subsystems, or the impact of erasure codes on the medium of choice (see [10] for a study on the reliability impact of erasure coding on SSD based storage).

We recognize that erasure codes are already being used for storing warm data, where access frequency is expected to be significantly lower than the case for hot data (and thus, use of erasure coding is deemed practical), and identify a gap in terms of the kinds of operations (particularly, read-modify-write operations) and strong consistency semantics (for example, sequential consistency) that existing techniques cannot support in conjunction, and accordingly we focus on how the erasure coded warm data stores can be augmented with these capabilities, so that a richer set of applications can be deployed over erasure coded storage systems.

To that end, we assume that coding is carried out at a granularity of fixed sized data objects (also called data blocks or symbols): each data object stores multiple arbitrary sized Binary Large OBjects (BLOBs). This data object abstraction is similar to the abstraction of 'chunk' in Google's GFS [11] or 'volume' in Facebook's f4 [2]. The storage system orchestrator determines in which subsystem to place individual BLOBs, and likewise, how to route access requests. Within the warm data store, the controller furthermore needs to handle the placement and subsequent location of the BLOBs within the data objects, coding of the data objects (a logical group over which erasure coded parities -"parity blocks/symbols"- are computed is formed by choosing data objects from different physical devices), the location of the data objects and corresponding parities within the storage pool, the optimization of space utilization, and reconstruction of the latest version of the data objects if necessitated by storage node failures. The resulting parities are stored in further other devices. This placement strategy avoids correlated failures of data objects due to co-location. The same idea of physical dispersal across different racks may also be carried out [2], [11], [12].

We note that unrelated BLOBs being grouped together, yields random access requests - both at the logical level among data objects, as well as among physical storage resources, so that accesses are independent. This can have performance benefits, since this naturally leads to independent operations, and fewer chance of consistency conflicts to be dealt with. This last assumption should be deemed optional and as a recommendation given its implications on the system's performance, however, it has no bearings on the correctness of the consistency mechanisms we propose. In fact, we will demonstrate that one particular variant of our mechanism (exploiting semi-global parities for the quorum mechanism) can guarantee a strong form of sequential consistency at the level of the group of coded objects, demonstrating the benefit of considering the proposed form of semi-global parities beyond their immediate impact of improved fault tolerance.

B. CONTRIBUTIONS AND STRUCTURE

The contributions in this work are two pronged. We build on the existing literature on local reconstruction codes to propose (i) a novel instance of highly fault-tolerant yet storage efficient code family, attuned to the (ii) design of flexible quorum mechanisms which help navigate trade-offs of consistency and concurrency over a collection of coded data objects, while supporting read-modify-write operations without any underlying atomic primitives.

The contributions in terms of erasure code design and analysis are found in Section III:

(1) A parameterized family of local reconstruction codes, which uses local and semi-global parities, whose instances achieve practical storage overheads, e.g. $1.32 - 1.875\times$ and low single repair costs e.g., $4-10$ blocks for coded collections with $30 - 132$ blocks.

(2) A rigorous analysis of fault-tolerance of a subset of practical instances, one using solely local parities having a lower bound of 2-fault tolerance, and another variation using semi-global parities that incurs marginally higher storage overhead but with a lower bound of 5-fault tolerance. The establishment of such a bound on fault-tolerance is harnessed to theoretical estimate of the resilience of the code instances under real-world data-center environments.

Recall that our ultimate objective is to devise a mechanism to support consistency, rather than create another locally repairable code optimized for repairability itself, for which a very rich literature already exists. Consequently, we base our approach on existing designs that have been demonstrated to be robust in terms of repairability, instead of creating a totally new design from scratch. In particular, we inherit and then adjust the design from [6], [13]. The proposed adjustments (in the choice of how the semi-global parities are created) are driven to create dependencies among the coded blocks of data, which enable us to achieve specific desirable behaviors which are in turn exploited in our quorum systems design (discussed next), while preserving the robust repairability property, while also paying attention to keep the storage overhead of the codes within ranges that have been deemed acceptable in real-world deployments.

A rich body of work on locally repairable codes exist, starting from the early pioneering works on the topic [1], [6], [14], and it continues to be a topic of research on its own accord, e.g., there are recent works [15], [16] which consider repairability of codes with similar array structures as used in this work. In that context, the current work should be viewed complementary in nature, and as adding a new design dimension laying foundations for further future work, on the amenability of the structures of these other existing repairable codes, to be leveraged for developing quorum systems for consistency, and to be evaluated in terms of the kind of consistency, fault-tolerance and concurrency that can be achieved by the quorum systems for these other instances of code families.

The design and analysis of the quorum systems over the proposed family of locally repairable code instances are presented in Section IV:

(3) Two quorum variants, one using only the local parities and the other using also the semi-global parities are proposed.

(4) The design choices of whether to deploy the system with semi-global parity, and even if they are deployed, on whether to use the semi-global parities for the quorum system, or use them solely for fault-tolerance lead to three possible deployment modes, meeting different fault-tolerance, concurrency and consistency guarantees (sequential consistency, and a weaker variant we christen semantic consistency following the concept semantic reconciliation from [12]).

II. RELATED WORKS & BACKGROUND

Works as early as from 2000 discussed the possible benefits of using erasure coding for fault-tolerance in storage systems [17]. They mostly used maximum distance separable (MDS) codes, where $n - k$ parity blocks are each computed using k data blocks, and come with the guarantee that one can recover the original data symbols as long as any k among the n blocks are available. However it took time for codes designed specifically for distributed storage systems to get traction, starting from the early work [6] and later [14] which argued that rather than looking at erasure codes as a black-box, one can instead consider parities involving different number of data symbols: a small number of data symbols makes a ''local'' parity which can be read and used to repair data symbols faster and using less network and storage I/O resources, while a large number of data symbols can be used to create ''global'' parities which provide better fault tolerance per unit of storage space. This naturally leads to a trade-off involving the degree and number of local parities and global ones, versus the storage overhead and fault tolerance of the code. See e.g. [18], [19] for surveys on storage specific coding techniques and bounds.

Contemporaneously, mechanisms to update parities efficiently were studied [20]–[23]. A key idea in these works is to propagate the differential of the value of a mutating data block to the parity nodes and use only that to incrementally update the parities instead of recomputing the parities from scratch. We are not concerned with re-designing such an algorithm, rather we use this efficient update technique as is. We focus on the issue of guaranteeing consistency, a topic less investigated.

In the context of replication, consistency refers to a setting where read and write operations are performed on shared replicas by different processes, yet there should be a "consistent" view of the system, as if a single process were accessing and modifying a single copy of the data. This is achieved by fixing a set of rules the processes obey when they want to read or write the data, so that when one replica is updated by one of the processes, it ensures that the other copies are updated accordingly.

There are many definitions of consistency: strict consistency guarantees that the operations follow a wall-clock time ordering, sequential consistency only requires some global ordering of the operations. Other forms of consistency include causal and eventual consistencies, see e.g. [24, Section 7].

Likewise, different write operational semantics are possible. Consider a data object x with the value x_t at time t. If different processes update based on its current value x_{t_1}, they would compute $x_{t_2} = f(x_{t_1})$ and $x_{t'_2} = f'(x_{t_1})$. If the constraint is that every process in the overall system concurs on a unique globally agreed sequence of values for x, then either $[x_{t_1}, x_{t_2} = f(x_{t_1}), x_{t'_2} = f'(x_{t_1})]$ or $[x_{t_1}, x_{t'_2} = f'(x_{t_1}), x_{t_2} = f(x_{t_1})]$ would satisfy the sequential consistency condition. However, in this scenario, both the second as well as the third values are determined based on the first value, and the third value is independent of the second value. This semantics is termed as atomic multi-reader multi-writer (MRMW) register. Though this operation semantics is adequate for certain applications, it is inappropriate for database like transactional semantics [25].

In contrast, a stronger form of **atomic read-modify-write** semantics requires a sequence of globally agreed changes ensuring that the third value is determined based on the second value in the sequence, and as such, either of $[x_{t_1}, x_{t_2} = f(x_{t_1}), x_{t'_3} = f'(x_{t_2}) = f'(f(x_{t_1})]$ or $[x_{t_1}, x_{t'_2} = f'(x_{t_1}), x_{t_3} = f(x_{t_2}) = f(f'(x_{t_1}))]$ would be agreeable valid outcome sequences. For this to hold, the operation creating the third value in the sequence is blocked until the previous operation is completed. The work presented in this paper looks at how to achieve this stricter write semantics for erasure coded data, and we propose doing so using quorums and locks.

We take a quorum system supported locking based approach [8, Def 3.4] to enable *read-modify-write* operations and propose techniques for achieving *sequential consistency* as well as a weaker form of consistency which we call *semantic consistency*. Our quorum system is designed by exploiting the nuances of a bespoke local reconstruction coding family we propose, allowing us to explore the trade-offs between (i) the system's fault-tolerance, (ii) concurrency of processing read/write requests for different data objects that are coded together, and (iii) the form of consistency guaranteed.

A complementary approach for consistency, apart quorum systems, is the class of primary-based protocols, where each data block has an associated primary, which is responsible for coordinating its operations. Above mentioned works [22], [23] where updates are disseminated in a best effort manner without well-defined consistency guarantees use primary based protocols. A third option is the adaptation of the Paxos family of protocols [26] for solving consensus algorithms for erasure coded data, see e.g. [27], [28]. Coded Atomic Storage (CAS) [29], [30] aim at mimicking shared memory MRMW atomic register abstraction for coded data, with an emphasis on reducing communication cost. Several works furthermore rely on storing distinct versions [29], [31], [32] rather than carrying out updates in-place. They naturally incur much higher storage overhead in order to store the older versions. None of the existing related works support the read-modify-write operational semantics over erasure coded data.

III. LOCAL RECONSTRUCTION CODES

A. CONSTRUCTION

Let d_1, d_2 and r be positive integers. Suppose that we have a $(4d_1d_2+r, 4d_1d_2)$ systematic code, that is a code that encodes $k = 4d_1d_2$ data symbols into $n = k+r = 4d_1d_2+r$ symbols by adding r parities. We assume every parity in this code is a linear combination of all data symbols. Since $k = 4d_1d_2$, we may arrange the $4d_1d_2$ data symbols in a $2d_1 \times 2d_2$ logical square grid, as shown on Figure 2, and label the data symbols as $u_{i,j}$, $i = 1, \ldots, 2d_1$, $j = 1, \ldots, 2d_2$ to attribute a logical position of each data symbol in the grid. Then row i of the grid contains $(u_{i,1}, u_{i,2}, \ldots, u_{i,2d_2})$. The r parities are computed using an $r \times k$ matrix G as

$$\underbrace{\begin{bmatrix} g_{1,1} & \cdots & g_{1,k} \\ \vdots & & \vdots \\ g_{r,1} & \cdots & g_{r,k} \end{bmatrix}}_{G} \begin{bmatrix} u_{1,1} \\ \vdots \\ u_{2d_1,2d_2} \end{bmatrix} = \begin{bmatrix} p_1(u_{1,1}, \ldots, u_{2d_1,2d_2}) \\ \vdots \\ p_r(u_{1,1}, \ldots, u_{2d_1,2d_2})) \end{bmatrix} \tag{1}$$

A local reconstruction code is obtained from the above *base* code by computing parities as follows:

(i) Take row i, keep data symbols $(u_{i,1}, u_{i,2}, \ldots, u_{i,2d_2})$ on this row and set all other data symbols to be 0. Then use (1) to compute r_1 $(r_1 \leq r)$ parities corresponding to the data symbols from this single row:

$$p_1(u_{i,1}, u_{i,2}, \ldots, u_{i,2d_2}), \ldots, p_{r_1}(u_{i,1}, u_{i,2}, \ldots, u_{i,2d_2}).$$

We use the convention that data symbols that are always set to 0 are not mentioned as inputs to the parities. Since there are $2d_1$ rows, this yields a total of $2d_1r_1$ parities, which we call **row parities**. In Figure 2, the r_1 parities obtained by setting $(u_{j,1}, u_{j,2}, \ldots, u_{j,2d_2}) = 0$ for all $j \neq i$ are shown on row i. This gives a logical $2d_1 \times r_1$ grid containing the row parities.

(ii) We repeat the same process for the columns, to obtain $2d_2r_2$ **column parities**, and in turn an $r_2 \times 2d_2$ logical grid of column parities. The column parities for column j are

$$p_{r_1+1}(u_{1,j}, u_{2,j}, \ldots, u_{2d_1,j}), \ldots, p_{r_1+r_2}(u_{1,j}, \ldots, u_{2d_1,j}).$$

We consciously avoided using parities $p_1, \ldots, p_{r_1}$ that were already used for the row parities, which results in the further constraint $r_1 + r_2 \leq r$. One could a priori decide to repeat parities, but we will infer from our analysis of the degraded mode that it is best not to (see Corollary 1).

(iii) Finally, consider the original grid containing the data symbols being divided as four quadrants Q_1, Q_2, Q_3, Q_4 of $d_1 \times d_2$ nodes each. For each pair of these quadrants, we set the data symbols contained in the two other quadrants to be zeroes, and repeat the parity computations, giving $\binom{4}{2}r_3 = 6r_3$ semi-global parities, which we call **quadrant parities**. To avoid parities used for either row or column parities (see Proposition 1), and to generate the six quadrant parities each using distinct parities from the base code, we need $r_1 + r_2 + 6r_3 \leq r$. The quadrant parities are also shown on Figure 2, and while there are $6r_3$ of them, there is no specific dimension (number of rows and columns) attached to the corresponding logical grid.

Remark 1: The layout of data and parities in grids is a logical abstraction decoupled from their actual physical placement. Furthermore, codes similar to the case when $r_3 = 0$ have been previously studied [6], [13], which provide a track-record of practicability, that our proposed generalization inherits.

B. PROPERTIES

1) RATE

Adding $2(d_1r_1 + d_2r_2) + 6r_3$ parities to the $k = 4d_1d_2$ data symbols results in a code rate of

$$\frac{k}{n} = \frac{4d_1d_2}{4d_1d_2 + 2(d_1r_1 + d_2r_2) + 6r_3}.$$

Table 1 explores the parameter choices in terms of the code properties: k, n, the storage overhead n/k and its reciprocal, the code rate. The last column shows the number $\min(2d_1, 2d_2) < k$ of blocks (data and parity) that need to be accessed to rebuild information when a single storage node is missing. In comparison, in a traditional maximum distance separable (MDS) coded system where every parity is a combination of all k data symbols, k data accesses and transfers are needed to read or reconstruct a single missing block. Many practical systems prefer a moderate value of k in and around 10. For instance Facebook's f4 [2] uses a MDS Reed-Solomon code [33] with $k = 10$. Because of the local repairability property, we can consider larger values of k without compromising on performance. Likewise, a storage overhead in the range of $\approx 1.4 - 2\times$ is considered practical. For instance, f4's Reed-Solomon code has an overhead of $1.4\times$ which combined with cross-object redundancy results in an ultimate $2.1\times$ storage overhead. These industry validated system parameter choices inform our choice of tabulated parameters. We see that a wide range of choices is feasible. We next explore the practical implications of the code design, elaborated with an example instance (highlighted in the table).

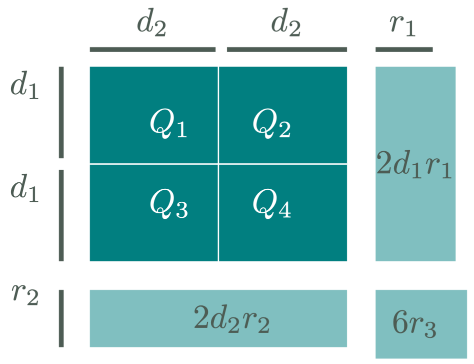
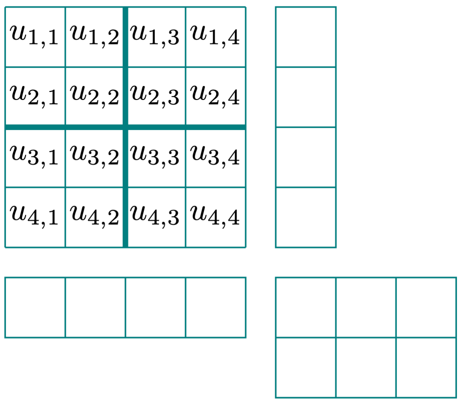


FIGURE 2. The grid layout for a code encoding $k = 4d_1d_2$ data symbols by adding $2d_1r_1$ row parities, $2d_2r_2$ column parities, and $6r_3$ quadrant parities.



$u_{1,1}$ $u_{1,2}$ $u_{1,3}$ $u_{1,4}$
$u_{2,1}$ $u_{2,2}$ $u_{2,3}$ $u_{2,4}$
$u_{3,1}$ $u_{3,2}$ $u_{3,3}$ $u_{3,4}$
$u_{4,1}$ $u_{4,2}$ $u_{4,3}$ $u_{4,4}$

FIGURE 3. The logical grid layout for a (30,16) code.

TABLE 1. Code parameter choices.

Design parameters		Code properties			Single repair
d_1, d_2	r_1, r_2, r_3	(n, k)	n/k	rate k/n	$2\min(d_1, d_2)$
1, 1	1, 1, 1	(14,4)	3.5	≈ 0.285	2
2, 2	1, 1, 1	(30,16)	1.875	≈ 0.533	4
3, 2	1, 1, 1	(40,24)	≈ 1.666	0.6	4
3, 3	1, 1, 1	(54,36)	1.5	≈ 0.666	6
4, 2	1, 1, 1	(50,32)	1.5625	0.64	4
5, 3	1, 1, 1	(82,60)	≈ 1.366	≈ 0.731	6
2, 2	1, 1, 2	(36,16)	2.25	≈ 0.444	4
3, 3	1, 1, 2	(60,36)	≈ 1.666	0.6	6
4, 4	1, 1, 2	(92,64)	1.4375	≈ 0.695	8
5, 5	1, 1, 2	(132,100)	1.32	≈ 0.757	10

Example 1: Consider that $d_1 = d_2 = 2$. We then have a $(16 + 4(r_1 + r_2) + 6r_3, 16)$ code, which would be a $(30, 16)$ code for $r_1 = r_2 = r_3 = 1$, for a rate of $8/15 \approx 0.533$ and a corresponding storage overhead of $1.875\times$. The logical grid layout for this code is illustrated in Figure 3. We illustrate the computation of row

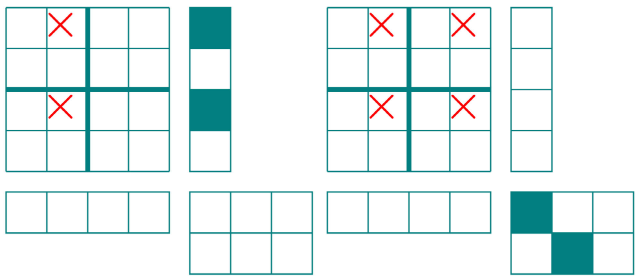


FIGURE 4. **Degraded reads for a (30,16) code. On the left, if the data symbol $u_{1,2}$ in position $(1, 2)$ is unavailable, either the row parity in row 1 or the column parity in column 2 could be used for degraded reads. If however two data symbols are available in column 2, say $u_{1,2}$ and $u_{3,2}$, then the column parity becomes useless, and both row parities 1 and 3 are used. On the right, none of the row/column parities can be used alone for recovery from this pattern of failures.**

parities on this example. We need an MDS code that maps $u_{1,1}, \dots, u_{4,4}$ to r parities (we need $r \geq 8 = r_1 + r_2 + 6r_3$). The first parity $p_1(u_{1,1}, \dots, u_{4,4})$ of the MDS code is computed from (1), based on which we compute the 4 row parities: $p_{1,1} := p_1(u_{1,1}, u_{1,2}, u_{1,3}, u_{1,4}) = g_{1,1}u_{1,1} + g_{1,2}u_{1,2} + g_{1,3}u_{1,3} + g_{1,4}u_{1,4}$, and similarly $p_{1,2} := p_1(u_{2,1}, u_{2,2}, u_{2,3}, u_{2,4})$, $p_{1,3} := p_1(u_{3,1}, u_{3,2}, u_{3,3}, u_{3,4})$, $p_{1,4} := p_1(u_{4,1}, u_{4,2}, u_{4,3}, u_{4,4})$, each obtained by keeping one row of data symbols, and setting the other rows to 0.

We repeat the same procedure to compute the column parities, using the second parity of the MDS code to get $p_{2,1} := p_2(u_{1,1}, u_{2,1}, u_{3,1}, u_{4,1})$, $p_{2,2} := p_2(u_{1,2}, u_{2,2}, u_{3,2}, u_{4,2}) = g_{2,2}u_{1,2} + g_{2,6}u_{2,2} + g_{2,10}u_{3,2} + g_{2,14}u_{4,2}$, $p_{2,3} := p_2(u_{1,3}, u_{2,3}, u_{3,3}, u_{4,3})$, $p_{2,4} := p_2(u_{1,4}, u_{2,4}, u_{3,4}, u_{4,4})$. Here, we showed explicitly the computation of the second column parity, since we will refer back to it in exposing some other aspect of the code's property in the following text.

2) RESILIENCE AGAINST ERASURES ON DATA SYMBOLS

If a failure affects the data symbol $u_{i,j}$ located in row i and column j of the logical grid, a degraded read can be performed using either one row parity or one column parity, along with the other data symbols in the row or column respectively. More generally, if there are r_1 row parities, up to r_1 data symbol failures can be recovered using row parities only and other data symbols from the given row, and similarly, up to r_2 data symbol failures are at most recoverable using only the column parities and other data symbols in the corresponding column. When $(1 + r_1)(1 + r_2)$ failures are aligned on a $(1 + r_1) \times (1 + r_2)$ subgrid of the logical grid, none of the rows/columns can be used independently to recover from any of the failures. However they may be combined to create an $(1 + r_1)(1 + r_2) \times (1 + r_1)(1 + r_2)$ matrix that enables the recovery of the unavailable data symbols. This is illustrated for a 2×2 failure cluster in Figure 4 for the above Example 1 coding scenario. For this instance of subgrid failure cluster, the system of linear equations that need to be solved for effectuating the data recovery is:

$$\begin{bmatrix} g_{1,2} & g_{1,4} & 0 & 0 \\ 0 & 0 & g_{1,10} & g_{1,12} \\ g_{2,2} & 0 & g_{2,10} & 0 \\ 0 & g_{2,4} & 0 & g_{2,12} \end{bmatrix} \begin{bmatrix} u_{1,2} \\ u_{1,4} \\ u_{3,2} \\ u_{3,4} \end{bmatrix} = \begin{bmatrix} \tilde{p}_{1,1} \\ \tilde{p}_{1,3} \\ \tilde{p}_{2,2} \\ \tilde{p}_{2,4} \end{bmatrix} \quad (2)$$

where $\tilde{p}_{i,j}$ denote the available part of the parity equation obtained from $p_{i,j}$, e.g. $\tilde{p}_{1,1} = p_{1,1} - g_{1,1}u_{1,1} - g_{1,3}u_{1,3}$, $\tilde{p}_{2,2} = p_{2,2} - g_{2,6}u_{2,2} - g_{2,14}u_{4,2}$. Now (2) has a unique solution if and only if the determinant of the recovery matrix is non-zero:

$$g_{1,4} \cdot g_{1,10} \cdot g_{2,2} \cdot g_{2,12} - g_{1,2} \cdot g_{1,12} \cdot g_{2,4} \cdot g_{2,10} \neq 0$$

This constraint is obtained from the specific configuration of the 4 node failures. A general expression is derived for any arbitrary group of 4 failures among the data nodes when $r_1 = r_2 = 1$ and d_1, d_2 are arbitrary (in our example $d_1 = d_2 = 2$), when these 4 failures share the same rows (say i and j) and columns (say k and l), giving the following set of constraints:

Lemma 1: If $r_1 = r_2 = 1$, a pattern of 4 data symbol erasures in positions (i, l), (j, k), (i, k), (j, l) is recoverable using only row and column parities if and only if $\forall i, j \in \{1, \dots, d_1\}$, $k, l \in \{1, \dots, d_2\}$, $(i, l) \neq (j, k)$

$$g_{1,4(i-1)+l} \cdot g_{1,4(j-1)+k} \cdot g_{2,4(i-1)+k} \cdot g_{2,4(j-1)+l} - g_{1,4(i-1)+k} \cdot g_{1,4(j-1)+l} \cdot g_{2,4(i-1)+l} \cdot (g_{2,4(j-1)+k}) \neq 0. \quad (3)$$

The i^{th} row k^{th} column node is the $4(i-1)+k^{th}$ symbol in our example, for the computation of any parity as per (1). The term $g_{1,4(i-1)+l}$ corresponds to the coefficient of a data symbol in any position (i, l). For the next coefficient $g_{1,4(j-1)+k}$, the second lost data symbol can be in any position in a distinct row and distinct column, that is $i \neq j$ and $j \neq l$. The remaining two positions are then determined by the subgrid layout.

Corollary 1: If a unique parity from the base coding scheme is used to compute both the row and column parities, i.e., $g_{2,*} = g_{1,*}$, then the constraints in (3) are violated by default.

This justifies the choice of distinct parities from the base coding scheme to compute the row and column parities.

Existence of suitable coefficients $g_{i,j}$ satisfying (3) depends on the alphabet: over a binary alphabet $\{0, 1\}$, the condition will not be satisfied. A concrete instance, satisfying (3) and further constraints as discussed below, for instantiating a (30, 16) code depicted in Example 1, is given in the Appendix.

This grid-like failure pattern sharing two rows and two columns among the data nodes is the worst case scenario from a data recoverability point of view, for 4 data nodes failures. Any other 4 failures pattern will necessarily have some failures which can be recovered using only a row or a column parity and the other data blocks in the same row or column, thus improving the 'health' of the system to a 3 data nodes failures state, which in turn is always recoverable. The 3 data node failures may happen in various patterns, but the most complicated scenario (where recoveries may need sequential execution) is a ‘L’ shaped layout, where there are two failures each in same row and same column (one failure being in the intersection of these groups). In that case, the nodes not in the intersection can be individually rebuilt first, before recovering the one in the intersection. In summary:

Proposition 1: The configuration with one set of row parities and one set of column parities can recover from at least any five simultaneous data symbol erasures.

Indeed, suppose that five data symbols are erased simultaneously. Even if any four of those five form a subgrid cluster, and even if the fifth erasure shares either a row or column with any of the other erasures, it can still be recovered using a column or row parity. This will revert the system to a four failure pattern, which has already been shown to be recoverable.

3) GENERAL FAULT TOLERANCE

Above, all erasures are assumed to be confined to data symbols. If we relax the condition, and consider erasures across data and parity symbols, then for $r_1 = r_2 = 1$ and $r_3 = 0$ (i.e., if there are no quadrant parities), we notice that if there are three failures affecting a data symbol and the two parity symbols sharing its row and column, then this data symbol and the parity symbols all become unrecoverable - thus reducing the general fault tolerance to two arbitrary erasures, down from the five erasures of only data symbols as observed above (Proposition 1). Any individual increment in r_1 or r_2 can be used to improve this tolerance by one, but at the minimal cost of increasing the storage cost by $\min(d_1r_1, d_2r_2)$. That is the motivation behind the use of ‘semi-global’ parities, where $k/2 = 2d_1d_2$ data symbols are used to generate individual parities, where the data symbols are picked as per the two quadrant groupings. This leads to an increase of storage overhead equivalent to six symbols, but significantly improves the general fault tolerance (lower bounded by five failures), as we discuss next. It might also improve the fault tolerance of erasures exclusively confined to data symbols, and as such Proposition 1 provides a pessimistic lower bound. We will show next a similar lower bound for a general failure pattern, possibly affecting any nodes, irrespective of whether it stores a data symbol or a parity of any kind.

Proposition 2: The configuration of $r_1 = 1$ row parities, $r_2 = 1$ column parities, and a set of six quadrant parities ($r_3 = 1$) guarantee recovery of all data from at least any five simultaneous erasures, irrespective of the combination of data and parity nodes involved.

We start with the ‘L’ configuration where a data symbol and its row and column parities are erased. Three out of the six quadrant parities still contain information about this data node. As long as any one of them is still available (which is the case if there are five arbitrary erasures in the whole system), the original data is recoverable, e.g., pattern ⓐ in Figure 5.

For more data symbol erasures, various cases arise:

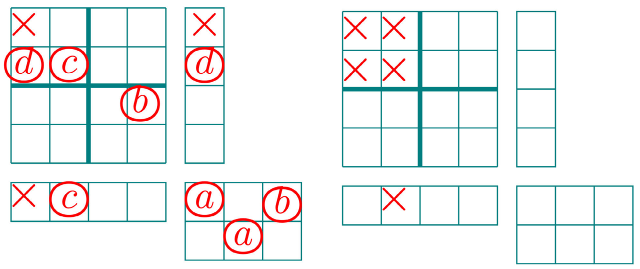


FIGURE 5. The data symbol $u_{1,1}$ and both its row and column parities are erased. Several failure patterns arise: Up to at most two quadrant parities, e.g. ⓐ may be erased. A second data symbol may be erased, it could be in a different quadrant, e.g. ⓑ, or in the same quadrant, e.g. ⓒ or ⓓ. On the right: A subgrid of four data symbols and one of the row or column parities fail.

(i) If there are two data erasures (shown in Figure 5 on the left), subject to the cap of five erasures in total, at most one quadrant parity is erased on top of the row and column parities of the first data symbol. As such, multiple pathways for recovery are possible. If the second erased data symbol (pattern ⓑ) is in a different quadrant, the first data symbol can be recovered independently (of the availability of the row/column parities of the second data symbol), using a parity from a quadrant different from that of the second data symbol. Even if the second data symbol is in the same quadrant instead (pattern ⓒ), it may be recovered using its row or column parity. However, if it further shares the row or column with the first data, and furthermore the other column/row parity is also missing (pattern ⓓ), then they will account for the five failures. This would then mean all the three quadrant parities for the affected quadrant are available. Say both erased data symbols are missing in quadrant 1, and let us call $q_{i,j}$ the parity obtained from quadrants i and j. Then $q_{1,2} = (g_{3,1}u_{1,1} + g_{3,2}u_{1,2} + g_{3,5}u_{2,1} + g_{3,6}u_{2,2}) + (g_{3,3}u_{1,3} + g_{3,4}u_{1,4} + g_{3,7}u_{2,3} + g_{3,8}u_{2,4})$ and similarly $q_{1,3} = (g_{4,1}u_{1,1} + g_{4,2}u_{1,2} + g_{4,5}u_{2,1} + g_{4,6}u_{2,2}) + (g_{4,9}u_{3,1} + g_{4,10}u_{3,2} + g_{4,13}u_{4,1} + g_{4,14}u_{4,2})$. The recovery matrix then becomes

$$\begin{bmatrix} g_{3,1} & g_{3,5} \\ g_{4,1} & g_{4,5} \end{bmatrix} \begin{bmatrix} u_{1,1} \\ u_{2,1} \end{bmatrix} = \begin{bmatrix} \tilde{q}_{1,2} \\ \tilde{q}_{1,3} \end{bmatrix}$$

where as earlier $\tilde{q}_{i,j}$ denote the available part of the parity equation obtained from $q_{i,j}$. Since all 3 quadrant parities are available, there were 3 choices of 2 parities, we chose one such a choice. This leads to further constraints, namely, that this condition should also hold for any combination of two quadrant parities (we chose the parities from the pairs (1,2) and (1,3)) to illustrate this example. In Appendix, as a proof of existence of codes satisfying all the constraints we have identified, we provide a concrete instantiation (in fact, we demonstrate two instances over different field-sizes, but for the same (30, 16) code parameter), accompanied with an evaluation of the code instance in terms of its resilience against failures, quantifying the cost of degraded reads and rebuilds to recover from failures, thus demonstrating the suitability of such codes for realizing a persistent and highly available distributed data store.

(ii) If three data symbols are missing, given a cap of five erasures, only two parities would be erased. It implies that some of the data symbols would retain at least one of their row or column parities, with which they can be recovered first.

(iii) Finally, a worst case scenario is when a subgrid of four symbols from the same quadrant as well as a parity from one of the rows or columns is missing (shown in Figure 5 on the right). In this case, there are still three quadrant parities, and three of the remaining row/column parities carry information about the four missing data symbols, which can be used to guarantee recovery using the following recovery matrix:

$$\begin{bmatrix} g_{1,1} & g_{1,2} & 0 & 0 \\ 0 & 0 & g_{1,5} & g_{1,6} \\ g_{2,1} & 0 & g_{2,5} & 0 \\ g_{3,1} & g_{3,2} & g_{3,5} & g_{3,6} \end{bmatrix} \begin{bmatrix} u_{1,1} \\ u_{1,2} \\ u_{2,1} \\ u_{2,2} \end{bmatrix} = \begin{bmatrix} \tilde{p}_{1,1} \\ \tilde{p}_{1,2} \\ \tilde{p}_{2,1} \\ \tilde{q}_{1,2} \end{bmatrix} \quad (4)$$

subject to choosing the coefficients so that the matrix is invertible. We obtain a set of similar constraints for the different combinations of failures satisfying the same pattern.

This analysis helps establish our lower bound on the general fault tolerance of five erasures for the proposed code. Note that the arguments used to establish this lower bound on 5-faults tolerance generalizes to arbitrary choices of r_1, r_2 and r_3. Larger values for these parameters may (or not) yield even better fault-tolerance. We defer the establishment of a tight lower bound for arbitrary code parameters to be studied in the future.

4) PRACTICAL IMPLICATIONS

An analysis of four years of failure operation tickets from Baidu's data centers [34] indicated that more than 80% of the failures are hard disk failures. A study from Google [35] likewise observed thousands of hard disk failures annually, accounting for between $1-5\%$ of the disks. Servers on an average crashed at least twice annually, accounting for $2-4\%$ failure rate. This amounted to at least one server failure daily. These statistics emphasize the need for fault tolerant storage solutions. They also provide indicators for the range of environments for which one needs to design. For instance, using a back of the envelope estimate based on the 5% annual disk failure rate mentioned above, one can estimate an indicative $5/365 \approx 0.014\%$ daily failure rate.

We study the behavior of some of the code choices from Table 1 assuming simultaneous failures are independent, for a rate of failures ranging between 0.1% and 2%, which is essentially an order up to two orders of magnitude higher than the estimated 'average'. We consider these higher rates of simultaneous failures, in part, (i) to compensate for the fact that real world failures are not always independent and in fact correlated failures are typical, while (ii) the independence assumption allows for simpler mathematical analysis of the system. Furthermore, note that we are studying the durability of the data, which will be immune to transient

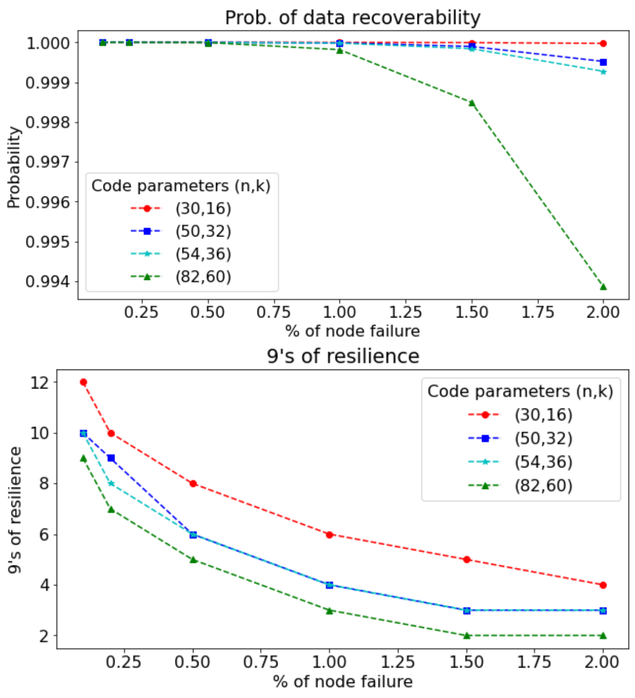


FIGURE 6. Data durability estimate.

failures, and is affected by permanent failures. As such, while catastrophic failures or disasters may lead to more than 2% simultaneous failures, it is a very pessimistic setup for normal operations.

Given the assumptions regarding failures stated above, if the probability of failure of a storage node is ϕ, then in a cluster of n nodes, the probability that less than six failures occur simultaneously (since, as per Proposition 2, the lower bound of arbitrary failures that the proposed coded system tolerates is five) is $\sum_i^5 \binom{n}{i}\phi^i(1-\phi)^{n-i}$. In Figure 6 we show the data durability in terms of probability of data recoverability, and the 9's of durability for certain code parameters. The latter metric, namely 9's of durability, is a popular and practical way to express a system's resilience, e.g., five nines of durability would mean a minimum of 99.999% durability.

We notice that even for 0.5% chance of failure of nodes individually, which is 36 times more than the above estimated daily failure rate of $\approx$ 0.014%, all the codes shown in the Figure achieve five 9s of resilience. Smaller values of n naturally give higher resilience, since 5 failures need to occur among a smaller set of nodes - thus the (30, 16) code has eight 9's of durability at this operational point, and has five 9's of durability even when chance of individual node failures is 1.5%, which is $\approx$100 times the estimated daily failure rate. Aiming beyond four or five 9's of reliability for the storage elements may not be meaningful, since the networking elements of a data center just about achieves four 9's [36] or reliability and becomes the primary bottleneck at this point. These results shows that multiple instances of our proposed code satisfy practical fault tolerance needs and have acceptable storage overhead and cost of repair of failures (the latter two are aspects we already discussed in the context of Table 1).

IV. QUORUM SYSTEMS

Definition 1: In the context of replicated systems a *quorum system* $\mathcal{Q}$ has been defined as a set of subsets of replica nodes called *quorums*, such that every two quorums intersect, i.e., $Q \cap Q' \neq \varnothing$ for all $Q, Q' \in \mathcal{Q}$.

Whenever a quorum needs to be built to read or write data, failed nodes cannot participate until not only they join the system anew, but are repopulated with up-to-date content. A "vote" (or a "lock") is attributed to every node in the system. Any attempt to either read or write data needs to gather enough votes in order to perform the operation. Given the intersecting property of quorums, mutual exclusion of a write operation with any other write or read operations is achieved. This mutual exclusion guarantees that a write operation reads the latest value of the data, and thus can support read-modify-write operations without any underlying atomic primitives. In general, it yields consistency by preventing different write operations simultaneously, or for read operations when a write is being carried out. Read locks are not mutually exclusive, but are exclusive with respect to write locks.

Remark 2: We note that the same idea of quorums could also be used without locks, and by storing multiple logical time-stamped versions of the data. In such a scenario, the application layer would have to determine a chronological ordering based on versions. Such a lock-free approach would be non-blocking, and thus enjoy higher level of concurrency, but with the compromise that read-modify-write operations would not be supported. That would be suitable for applications where write operations are agnostic of previously written values. For the rest of the paper, we focus on the approach that supports causally related read-modify-write operations.

This locking principle ensures the safety property for consistency, so that two overlapping writes are not possible, so a new write operation can read the latest previously written value, modify it as needed and then complete the write process before any other write operations try to read and modify the data. Accordingly, the system achieves sequential consistency, namely, there is a globally unique ordering for the read/write operation sequences, where the ordering is determined as per the sequence in which the locks for the respective operations are acquired.

There is also the issue of liveliness, which is outside the scope of this work. If partial locks are acquired by different processes, it may lead to deadlocking. Livelocks may also hinder progress. These are well studied problems and techniques to detect and deal with these exist (see for example [37], [38]). Our work assumes the existence of an extrinsic distributed locking service such as [39] for orchestrating the acquisition and release of locks.

Quorum systems are broadly categorized as symmetric and asymmetric. For the latter, a write quorum's membership is different from a read quorum's, while the former does not distinguish the two. We consider two variations of our local reconstruction codes to propose symmetric quorums.

The following discussions on consistency will rely on three practical assumptions: (i) Any update is atomically executed across all members of the quorum. (ii) There is a per object gossip group $\mathcal{G}_{i,j}$ comprising the node storing the data block $u_{i,j}$, and all the parities carrying information about this block, irrespective of whether these parities are part of the quorum or not. When the data object is updated, the (bit wise) differential of the new version and the previous version is propagated in $\mathcal{G}_{i,j}$. Updates are incorporated at the parities that were not in the quorum but receive them by gossip only in proper order using logical clocks [40]. Parities in the quorum necessarily execute them in order. Such differentials are garbage collected at all the nodes only after ascertaining that all the other members of the group have incorporated the update. We note that the size of each of these gossip groups is $1+r_1+r_2+3r_3$ (which is 6 when $r_1 = r_2 = r_3 = 1$) and thus the update propagation should be very fast, posing negligible and temporary overheads of storing the differentials. (iii) The number of total failures in the system is capped at the lower bound of fault tolerance of the coded system deployed.

Definition 2: Quorum system $\mathcal{Q}_{rc}$ for codes with *only row and column parities, i.e.* $r_3 = 0$*:* The quorum $Q_{i,j}$ to access the data symbol $u_{i,j}$ comprises the nodes in position $\{(i, j), (i, 2d_2 + l), (2d_1 + k, j)\}$ for $l = \{1, \dots, r_1\}$ and $k = \{1, \dots, r_2\}$, i.e., the data symbol and the parities sharing its row and column in the logical grid layout.

The proposed quorum in Definition 2 does not satisfy the condition from Definition 1, since $Q_{i,j} \cap Q_{k,l} = \varnothing$ if both $i \neq k$ and $j \neq l$ hold. Still, it is adequate to satisfy necessary mutual exclusion for guaranteeing consistency subject.

Given that we consider the case of $r_3 = 0$, i.e., there are no quadrant parities, we need mutual exclusions of two write operations, or a write operation with other read operations only in the rows and columns of the concerned data objects. If this mutual exclusion is not guaranteed, simultaneous write operations at different data objects in the same row or column may render the corresponding parities in inconsistent state, thus rendering them useless for data recovery.

Example 2: As a very simplified example, consider two data objects with the values 3 and 7 and the parity carries the sum of these two values, thus initially it is 10. If the two write operations are increasing the values of the two data objects to 5 and 12 respectively, but these updates happen simultaneously and thus one of the changes in the parity is overwritten, it may end up with a value of 12, reflecting only the change of value from 3 to 5. If this data object is later lost, and one tried to recover it using the parity 12 and the other surviving data object, which also has a value 12, then one would wrongly 'recover' the value 0 which is neither the present nor even any of the past values of the data object.

When two data objects share no row or column in the grid layout, the parities they influence comprise disjoint sets. Mutual exclusion among them is thus immaterial to prevent inconsistency. Hence the proposed quorum achieves sequential consistency at the granularity of individual data objects.

The quorum composition can be extended for the case where quadrant parities also exist (i.e., $r_3 > 0$):

Definition 3: Quorum system $\mathcal{Q}_{rcq}$ for codes with *row, column as well as quadrant parities, i.e.* $r_3 > 0$*:* The quorum $Q_{i,j}$ to access the data symbol $u_{i,j}$ comprises the nodes from Definition 2 as well as the $3r_3$ quadrant parities that are determined according to the quadrant to which $u_{i,j}$ belongs.

The quorum system $\mathcal{Q}_{rcq}$ satisfies the condition from Definition 1. Consider $Q_{i,j}$ and $Q_{k,l}$. If positions i, j and k, l belong to the same quadrant, then their quorums will intersect at all the $3r_3$ quadrant parities of each of their respective quorums. If they belong to two different quadrants, say Q_x and Q_y where $x \neq y$, they will intersect at the r_3 quadrant parities that are computed using the data symbols from the quadrants x and y. This ensures mutual exclusion and thus sequential consistency at the granularity of the set of data objects being coded together, and thus also for the individual data objects. The immediate downside in this case, unlike the former scenario without quadrant parities is that, because of the mutual exclusion among all the data objects, parallelization of processing is by design not possible.

This observation prompts us to propose an alternate mechanism which considers a laxer form of consistency (see Definition 4) where the quadrant parities are used for fault-tolerance but are not involved in the quorum mechanism, that is, deploy a code with $r_3 > 0$ but use the quorums as per Definition 2. In this setup, the quadrant parities are to be updated in a best-effort manner [20]–[23] through a background propagation of the updates to the quadrant parities.

The background best-effort update process is possible in linear codes by propagating the update deltas of the original data object values, since only the product of this update delta and the corresponding code coefficient need to be locally multiplied and added to the previous value of the parity, and these operations commute across updates from different objects. While the operations would also commute for multiple updates for the same data object, we apply multiple updates from a single object only in sequence determined by a logical clock. The implications of this choice are discussed later.

Furthermore, under most failure scenarios, only the row and column but not the quadrant parities are used for recovering the lost data objects, in which case, they continue to exhibit sequential consistency at the object level granularity. In some situations where the data can only be recoverable using quadrant parities, we guarantee a weaker kind of consistency, which we call *semantic consistency*, inspired by the concept of semantic reconciliation studied in the Dynamo system [12]. Consider an object that undergoes updates over time, following the sequence of versions $V = (v^0, v^1 \ldots v^t)$ and then the disk storing it fails, so that it has to be recovered using the coded redundancy. For guaranteeing sequential consistency, one should recover the last updated version v^t. However, the idea of semantic reconciliation is to let the application layer to deal with inconsistencies based on application logic, instead of enforcing it at storage layer. The weaker form of guarantees that the data object takes any one of the historical values from V above (and avoids a scenario as the one discussed in Example 2), which we term as semantic consistency.

Definition 4: Semantic consistency is said to be achieved if the value of a data symbol being recovered from a failure corresponds to any of the past legitimate values $V = (v^0, v^1 \ldots v^t)$ it has had before the failure.

Semantic consistency suffices in many scenarios where applications can carry out semantic reconciliation [12]. Our overall approach thus allows the deployment of the quorum system in different modes - using the quorums described in Definition 3 to guarantee sequential consistency which will however by design (as per traditional quorum systems) not support concurrency of reads and writes, or deploy it using quorums described in Definition 2 to achieve semantic consistency which allows such concurrent operations.

The semantic consistency guarantee is achieved as follows. There are broadly three possible scenarios. The latest update(s) of the data node which needs to be recovered has been received by (i) some or (ii) all of the concerned quadrant parities. In these cases, they can ensure that all of them incorporate these updates. The other situation is when (iii) all the live parities miss certain updates. In this case they ignore any updates if intermediate differentials are unavailable. So they all reflect the same legitimate past version. Likewise, all the parities incorporate any other updates corresponding to all the other live data objects in the system, obtained by participating in the respective update propagation gossip groups.

The recovery operation is carried out subsequent to these steps. As a consequence, if there are certain missed updates, then the recovered data object corresponds to an older version, while if there were no missed updates, the latest version happens to be created; in all cases the system strictly generates a version that was the result of legitimate past operations.

A. SIMULATIONS

We carried out a discrete rounds based simulation to study the effect of the granularity of mutual exclusions achieved by $\mathcal{Q}_{rc}$ (only row/column parities are used) versus $\mathcal{Q}_{rcq}$ (quadrant parities are also used) quorums, to determine their effect on concurrency.

1) WORKLOAD

We study how the proposed mechanism performs over one group of coded data, using a $(54, 36)$ code, i.e., there were 36 original data blocks, and further 18 parity blocks, thus incurring a storage overhead of $1.5\times$. A workload of read/write requests was randomly generated, with the

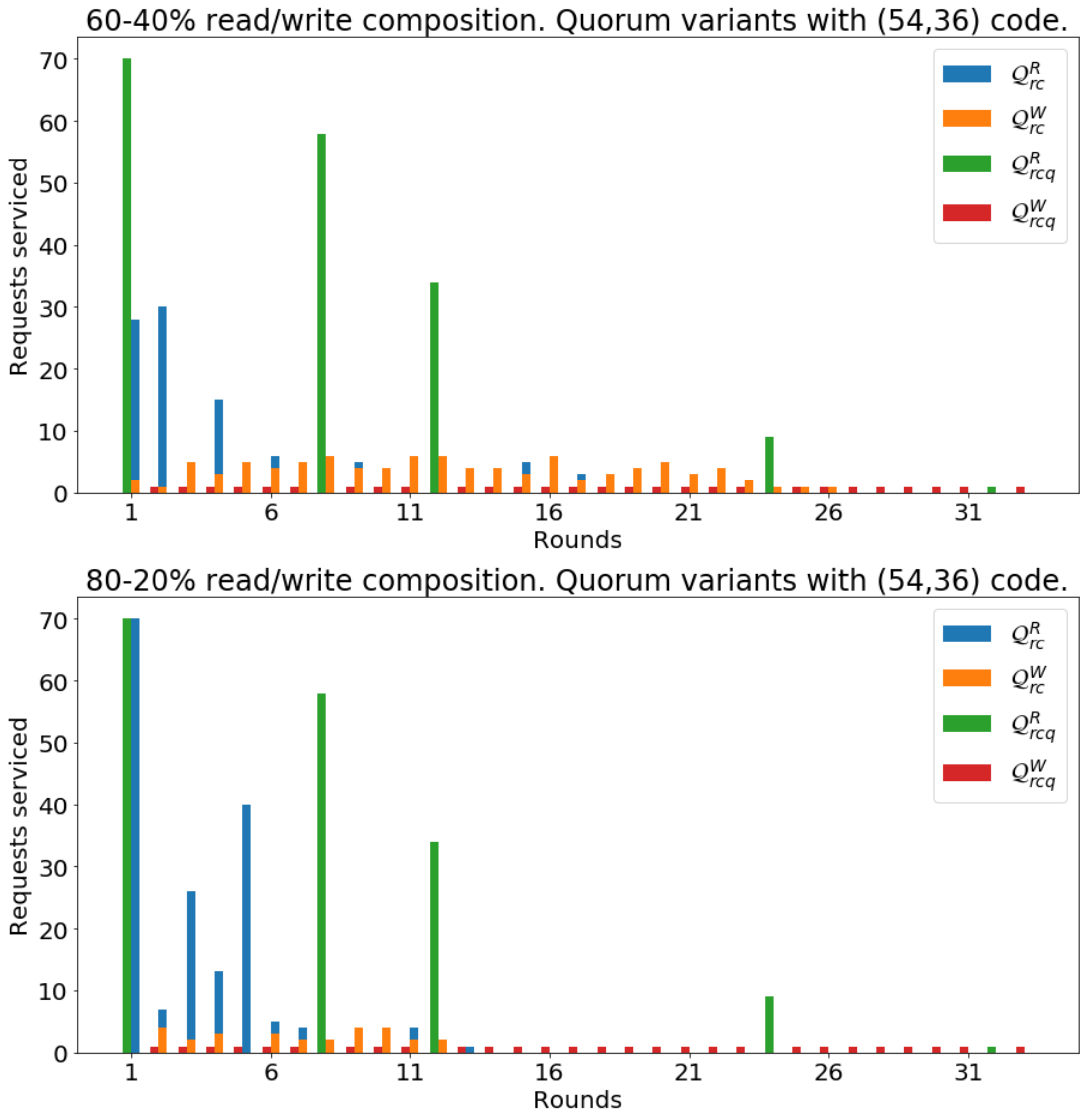


FIGURE 7. Performance of $\mathcal{Q}_{rc}$ and $\mathcal{Q}_{rcq}$ quorums for different read/write mixes. Superscripts R/W indicate read/write.

probability of read requests being 0.6 and 0.8 in two sets of separate experiments, for which we show the results in Figure 7. Recall that the proposed mechanism is designed for warm (and not hot) data. As such, the absolute access rate on the data is supposed to be low, and the relative frequency of write operations is supposed to be particularly low. We nevertheless choose a very high proportion of write accesses, to study the proposed mechanisms under what may be termed as a 'stressed' condition. We consider a snapshot of the system, which involves a 200 requests overall, with the request types mix as described above. Additionally, we assume that the requests are furthermore for data objects chosen uniformly at random within the group of coded data blocks. The choice of random accesses for the workload is aligned with the systems architecture and assumptions presented earlier in Section I-A. In particular, recall that the logical data objects comprise independent BLOBs, and the group of such data objects to be coded together are placed in physically distinct nodes, and the derived parities are placed in further distinct storage nodes.

2) THROUGHPUT THROTTLED SIMULATIONS

In order to achieve a read-modify-write operational semantics, write operations on the same data object are necessarily mutually exclusive, yielding a blocking algorithm that needs to prevent other reads or writes simultaneously on the same data object. Conversely, since a read lock in $\mathcal{Q}_{rcq}$ would make any other simultaneous write operations infeasible, if we consider only the quorum based locking constraint, all other read requests can be serviced immediately. However, in practice, storage nodes may not have the capacity to process all requests simultaneously. As such, in the simulations, we throttle the number of read requests that a given storage node processes in a single time round in order to capture such extrinsic limitations. Thus, overall, both because of this artificially imposed cap to throttle the processing rate, as well as because we consider a high relative frequency (20%-40%) of write operations, the results presented here are under extremely pessimistic set-up, and thus provides a very conservative estimate of performance of the envisioned system applying QLOC for concurrency control in order to achieve consistency. The reported results are with a cap of 2 operations per storage node. Since we used a (54, 36) code for these experiments, this constraint meant that not more than 72 read requests across the 36 data objects could be processed in a given round. Even without this cap, the characteristic behavior of $\mathcal{Q}_{rcq}$ remains the same - namely, whenever there is a write operation, no other read or write operations are possible in the given round.

3) FINDINGS FROM THE SIMULATIONS

For the simulated setups, with $\mathcal{Q}_{rcq}$ we needed 33 rounds to process all the requests in the system. In contrast, with $\mathcal{Q}_{rc}$, these jobs could be finished in 26 and 13 rounds respectively, for the ~60/40% and ~80/20% read/write mixes. This is so, because with $\mathcal{Q}_{rc}$ simultaneous read and even multiple write operations can be carried out concurrently involving data objects which do not share either the row or the column in the logical grid. On the downside, when a write lock is acquired for a given data object, it prevents reads on other data objects sharing the same row or column. Thus the peak number of read requests processed per round is typically lower than with $\mathcal{Q}_{rcq}$. If the workload contains relatively more write requests, then $\mathcal{Q}_{rc}$ may require more time than $\mathcal{Q}_{rcq}$ to service some of the read requests, even though $\mathcal{Q}_{rc}$ services the complete set of requests faster than $\mathcal{Q}_{rcq}$. We have tried other parameter values in terms of code configurations, work-load mix, cap on the number of simultaneous reads on any given data object. We consistently observed the same relative behaviors.

The earlier theoretical analysis had helped us establish the extent of fault-tolerance one can achieve, depending on whether the semi-global parities are deployed ($r_3 > 0$) or not ($r_3 = 0$), namely, guaranteed data recovery in the presence of arbitrary 5 or 2 failures respectively. We had also determined by logical analysis the characteristics of consistency depending on the choice of nodes involved in creating the quorums. Namely, sequential consistency at the level of objects or group of coded objects can be achieved with $\mathcal{Q}_{rc}$ when $r_3 = 0$ versus $\mathcal{Q}_{rcq}$ ($r_3 > 0$ by default) respectively. In contrast, if the semi-global parities are leveraged solely for fault-tolerance and not for the quorum induced locking mechanism, then a weaker semantic consistency at object level can be achieved. The simulations complement these analysis, to establish the relative level of concurrency and thus throughput that are yielded by these different choices. In Table 2 this spectrum is summarized. In particular, we notice that with $\mathcal{Q}_{rc}$, we can support some extent of concurrent write operations within a coded group while reducing marginally the read throughput, while $\mathcal{Q}_{rcq}$ does not allow any concurrent writes across the

TABLE 2. QLOC deployment modes.

Code	Quorum	Fault tolerance	Concurrency (read/write)	Consistency guarantee
$r_3 = 0$	$\mathcal{Q}_{rc}$	any two failures	high /moderate	sequential (object level)
$r_3 > 0$	$\mathcal{Q}_{rcq}$	any five failures	very high /none	sequential (objects group)
$r_3 > 0$	$\mathcal{Q}_{rc}$	any five failures	high /moderate	semantic (object level)

whole group, but consequently higher throughput for read operations become possible. More crucially, $\mathcal{Q}_{rcq}$ allows for the strongest form of consistency guarantee, since it extends the guarantee at the granularity of the whole object group. As such, it would be effective for providing strong sequential consistency guarantee even for applications which transcend our optional assumption that the BLOBs within data objects and the data objects being coded together are independent.

B. QUORUM LOAD

A classical metric to evaluate a quorum system is to study its load - a measure of how often quorums are accessed. For every quorum $Q \in \mathcal{Q}$, an access probability $P_S(Q)$ is defined for an access strategy S. By definition, $\sum_{Q\in\mathcal{Q}} P_S(Q) = 1$. Then the load $L_S(m)$ of a node m using the access strategy S is

$$L_S(m) = \sum_{\substack{Q\in\mathcal{Q}\\ m\in Q}} P_S(Q) \tag{5}$$

so that the load induced by S on the system is the load imposed by the busiest node $L_S(\mathcal{Q}) = \max_m L_S(m)$. The load of a quorum system $\mathcal{Q}$ is the minimal load, across all possible access strategies that can be used.

Quorums in both $\mathcal{Q}_{rc}$ and $\mathcal{Q}_{rcq}$ are invoked only because of access requests for data nodes. Assuming data nodes are equally likely to be accessed, we thus have $P_S(Q_{i,j}) = \frac{1}{4d_1d_2}$ and $P_S(Q)$ is uniform. From (5), the load of node m for S uniform is

$$L_S(m) = \sum_{\substack{Q\in\mathcal{Q}\\ m\in Q}} \frac{1}{4d_1d_2} = \frac{|Q \in \mathcal{Q},\ m \in Q|}{4d_1d_2}$$

so $L_S(m) \in \left\{\frac{1}{4d_1d_2}, \frac{1}{2d_2}, \frac{1}{2d_1}, \frac{1}{2}\right\}$ since node m belongs to a single quorum if it is a data symbol, to $2d_1$ quorums if it is a column parity, to $2d_2$ quorums if it is a row parity, and to $2d_1d_2$ quorums if it is a quadrant parity. In summary:

Proposition 3: For a quorum access probability $P_S(Q)$ uniform over $Q \in \mathcal{Q}$, the quorum system $\mathcal{Q}$ has load

$$L_S(\mathcal{Q}) = \begin{cases} \dfrac{1}{2\min\{d_1, d_2\}} & \text{for } \mathcal{Q} = \mathcal{Q}_{rc} \\ \dfrac{1}{2} & \text{for } \mathcal{Q} = \mathcal{Q}_{rcq}. \end{cases}$$

We immediately see from the analysis that the quorum system $\mathcal{Q}_{rcq}$ involving quadrant parities has a higher load than $\mathcal{Q}_{rc}$. This has two related implications. The higher load reflects that $\mathcal{Q}_{rcq}$ consumes more system resources. A direct consequence is that fewer requests can be handled for a given capacity of resources, impacting the request processing throughput in the system. Since the load indicates how busy nodes are, in our setting it also acts as a proxy to theoretically estimate the relative extent of concurrency the two modes of quorum deployments can handle.

V. CONCLUSION

In this paper we extended the work on local reconstruction codes and proposed a novel parameterized family of codes using local and semi-global parities. We established theoretical results on the lower bound of fault-tolerance for code instances with practical parameters, and established their efficacy in real-world set-ups in terms of storage overhead and resilience. We demonstrate the existence of such codes, by providing a concrete instantiation, and benchmarking the fault-tolerance characteristics of the instance to further demonstrate the suitability of such codes for practical deployment. We laid conceptual foundations for flexible quorum systems over coded data, exploiting the existence of local and semi-global parities to navigate trade-offs in terms of fault-tolerance, concurrency of read and (read-modify-)write operations, consistency guarantees and quorum system load. In the process, we proposed a weaker form of consistency which we named semantic consistency. The trade-offs arising in the different deployment modes of our flexible framework are summarized in Table 2.

Providing mechanisms to guarantee consistency in erasure coded data will enable the deployment of a richer variety of applications using such storage cost efficient approach, triggering a second stage of wider adoption of erasure coded storage systems, similar in impact as the earlier research on efficient computation and repair of erasure coded data.

This is a first work of its kind - establishing the theoretical foundations and demonstrating the feasibility to achieve sequential consistency for read-modify-write operational semantics over locally repairable erasure-coded distributed data. It ushers into multiple future research directions. For the specific locally repairable code family, we have showcased their feasibility with code instances derived using numerical methods. Other ways to construct codes satisfying the constraints is of immediate interest. At a fundamental level and in terms of longer term implications, this work is embryonic in nature — properties of locally repairable codes with very different code structures may also potentially be exploited to build quorum systems that provide same functionalities of concurrency control, consistency and operations semantics which we achieve. This thus yields a new aspect that both coding theorist and distributed algorithms researchers alike may pursue, to establish different, and possibly better performing alternative solutions.

APPENDIX
EXAMPLE CODE INSTANCE

The objective of this paper was to identify mechanisms to achieve a strong form of consistency, namely sequential consistency, while supporting a strict form of operational semantics, namely read-modify-write operations, over distributed data stored redundantly using locally repairable codes. The QLOC framework, and the high-level description of the code (accompanied with the set of constraints that it needs to satisfy) suffices to that end. Nonetheless, we also need to demonstrate that such codes can actually be realized. To that end, in this appendix, we provide an explicit construction of such a code instance. We use the specific instance to also benchmark its properties and demonstrate its suitability in meeting practical requirements in terms of storage overhead, fault-tolerance, and cost of data recovery or degraded reads in presence of faults.

Specifically, we choose a (30, 16) code that was analyzed in Example 1. We show next how to instantiate such a code, starting from a (24, 16) base code. We find the code instance using a numerical approach analogous to [1]. This example uses $\mathbb{F}_{2^6}$ for the base field – specifically, $\mathbb{F}_{2^6} = \{0, \alpha^i, \ i = 0, \ldots, 63\}$, for α satisfying $\alpha^6 + \alpha + 1 = 0$ over $\mathbb{F}_2$. The (24, 16) code can be described by the generator polynomial $\prod_{j=1}^{r}(x - \alpha^j)$ for $r = 8$ ($r = 8$ gives the number of parities), that is $g(x) = x^8 + \alpha^{43}x^7 + \alpha^{59}x^6 + \alpha^{31}x^5 + \alpha^{10}x^4 + \alpha^{40}x^3 + \alpha^{14}x^2 + \alpha^7 x + \alpha^{36}$.

Since we will use the systematic form of this code, we only store the coefficients used to compute the parities, giving an $r \times k = 8 \times 16$ matrix. As for the entries of this matrix, we calculate the remainder of $x^{(n-j)} \cdot x^r$ modulo $g(x)$ for $j = 1, 2, \ldots, n$, and write the coefficients of these remainders as entries, in descending order of power. For example, we compute that the remainder of x^{15} modulo $g(x)$ is $\alpha^{23}x^7 + \alpha^{31}x^6 + \alpha^{43}x^5 + \alpha^{48}x^4 + \alpha^{50}x^3 + \alpha^{43}x^2 + \alpha^{53}x + \alpha^{13}$, the remainder of x^{14} modulo $g(x)$ is $\alpha^{40}x^7 + \alpha^{52}x^6 + \alpha^{30}x^5 + \alpha^{21}x^4 + \alpha^{60}x^3 + \alpha^{33}x^2 + \alpha^5 x + \alpha^{48}$.

Repeating the above computations for each power of x yields the following matrix:

$$G^T = \begin{bmatrix}
\alpha^{23} & \alpha^{31} & \alpha^{43} & \alpha^{48} & \alpha^{50} & \alpha^{43} & \alpha^{53} & \alpha^{13} \\
\alpha^{40} & \alpha^{52} & \alpha^{30} & \alpha^{21} & \alpha^{60} & \alpha^{33} & \alpha^{5} & \alpha^{48} \\
\alpha^{12} & 1 & \alpha^{45} & \alpha^{2} & \alpha^{27} & \alpha^{37} & \alpha^{52} & \alpha^{57} \\
\alpha^{21} & \alpha^{26} & \alpha^{47} & \alpha^{8} & \alpha^{62} & \alpha^{58} & \alpha^{47} & \alpha^{32} \\
\alpha^{59} & \alpha^{48} & \alpha^{23} & \alpha^{23} & \alpha^{18} & \alpha^{43} & \alpha^{18} & \alpha^{40} \\
\alpha^{4} & \alpha^{49} & \alpha^{8} & \alpha^{25} & \alpha^{59} & \alpha^{25} & \alpha^{29} & \alpha^{37} \\
\alpha & \alpha^{59} & \alpha^{11} & \alpha^{12} & 1 & \alpha^{5} & \alpha^{13} & \alpha^{50} \\
\alpha^{14} & \alpha^{20} & \alpha^{48} & \alpha^{42} & \alpha^{14} & \alpha^{36} & \alpha^{20} & \alpha^{61} \\
\alpha^{25} & \alpha^{50} & \alpha^{26} & \alpha^{33} & \alpha^{61} & \alpha^{4} & \alpha^{5} & \alpha^{22} \\
\alpha^{49} & \alpha^{49} & \alpha^{44} & \alpha^{62} & \alpha^{40} & \alpha^{39} & \alpha^{24} & \alpha^{58} \\
\alpha^{22} & \alpha^{19} & \alpha^{52} & \alpha^{26} & \alpha^{15} & \alpha^{27} & \alpha^{5} & \alpha^{23} \\
\alpha^{50} & \alpha^{7} & \alpha^{37} & \alpha^{49} & \alpha^{57} & \alpha^{17} & \alpha^{8} & \alpha^{19} \\
\alpha^{46} & \alpha^{30} & \alpha^{20} & \alpha^{29} & \alpha^{12} & \alpha^{54} & \alpha^{56} & \alpha^{17} \\
\alpha^{44} & \alpha^{11} & \alpha^{28} & \alpha^{60} & \alpha^{40} & \alpha^{57} & \alpha^{15} & \alpha^{50} \\
\alpha^{14} & \alpha^{16} & \alpha^{16} & \alpha^{12} & \alpha^{15} & \alpha^{29} & \alpha^{25} & \alpha^{16} \\
\alpha^{43} & \alpha^{59} & \alpha^{31} & \alpha^{10} & \alpha^{40} & \alpha^{14} & \alpha^{7} & \alpha^{36}
\end{bmatrix}$$

The same procedure, changing the base field and using instead the element α such that $\alpha^8 + \alpha^4 + \alpha^3 + \alpha^2 + 1 = 0$

yields the following G^T matrix:

$$\begin{bmatrix}
\alpha^{60} & \alpha^{226} & \alpha^{198} & \alpha^{10} & \alpha^{65} & \alpha^{188} & \alpha^{84} & \alpha^{44} \\
\alpha^{8} & \alpha^{240} & \alpha^{206} & \alpha^{208} & \alpha^{106} & \alpha^{247} & \alpha^{145} & \alpha^{96} \\
\alpha^{60} & \alpha^{85} & \alpha^{117} & \alpha^{113} & \alpha^{201} & \alpha^{185} & \alpha^{101} & \alpha^{54} \\
\alpha^{18} & \alpha^{237} & \alpha^{62} & \alpha^{124} & \alpha^{206} & \alpha^{125} & \alpha^{139} & \alpha^{110} \\
\alpha^{74} & \alpha^{191} & \alpha^{210} & \alpha^{65} & \alpha^{213} & \alpha^{126} & \alpha^{75} & \alpha^{144} \\
\alpha^{108} & \alpha^{60} & \alpha^{232} & \alpha^{26} & \alpha^{222} & \alpha^{201} & \alpha^{144} & \alpha^{148} \\
\alpha^{112} & \alpha^{232} & \alpha^{239} & \alpha^{186} & \alpha^{66} & \alpha^{93} & \alpha^{102} & \alpha^{100} \\
\alpha^{64} & \alpha^{252} & \alpha^{172} & \alpha^{209} & \alpha^{242} & \alpha^{208} & \alpha^{10} & \alpha^{74} \\
\alpha^{38} & \alpha^{207} & \alpha^{195} & \alpha^{145} & \alpha^{13} & \alpha^{132} & \alpha^{128} & \alpha^{240} \\
\alpha^{204} & \alpha^{205} & \alpha^{174} & \alpha^{192} & \alpha^{228} & \alpha^{182} & \alpha^{76} & \alpha^{127} \\
\alpha^{91} & \alpha^{4} & \alpha^{60} & \alpha^{59} & \alpha^{163} & \alpha^{30} & \alpha^{14} & \alpha^{218} \\
\alpha^{182} & \alpha^{218} & \alpha^{186} & \alpha^{17} & \alpha^{102} & \alpha^{37} & \alpha^{189} & \alpha^{228} \\
\alpha^{192} & \alpha^{243} & \alpha^{79} & \alpha^{77} & \alpha^{249} & \alpha^{165} & \alpha^{130} & \alpha^{82} \\
\alpha^{46} & \alpha^{3} & \alpha^{109} & \alpha^{230} & \alpha^{59} & \alpha^{62} & \alpha^{8} & \alpha^{28} \\
\alpha^{247} & \alpha^{163} & \alpha^{175} & \alpha^{56} & \alpha^{8} & \alpha^{178} & \alpha^{211} & \alpha^{212} \\
\alpha^{176} & \alpha^{240} & \alpha^{211} & \alpha^{253} & \alpha^{220} & \alpha^{3} & \alpha^{203} & \alpha^{36}
\end{bmatrix}$$

Once these matrices are constructed, one can check whether these constructions satisfy the constraints or not (we have verified that they do, for these examples).

In Table 3 we report the fault tolerance of the above codes, as determined by exhaustively enumerating all failure combinations. Since the construction of our codes is numerical in nature, the actual fault tolerance depends on the choice of coefficients used to compute the parities, which in turn may be influenced by the underlying size of the Galois Field. Consequently, we also report the absolute number of combinations of configurations for a given number of failures that were unrecoverable, for given choice of code. For instance, there are $\binom{30}{9}$ = 14,307,150 configurations of 9 faults. Among these, for 53,995 or 53,468 configurations respectively for the codes instantiated with field sizes $\mathbb{F}_{2^6}$ and $\mathbb{F}_{2^8}$, not all the 9 erasures can be recovered. Effectively, in both these scenarios, for 99.62% of the fault configurations the system is fully recoverable.

TABLE 3. **Fault tolerance of the code instances.**

Design parameters			# of erasures	Recovery rate	
d_1, d_2	r_1, r_2, r_3	F		% recoverable	# unrecoverable
2, 2	1, 1, 1	$\mathbb{F}_{2^6}$	4	100.00%	0
			5	100.00%	0
			6	99.99%	16
			7	99.97%	454
			8	99.89%	6,246
			9	99.62%	53,995
2, 2	1, 1, 1	$\mathbb{F}_{2^8}$	4	100.00%	0
			5	100.00%	0
			6	99.99%	16
			7	99.97%	451
			8	99.89%	6,189
			9	99.62%	53,468

From these results, we first observe that the analytical bounds we had established in Proposition 2 hold, i.e., the system recovers for any arbitrary combination of 5 simultaneous faults. Moreover, for these particular code instances, the bound is also tight, i.e., there exists at least some (16 to be precise) 6-failures configurations, which are not recoverable, even though almost all (593,759 among $\binom{30}{6} = 593,775$ to be precise) configurations of 6-failures are recoverable. To put this in context, 6 simultaneous failures among 30 nodes amounts to 20% simultaneous failures. Usually, the system would continuously monitor and repair isolated failures before failures can accumulate.

Most theoretical works on code designs assume that repair happens after a single failure, thus looking at the case of one erasure. A few works studying codes tolerating several failures can be found in the category of codes with availability, or codes with sequential recovery [41, Section VIII]. None of those works study the implications which primarily concern our current work which focuses on consistency. Even so, even from a repairability and resilience perspective, these theoretical works do not yield practical solutions. For instance, [42] proposes a code for $n = 7m$, $k = 3m$ where m is a (positive integer) parameter, where the code can support up to 3 erasures (as opposed to our proposed code, which tolerates up to 5 arbitrary erasures), thus tolerating fewer erasures while incurring a fixed and high storage overhead (specifically, it incurs 2.33× overhead which is beyond what has been deemed desirable in practice [2]). In [43], codes supporting 5 erasures would need a parameter $r \geq 6$, yielding $n = \binom{5+r}{5}$, meaning that say for $r = 6, 462$ nodes are needed. Though these codes have nominally more reasonable storage overhead (e.g., 1.83× for the mentioned parameters) a single system of coded data dispersed over such a large number of nodes is plainly impractical. Moreover, for such a large number of nodes, the chances of more than 5 failures are considerable, creating a serious threat to data durability, adding to their impracticality.

Data centers have reasonably stable environments [3], [34]–[36], [44]. e.g., [3] reported between 2-10% annualized failure rates for disk drives, though frequent but short duration outage events (more than 90% of these were for less than 10 minutes) in individual storage system components occur. Thus, with a storage overhead of 1.875×, tolerance of 5 simultaneous failures, and its ability to be recovered from a single failure with only 4 node accesses (refer to Table1) the code and the family of codes in general satisfy well the basic requirements for practical deployments, even if one were to ignore their amenability to support protocols for consistency, these include — choice of a range of code parameters that fit the number of nodes involved in a single system of coded data, practical storage overheads, low cost for repairs, resilience from multiple erasures.

In Figure 8 we report results from an in-depth study of the code's resilience. Specifically, for the range of failures where data is fully recoverable, till the configurations with 6 failures, when, for some configurations data cannot any more be recovered, we study the number of node accesses required for data recovery (only if the data is recoverable).

This is achieved using a greedy algorithm, which builds a matrix, adding rows one by one: each row is related to some parity node that is computed from one or more failed data

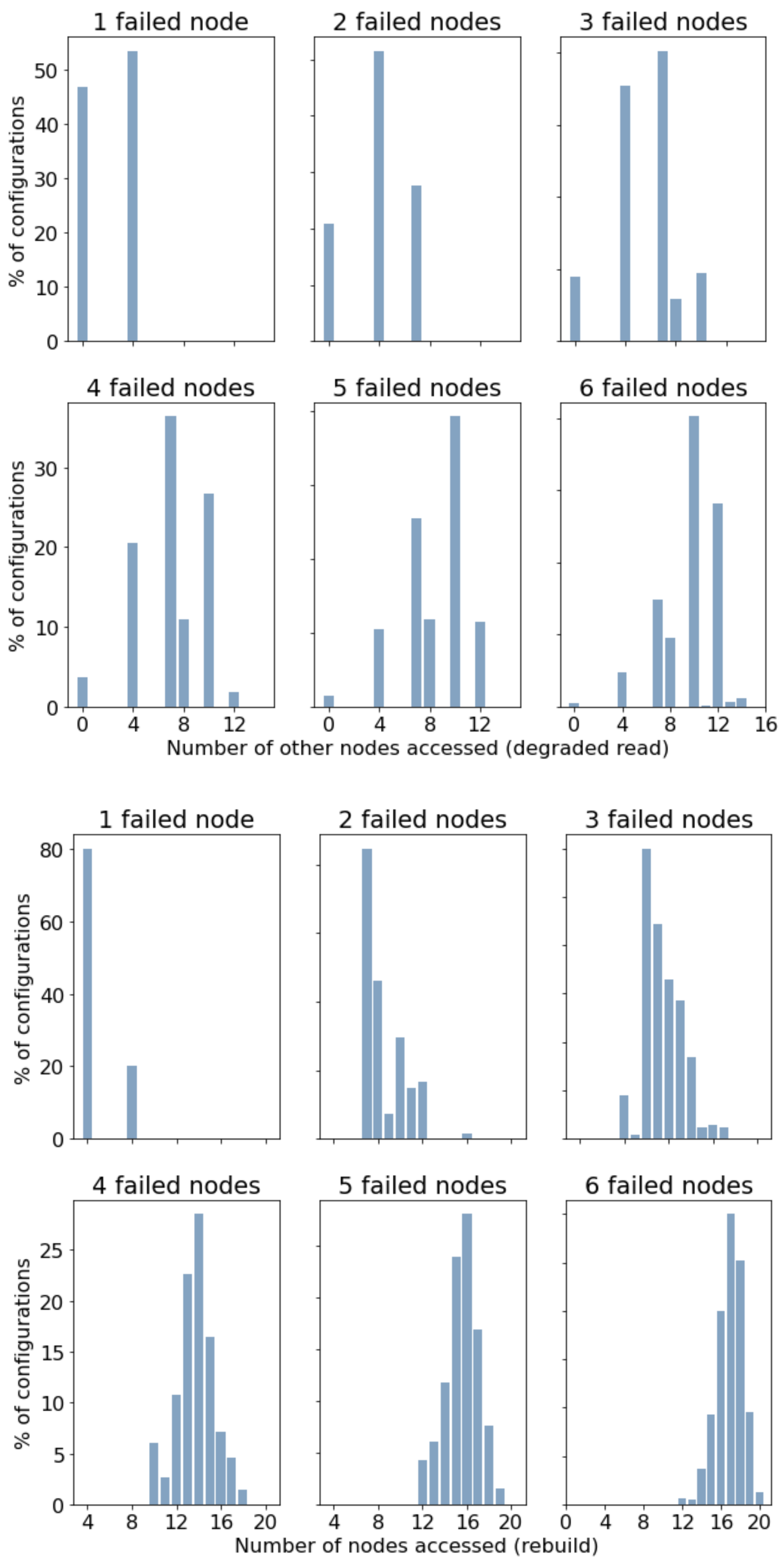


FIGURE 8. Cost of carrying out degraded reads and rebuilds under a given number of node failures (for the code over $\mathbb{F}_{2^6}$).

nodes, and rows are kept linearly independent, meaning that if a row introduces a linear combination, it is discarded.

We study two distinct scenarios: (i) *degraded reads* shown on top in Figure 8 , where only original missing data but not parity blocks are recreated, and (ii) *rebuild* shown at the bottom in Figure 8 , where all missing data blocks, original as well as parity, are recovered. In both scenarios, we consider that the failures can identically affect data and parity nodes. The former scenario is relevant to deal with relatively frequent but transitionary outages, in which case, an application may still want to carry on with its operations, by reading some or all the missing data by accessing other available nodes. The latter is the scenario where the system determines that the missing data needs to be recreated at a new live node.

Degraded Reads: Since there is a non-trivial probability that all the failures affect nodes storing parities only, there is a non-zero probability that no other nodes need to be accessed to read the data nodes. In general, the local repairability helps significantly, and for all the recoverable configurations for even up to 6 simultaneous failures, access to less than $k = 16$ other nodes was adequate to carry out the degraded reads. In Table 4 we report the expected number of other data blocks that need to be accessed for carrying out degraded reads, given a particular number of node failures. Observe that the expected numbers are significantly lower than 16 in all these scenarios. Furthermore, for up to 4 simultaneous failures, degraded reads could be carried out by accessing less than 8 other nodes for much more than a simple majority of the configurations. Moreover, depending on end-user and application needs, a degraded read may not need to access all the missing data blocks but only a subset of those, in which case, the cost of degraded read would potentially be lower, bounded by the ones determined in this study.

TABLE 4. Expected number of block accesses.

# of failures	Degraded Reads	Rebuilds
1	2.13	4.8
2	3.99	8.51
3	5.65	11.45
4	7.12	13.74
5	8.46	15.53
6	9.66	16.93

Rebuilds: Rebuilds are carried out for data as well as parity nodes. As such, even for a single failure, if a semi-global parity is affected, 8 nodes need to be accessed, while data or local parities can be rebuilt with 4 accesses. For up to two failures, most configurations can be rebuilt by accessing 8 or less nodes. These, i.e., one or two failure cases are the typical scenarios, since the system should not let failures to cumulate - doing so not only risk the loss of data, but also has performance penalties in the form of degraded operations. For larger number of failures, we notice that for certain configurations, even more than $k = 16$ nodes may need to be contacted. This exhibits a fundamental trade-off that originates from the design of locally repairable codes. In the process of optimizing for repair of data nodes, these codes cease to have the optimality of maximum distance separable (MDS) codes, leading to certain configurations where more than 16 nodes may need to be involved for rebuild. Even so, even till 5 simultaneous failures, more than a majority of the configurations require 16 or less node accesses, in contrast to MDS codes, which would require 16 accesses in all cases. In Table 4 we also report the expected number of other data blocks that need to be accessed for carrying out rebuilds. For up to 5 failures, the expected number of blocks accessed is less than 16. Furthermore, as stated above, in all these scenarios, degraded reads continue to be much more efficient.

Together, these experiments provide a complete understanding of the behavior of the particular code instance and demonstrate its practicality. They also provide qualitative understanding of this family of codes. We conclude the evaluation of the particular code instances studied here, we note that the QLOC framework provides a set of constraints that the codes need to satisfy. As such, other constructions, possibly with further better properties may or not exist. In any case, it might be possible to device constructions following a different methodology than the numerical approach presented here. The principal objective of this work was to demonstrate the viability of a quorum system that exploits the structural properties of a locally repairable code, and to that end, to design a locally repairable code that leverages on existing practical codes. Nevertheless, it was also essential to demonstrate the actual feasibility to realize a code satisfying the constraints laid out previously. The code instances presented here fulfil that purpose, demonstrating the existence of such codes by example. On top of that, we see from the experiments that the codes also meet multiple practically desirable properties - particularly in terms of the storage overhead, fault-tolerance, degraded read and rebuilds from failures.

REFERENCES

[1] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows Azure storage," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2012, pp. 15–26.

[2] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar, "F4: Facebook's warm BLOB storage system," in *Proc. Symp. Oper. Syst. Design Implement. (OSDI)*, 2014, pp. 383–398.

[3] A. Fikes, "Storage architecture and challenges," Google, Mountain View, CA, USA, Tech. Rep., 2010. [Online]. Available: https://cloud.google.com/files/storage_architecture_and_challenges.pdf

[4] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong, "Atlas: Baidu's key-value storage system for cloud data," in *Symp. Mass Storage Syst. Technol. (MSST)*, 2015, pp. 1–4.

[5] A. Wang, "Introduction to HDFS erasure coding in apache Hadoop," Cloudera, Palo Alto, CA, USA, Tech. Rep., 2015. [Online]. Available: https://blog.cloudera.com/introduction-to-hdfs-erasure-coding-in-apache-hadoop/

[6] C. Huang, M. Chen, and J. Li, "Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems," in *Proc. 6th IEEE Int. Symp. Netw. Comput. Appl. (NCA)*, 2007, pp. 79–86, doi: 10.1109/NCA.2007.37.

[7] R. C. Merkle, "A certified digital signature," in *Proc. Conf. Theory Appl. Cryptol.*, 1989, pp. 218–238.

[8] M. Vukolić, *Quorum Systems With Appllications to Atorage and Consensus*. San Rafael, CA, USA: Morgan & Claypool, 2012.

[9] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage," in *Proc. Conf. Oper. Syst. Design Implement.*, 2010, pp. 47–60.

[10] S. A. Chamazcoti, B. Safaei, and S. G. Miremadi, "Can erasure codes damage reliability in SSD-based storage systems?" *IEEE Trans. Emerg. Topics Comput.*, vol. 7, no. 3, pp. 435–446, Jul. 2019.
[11] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. 19th ACM Symp. Oper. Syst. Princ. (SOSP)*, 2003, pp. 29–43.
[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.
[13] K. S. Esmaili, L. Pamies-Juarez, and A. Datta, "CORE: Cross-object redundancy for efficient data repair in storage systems," in *Proc. IEEE Int. Conf. Big Data*, Oct. 2013, pp. 246–254.
[14] F. Oggier and A. Datta, "Self-repairing homomorphic codes for distributed storage systems," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 1215–1223.
[15] Y. Wu, H. Hou, Y. S. Han, P. P. C. Lee, and G. Han, "Generalized expanded-blaum-roth codes and their efficient encoding/decoding," in *Proc. IEEE Global Commun. Conf.*, Dec. 2020, pp. 1–6.
[16] M. Blaum and S. R. Hetzler, "Array codes with local properties," *IEEE Trans. Inf. Theory*, vol. 66, no. 6, pp. 3675–3690, Nov. 2020.
[17] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "OceanStore: An architecture for global-scale persistent storage," *ACM SIGARCH Comput. Archit. News*, vol. 28, no. 5, pp. 190–201, Dec. 2000.
[18] F. Oggier and A. Datta, "Coding techniques for repairability in networked distributed storage systems," *Found. Trends Commun. Inf. Theory*, vol. 9, no. 4, pp. 1–96, 2013.
[19] S. Liu and F. Oggier, "An overview of coding for distributed storage systems," in *Network Coding and Subspace Designs* (Signals and Communication Technology), M. Greferath, M. Pavcevic, N. Silberstein, and M. Vázquez-Castro, Eds. Cham, Switzerland: Springer, 2018, doi: 10.1007/978-3-319-70293-3_14.
[20] K. S. Esmaili, A. Chiniah, and A. Datta, "Efficient updates in cross-object erasure-coded storage systems," in *Proc. IEEE Int. Conf. Big Data*, Oct. 2013, pp. 28–32.
[21] A. Singh Rawat, S. Vishwanath, A. Bhowmick, and E. Soljanin, "Update efficient codes for distributed storage," in *Proc. IEEE Int. Symp. Inf. Theory Proc.*, Jul. 2011, pp. 1457–1461.
[22] K. Peter and A. Reinefeld, "Consistency and fault tolerance for erasure-coded distributed storage systems," in *Proc. Int. Workshop Data-Intensive Distrib. Comput.*, 2012, pp. 23–32.
[23] M. K. Aguilera, R. Janakiraman, and L. Xu, "Using erasure codes efficiently for storage in a distributed system," in *Proc. Int. Conf. Dependable Syst. Netw. (DSN)*, 2005, pp. 336–345.
[24] A. S. Tanenbaum and M. V. Steen, *Distributed Systems, Principles and Paradigms*. London, U.K.: Pearson, 2007.
[25] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok, "Extending ACID semantics to the file system," *ACM Trans. Storage*, vol. 3, no. 2, p. 4, Jun. 2007.
[26] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.
[27] S. Mu, K. Chen, Y. Wu, and W. Zheng, "When Paxos meets erasure code: Reduce network and storage cost in state machine replication," in *Proc. 23rd Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2014, pp. 61–72.
[28] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips, "Giza: Erasure coding objects across global data centers," in *Proc. Annu. Tech. Conf. (ATC)*, 2017, pp. 539–551.
[29] V. R. Cadambe, N. Lynch, M. Mèdard, and P. Musial, "A coded shared atomic memory algorithm for message passing architectures," *Distrib. Comput.*, vol. 30, no. 1, pp. 49–73, Feb. 2017.
[30] N. Nicolaou, V. Cadambe, N. Prakash, K. Konwar, M. Medard, and N. Lynch, "ARES: Adaptive, reconfigurable, erasure coded, atomic storage," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 2195–2205.
[31] K. Taranov, G. Alonso, and T. Hoefler, "Fast and strongly-consistent per-item resilience in key-value stores," in *Proc. 13th EuroSys Conf.*, Apr. 2018, pp. 1–14.
[32] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter, "Efficient Byzantine-tolerant erasure-coded storage," in *Proc. Int. Conf. Dependable Syst. Netw. (DSN)*, 2004, pp. 135–144.
[33] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, Jun. 1960.
[34] G. Wang, L. Zhang, and W. Xu, "What can we learn from four years of data center hardware failures?" in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2017, pp. 25–36.
[35] J. Dean, "Designs, lessons and advice from building large distributed systems," in *Proc. Keynote LADIS, Large Scale Distrib. Syst. Middleware (LADIS)*, 2009.
[36] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proc. ACM SIGCOMM Conf.*, 2011, pp. 350–361.
[37] K.-C. Tai, "Definitions and detection of deadlock, livelock, and starvation in concurrent programs," in *Proc. Int. Conf. Parallel Process. (ICPP)*, Aug. 1994, pp. 69–72.
[38] A. Ho, S. Smith, and S. Hand, "On deadlock, livelock, and forward progress," Comput. Lab., Univ. Cambridge, Cambridge, U.K., Tech. Rep. UCAM-CL-TR-633, 2005.
[39] R. Labs. *Distributed Locks With Redis*. Accessed: May 11, 2021. [Online]. Available: https://redis.io/topics/distlock
[40] M. Raynal, "About logical clocks for distributed systems," *ACM SIGOPS Oper. Syst. Rev.*, vol. 26, no. 1, pp. 41–48, Jan. 1992.
[41] S. Balaji, M. Krishnan, and M. E. A. Vajha, "Erasure coding for distributed storage: An overview," *Sci. China Inf. Sci.*, vol. 612, no. 10, pp. 1–45, 2018.
[42] S. Kadhe and R. Calderbank, "Rate optimal binary linear locally repairable codes with small availability," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2017, pp. 166–170.
[43] A. Wang, Z. Zhang, and M. Liu, "Achieving arbitrary locality and availability in binary codes," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2015, pp. 1866–1870.
[44] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky, "Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics," *ACM Trans. Storage*, vol. 4, no. 3, pp. 1–25, Nov. 2008.

ANWITAMAN DATTA is currently an Associate Professor with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. He serves as a Senior Scientific Officer in a consulting role with QPQ.IO. His core research interests include large-scale resilient distributed systems, information security, and applications of data analytics. He is exploring topics at the intersection of computer science and public policies and regulations along with the wider societal and (cyber) security impact of technology. This includes the topics of social media and network analysis, privacy, cyber-risk analysis and management, cryptocurrency forensics, the governance of disruptive technologies, and impact and use of disruptive technologies in digital societies and government.

ADAMAS AQSA FAHREZA is currently a Researcher with the School of Computer Science and Engineering, Nanyang Technological University, Singapore.

FRÉDÉRIQUE OGGIER is currently an Associate Professor with the Division of Mathematical Sciences, Nanyang Technological University, Singapore. Her interests include algebra and number theory and their applications to coding theory and security.