IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 13 September 2022, accepted 1 November 2022, date of publication 4 November 2022,
date of current version 15 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3219875

SURVEY

Concurrency Control and Consistency Over Erasure Coded Data

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

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

The work of Anwitaman Datta was supported by the Ministry of Education (MoE), Singapore, under its Academic Research Fund Tier 1 though the Project Title ''StorEdge: Data Store Along a Cloud-To-Thing Continuum with Integrity and Availability'' under Project 2018-T1-002-076. The work of Frédérique Oggier was supported by Nanyang Technological University, Singapore, Start-Up Grant.

ABSTRACT For over a decade, erasure codes have become an integral part of large-scale data storage solutions and data-centers. However, in commercial systems, they are, so far, used predominantly for static data. In the meanwhile, there has also been almost a decade and a half of research on mutable erasure coded data, looking at various associated issues, including update computation, concurrency control and consistency, which has led to a variety of reasonably mature techniques. In this work we aim at curating and systematizing this knowledge on managing mutable erasure coded data. We believe the time is right, both because of the richness and maturity of the literature itself, and also, given the pervasiveness of erasure codes in data-centers, because it is natural to expect a transition to accommodate mutable content using erasure coded redundancy in order to support more diverse and versatile overlying applications, while benefiting from the advantages (particularly, that of significantly lower storage overhead) of erasure codes.

INDEX TERMS Concurrency, consistency, distributed storage, erasure codes, mutable data, survey, tutorial.

I. INTRODUCTION

Large scale storage systems rely on data distribution and redundancy to achieve scalability and provide fault-tolerance in the event of failure of network components or storage devices. At the data level, redundancy may be realized by the means of replication or alternatively, erasure coding. Prior to an explosion of the data volume to be stored, replication has been the traditional choice of data redundancy. As of 2010, researchers were still in a somewhat reluctant and exploratory phase vis-a-vis erasure codes [1], wondering whether they are suitable for data centers. However, considering on the one hand the storage overheads incurred by replication, and on the other hand, the significant reduction in storage overhead for the same or even superior reliability provided by erasure codes, the latter became pervasive in data centers shortly thereafter, see e.g. [2] for Google architecture, [3] for Microsoft Azure, [4] for Facebook, and [5] for Baidu.

The associate editor coordinating the review of this manuscript and approving it for publication was Theofanis P. Raptis.

Having redundancy, whether in replicated or erasure coded form, means that when some data is updated, then all forms of redundancy involving this content must be updated accordingly. This is not a trivial process. Every node in the storage system is busy handling numerous tasks and thus might not update data instantaneously, on top of being subject to failures and communication outages. It is not clear (and in fact not true) that at any given time point, all nodes have the same updated content. This is the issue of consistency: when a client user or application is requesting to access or modify data, the storage system needs to agree on the content to be returned or stored. The challenges are accentuated under concurrency, when multiple clients may try to access/update the data simultaneously. Techniques for concurrency control and consistency over replicated data are mature. Already by 2010, there were over three decades of research on these topics, see e.g. [6].

Depending on the application needs and user behaviors, different stringency of consistency could suffice or be necessary. As a first example, consider a collaboratively written cloud hosted document. If the latest update of some data

becomes unavailable to some users due to failures or latency in the system, while such occasional glitches leading to users working on stale copies may be detrimental to the user experience and (depending on the egregiousness of the glitches) even to the reputation of a cloud service provider, it does not entail the same consequences as losing or corrupting a financial transaction for a banking system. Suppose two financial transactions are being requested at the same time, each trying to add the amounts of x and y respectively to an account with an initial amount of z. The legitimate outcome of these operations should lead to a final balance of $x + y + z$. However, if each process reads the value z before the other's addition, and computes and writes the final amount by adding x or y respectively to z, then, in absence of a proper control and sequencing of the operations, the process that last writes the content of its transaction will overwrite the addition of amount done by the other process, resulting in a wrong final amount of $x + z$ or $y + z$. These scenarios demonstrate that different applications may or not tolerate different extent of inconsistencies, and accordingly, need to manage concurrency of operations differently.

Managing data mutation over erasure coded redundancy is complex. When an update is effectuated, erasure coded pieces need to be recomputed, in a consistent manner when different write processes happen concurrently. This implies the need for keeping track of versions, the ability to establish an ordering of the operations, to recompute redundancy efficiently, and yet to be able to support possible failures. That is why most existing real-world deployments focus on storing cold (that is infrequently accessed) data, and particularly static data that does not mutate, while replication based redundancy is preferred for hot and dynamic data. However, the past decade has also witnessed a considerable body of work addressing many of the prior gaps (see [7], for arguments advocating for erasure coded storage systems to go beyond archival storage by providing better update methods and provision of multiple levels of consistency). Erasure coding for mutable content is thus becoming practical (at least for warm data, where update rates are low or moderate, even if not yet for hot data with high rates of change). We are at a cusp. Similar to the transition roughly a decade back from purely replication based redundancy to wide-spread use of erasure codes for storing (static) data, we expect another transition in the horizon, where even mutable content is increasingly stored with erasure coding. To that end, this paper aims at systematizing the knowledge around the piece-meal techniques for managing mutable content stored with erasure coding, exploring the issues of concurrency and consistency.

Survey organization and contributions: This survey comprises two logical parts. Section II serves both as background and a high-level survey: we first introduce erasure codes, how they are used in distributed storage systems, and detail the notion of granularity at which individual (data read or write) operations might be carried out, which in turn results in a number of choices with which re-computation of updated data can be carried out for one data update in isolation. We then define consistency semantics, and systematize properties of the reviewed works along three axes: (i) the properties or functionalities being achieved, e.g., the nature of consistency, limits on concurrency, the various associated overheads, (ii) the techniques used to achieve those, and (iii) the underlying assumptions and system model which determine the practicality or impracticality of various approaches. This yields a structured survey of the reviewed works, resulting in a taxonomy exhibiting which forms of consistency are achieved, under which architecture, system models and assumptions; how they are positioned with respect to each other.

Sections III, IV and V form the second logical part and heart of this survey, where we review individual works, grouped first as per the nature of consistency guarantees achieved by the works, and then by the overall characterizing mechanisms. The treatment here is deliberately detailed to ensure that nuances across similar approaches can be discerned, even as we abstract out the specific implementation details and focus on the families of approaches as per the underlying design principles. We conclude in Section VI, where we transcend the individual works and instead delve into issues of design philosophies pertaining broader issues, e.g., how the concurrency control and consistency mechanisms for erasure coded systems borrow ideas yet (need to) differ in details with respect to replicated systems.

II. ERASURE CODED REDUNDANCY, CONSISTENCY AND SYSTEM MODELS

A. ERASURE CODES

Erasure coding is a process where a collection of information symbols $\mathcal{D} = \{d_1, \ldots, d_k\}$ is mapped into a collection of **encoded** symbols $\mathcal{E} = \{e_1, \ldots, e_n\}$, $k < n$, where the encoded symbols are linear combinations of the information symbols. The notation (n, k), or the term (n, k) code, is used to emphasize both parameters. When all the information symbols are part of the encoded symbols, i.e. $\forall x \in \mathcal{D} : x \in \mathcal{E}$, the code is called **systematic**. This is desirable in storage systems, since the information symbols, if/when they are available, allow for immediate access to the original data. We shall only consider systematic codes, and thus suppose that the encoded set of symbols is of the form $\mathcal{E} = \{d_1, \ldots, d_k, c_1, \ldots, c_{n-k}\}$ where the information symbols $d_1, \ldots, d_k$ are referred to as **systematic pieces**, while the linear combinations

$$c_j = \sum_{i=1}^{k} \xi^{j,i} d_i, \; j = 1, \ldots, n-k, \tag{1}$$

of the systematic pieces d_is are called **parities**. Replication is a special case of erasure coding, with $k = 1$ and $c_j = d_1$, $j = 1, \ldots, n$, where n determines the number of replicas.

The linear combinations are designed to allow erasure/error recovery capabilities. Suppose some of the symbols become unavailable, because they are either erased or corrupted. Then, it should be possible to reconstruct $\mathcal{E}$ through

a **decoding** algorithm (we speak of 'erasure' or 'error-correcting codes' respectively), provided that sufficient information is still available. This paper focuses on erasures, in which case at most $n-k$ erasures can be tolerated, meaning that after experiencing the worst tolerable amount of erasures, k arbitrary symbols from $\mathcal{E}$ are left. An (n, k) code such that $\forall \mathcal{E}' \subset \mathcal{E}$ of size $|\mathcal{E}'| = k$, $\mathcal{E}'$ contains enough information to reconstruct the erased symbols is called a **maximum distance separable** (MDS) code. MDS codes provide the best trade-off between storage overhead (measured as n/k) and reliability, particularly in low and moderate churn environments [8].

B. ERASURE CODES FOR DISTRIBUTED STORAGE SYSTEMS

In the context of distributed storage systems, a data object is split into k symbols, encoded with an (n, k) code (or replicated), after which the encoded pieces (we may use interchangeably block/piece/symbol) are dispersed.

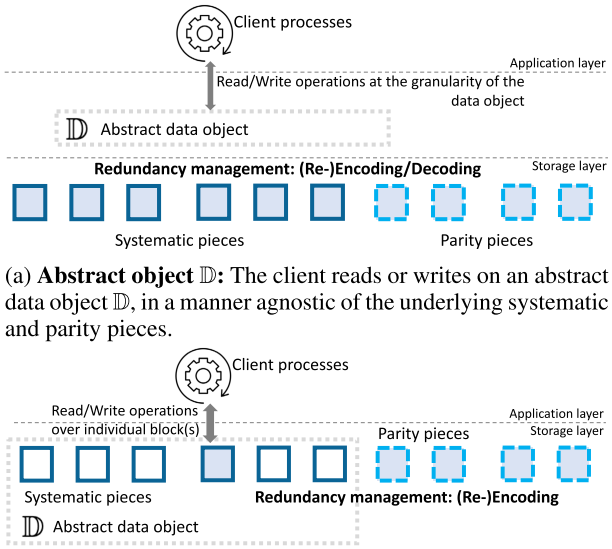
1) PLACEMENT

The n encoded pieces of a data object are stored in n distinct storage nodes, to avoid losing more than one piece of redundancy, should one node fail. Dispersing strategies are also applied across different racks, or even across multiple data centers in different geographic locations (e.g., [4]), in order to reduce the chances of data loss in the event of correlated failures. Furthermore, many data objects are coded and stored in $N \geq n$ storage devices. Choices are then made on how to collocate the pieces of several objects, which have implications on load-balancing and performance. Those issues are however orthogonal to this study. Here we consider the management of n encoded blocks from a single data object (which includes the case of an (ln, lk) code, where n groups of l pieces are formed, each l pieces are co-located and can be grouped as one symbol) and the n nodes store these encoded blocks at given time points. Because of churn in the system, induced e.g. by node failures, these n physical nodes may actually change over time, an issue considered transparent, unless stated otherwise. We will thus refer to an encoded piece and the node where it is stored interchangeably.

2) REPAIR

Since the n encoded pieces of a data object are each allocated to a distinct node, a node failure is akin to an erasure, and the erasure recovery ability of code ensures the missing piece of data can be **repaired**. It is important for systematic pieces, so that read requests can be honoured, but the reconstruction of parities is also critical to the maintenance process, ensuring a healthy amount of redundancy over time. The process of being able to 'read' a missing systematic piece by accessing other systematic and parity pieces and performing an on-demand re-construction is called a **degraded read**.

Recovering any erased symbol requires the access to k other symbols for an MDS code, associated with the corresponding disk I/O operations, network bandwidth and computation costs, referred to as repair costs. Studying repairs of specific symbols using smaller (than k) subset(s) of symbols to reduce repair costs has been extensively researched, leading to the so-called 'local repair problem', and the creation of non-MDS **locally reconstructable/repairable codes (LRC codes)**. See e.g, [9], [10] for surveys on the design of erasure codes tailored for distributed storage systems.



(a) **Abstract object $\mathbb{D}$:** The client reads or writes on an abstract data object $\mathbb{D}$, in a manner agnostic of the underlying systematic and parity pieces.

(b) **Individual systematic pieces:** The client directly reads from systematic pieces (or chosen parities in case of degraded reads), and writes only affect specific systematic piece(s) and the corresponding parities.

FIGURE 1. Logical abstractions of data storage and operations: the abstract storage $\mathbb{D}$ as a unit versus individual systematic pieces as units.

3) UNITS

To discuss the different ways in which the application and storage layers manage the data redundancy, or client processes perform read and write operations, we view a data object in two manners (illustrated in Figure 1): (a) an abstract object $\mathbb{D}$ as the unit, (b) individual systematic pieces as units.

When **the data object $\mathbb{D}$** is the unit, any read or write requires the (implicit) access to enough encoded pieces, for $\mathbb{D}$ to first be reconstituted. Write operations on $\mathbb{D}$ need to be followed by re-computation of the encoded pieces (the whole of $\mathcal{E}$ is affected), and repopulating the storage nodes. This approach is typically applied in systems that carry out encoding at the granularity of files, and in object storage systems which may require the file or object to be manipulated as a whole.

When **individual systematic pieces are the units**, reads are generally done without any decoding (barring the cases of degraded reads), while writes only affect the specific systematic piece(s) being written to and their corresponding parities. This fits block storage systems which either store fixed sized blocks, each containing multiple files or objects, or stores a single file or object spread across multiple fixed

sized blocks, but an agnostic storage layer handles data at the granularity of the individual blocks. For such systems, a collection of arbitrary k blocks can be encoded together for storage efficiency, even though operations would still be at the granularity of individual, systematic blocks. See e.g. [11], [12] for details on file, block and object based storage models.

Notwithstanding these two logical abstractions, the implementation of the (re-)encoding and decoding might be at the application layer (and may even be integrated with the clients) or at the storage layer, or possibly be realized as an explicit intermediate process.

C. CONSISTENCY

In layman's term, consistency in the context of distributed storage systems refers to the ability of having every node with the same view of data at a given point in time, irrespective of which client(s) may have updated the data. There are several forms of consistency (see e.g. [28], [29]), which have been well studied in different contexts, including replication-based storage systems. We review classical definitions of consistency, to then explain how they have been adjusted to erasure code-based storage systems.

Properties of distributed systems in terms of concurrency are categorized into safety and liveness. Safety properties specify undesirable things that should not happen, while liveness properties assert desired outputs that eventually happen for every execution. Therefore consistency properties can also be expressed in terms of safety and liveness.

To refer to a perfect time line, which may not be perceived in practice since processors are relying on clocks which are unlikely to be perfectly synchronized given possible arbitrary delays and disruptions in communication, the abstraction of a real-time/wall clock is used as as a point of reference. For a single write client holding a single value, we recall Lamport's semantics [30], defined in terms of a real-time clock beginnings and ends of operations to the object:

- **safe**: asks that a read not concurrent with any write obtains the previously written value.
- **regular**: asks to be safe, and furthermore that a read that overlaps a write obtains either the old or new value.
- **atomic**: asks to be safe, and that reads and writes behave as if they occur in some definite globally unique order.

Since clients are accessing the stored data to either read or write it (be it new data or updating existing data), we may call a client a reader or a writer respectively. Correspondingly, a single writer means no concurrency in write operations. There are four possible scenarios: single reader/single writer, single reader/multiple writers, multiple readers/single writer, and multiple readers/multiple writers. The situations of shared objects allowing multiple readers and/or writers are more complex than the above Lamport's semantics, leading to different forms of consistency, ranging from weak consistency to strong consistency, which requires a behavior as if there were a single (copy of the) stored object.

Table 1 summarizes the works that will be reviewed, together with the consistency they are achieving, ranked from the weakest to the strongest: eventual - regular - sequential - aRMW (see Sections III to V for the formal definitions). Key properties of the system models and update processes that are enabling the achieved consistency levels are summarized:

(1) Active/passive nodes: Passive storage nodes only provide basic functionalities such as storing, returning and deleting the data, responding to basic commands (e.g. pings) and storing meta-data. Active storage nodes can compute over the data and communicate with other storage nodes to establish consensus. Many data center storage appliances are collocated with computational resources, and satisfy the latter requirement.

(2) Update processes: an update could be computed either incrementally, or through an overall re-computation (see Subsection II-D for more details).

(3) In-place updates vs. storing older versions: In-place update means overwriting the older version of data with the latest one. Alternatively, copies of older versions are retained, and the new data is stored separately as tentative candidates; the older copies and out-of-synch candidate data may subsequently be garbage collected e.g., after reconciliation processes to establish the update ordering.

(4) Different system models are considered, in terms of their synchronicity, fault models, and how computation and coordination are performed, either centrally or in a distributed manner (see Subsection II-E).

(5) Non-/blocking algorithms: Non-blocking algorithms allow operations to be carried out by multiple processes optimistically, carrying out reconciliation of conflicts in the background. Blocking algorithms rely on allowing only a single process to carry out a write (possibly read) operation, while blocking other processes e.g., via locks. Moreover, some mechanisms navigate a middle-ground, where a weak form of liveness is guaranteed and read operations can progress provided that there are a (parameterized) finite number of writes, a.k.a., finite-write termination.

Other features that are less prominent but are still worth being noted, such as reliance on extrinsic meta-information, e.g., a record of cryptographic hash(es) of the data or parities, are indicated as remarks.

In early works discussing erasure coding for distributed storage systems, codes were implicitly assumed to be MDS. Codes such as LRC codes only appeared later on, with a better understanding of maintenance needs. The same trend is observed for existing works on concurrency control and consistency: early works mostly rely on MDS codes, or on generic codes [13], [14], [15], [25], which could be in particular either MDS or LRC codes. Only [27] focuses specifically on LRC codes. While the approach in [24] is general enough to work for arbitrary codes, the existing implementation employs $n = k + 1$ MDS codes since it was geared towards multi-datacenter deployments, i.e., only a single parity was actually used for cross data-center dispersal.

TABLE 1. Summary of the reviewed works, with the consistency they achieve: The column 'arch' (for architecture) contains two attributes, respectively indicating computation/coordination (Centralized or Distributed), and the coupling between end-user clients and the rest of the storage system, viz. strongly coupled (sC), weakly coupled (wC) and decoupled (dC). 'FS' in the column 'failure' stands for fail-stop. Entries with symbols (e.g. †) have explanations for corresponding caveats, with the same symbol, in the 'remarks' column.

results		update processes				model				
ref.	consistency	storage	unit/update	in-place	arch.	synch.	blocking	failure	code	remarks
	Weak									
[13]	eventual	passive	$\mathbb{D}$ re-compute	no	CC sC	timed-asynch.	yes (write)	byz (store)	any	multi-cloud, cryptographic hash
[14]	regular on $\mathbb{D}$	active	indv. pieces distr. variant 2	yes	DD wC	timed-asynch.	no	FS (store)	any	atomic swaps at storage nodes
[15] -opt	regular on $\mathbb{D}$	active	indv. pieces distr. variant 1	yes	DD dC	timed-asynch.	no	FS (store)	any	systematic nodes decide ordering
	Strong									
[15] -pess	sequential	active	indv. pieces distr. variant 1	yes	DC sC	timed-asynch.	yes	FS (store)	any	centralized locks & versioning
[16]	sequential	active	$\mathbb{D}$ re-compute	no	CD sC	asynch.	no	byz (store) byz (client)	MDS	cryptographic hash
[17]	sequential	active	$\mathbb{D}$ re-compute	yes$^{\odot}$	CD sC	asynch.	no	byz (store) byz (client)	MDS	$^{\odot}$buffer, homomorphic fingerprinting
[18]	sequential	active	$\mathbb{D}$ re-compute	yes$^{\amalg}$	CD sC	asynch.	no	byz (store) byz (client)	MDS	$^{\amalg}$relies on atomic broadcast
[19] -A	sequential	active	$\mathbb{D}$ re-compute	yes$^{\diamond}$	DC sC	asynch.	no	FS (store)	MDS	$^{\diamond}$replicate first encode later
[19] -B	sequential	passive	$\mathbb{D}$ re-compute	no	CC sC	asynch.	no	FS (store)	MDS	larger code first trim later
[20]	sequential	active	$\mathbb{D}$ re-compute	no	CD sC	asynch.	no†	FS (store)	MDS	†bounded versions w/ concurrency limit
[21]	sequential	active	$\mathbb{D}$ re-compute	no	CD sC	asynch.	no	FS (store)	MDS	builds on [20], reconfiguration
[22]	sequential	active	$\mathbb{D}$ re-compute	no	CD sC	asynch.	no	FS (store)	MDS	builds on [21], larger code
[23]	sequential	passive	$\mathbb{D}$ re-compute	no	CD sC	asynch.	no	byz (store)	MDS	bounded versions, SWMRs for meta-data
[24]	sequential	passive‡	$\mathbb{D}$ re-compute	no	CD dC	asynch.	no	FS (store)	any	cross data-center, ‡active meta-data store
[25]	aRMW§	active	indv. pieces distr. variant 1	yes	DD wC	timed-asynch.	yes	FS (store)	any	§relies on (distributed) locks
[26]	aRMW§	active	indv. pieces distr. variant 1	yes	DD wC	timed-asynch.	yes	FS (store)	MDS*	*limited choices of code parameters
[27]	aRMW§	active	indv. pieces distr. variant 1	yes	DD wC	timed-asynch.	yes	FS (store)	LRC*	*specific code class

D. UPDATING THE SET $\mathcal{E}$ OF ENCODED BLOCKS

Whenever there is any change in the stored data, this change needs to be reflected in both the systematic piece(s) and the corresponding parities of the affected data.

There are different ways in which such updates are effectuated, which we will broadly categorize into:

- **Overall re-computation** from scratch of all parity updates: this is applicable both when $\mathbb{D}$ is treated as the unit and when individual systematic pieces are the units.
- **Incremental computation** of individual parity updates: this naturally fits individual systematic pieces as the units.

We describe the simplest case where the only process involved is a single client operating an update. How to handle variations in case of simultaneous multiple updates by different client applications, or attempts to read the data during (an) ongoing update process(es), will be treated later, when consistency mechanisms are discussed.

1) OVERALL RE-COMPUTATION OF ALL PARITY UPDATES

Suppose x pieces of the stored object need to be updated. Overall re-computation means that its k systematic pieces need to be present, among which x need to be updated, for all the parities to be re-computed. Since the data is dispersed and no single entity possesses all the k systematic pieces, re-computation typically involves a centralized entity, which starts by fetching enough information. We will assume the usage of an MDS code, and that enough information means reading and transferring k pieces from the storage nodes. This is because the centralized entity may optionally recompute (by decoding) the systematic pieces, if some systematic pieces were missing. This is a pessimistic scenario; it could also be (as a best case scenario in terms of data transfer requirements) that we have the x new pieces independent of the previous ones, and the centralized entity could just transfer the $n-x$ systematic pieces that do not need updating, which happen to be available at live storage nodes. For updating the x systematic pieces, the centralized entity (i) recomputes (by encoding) the $n-k$ new values of the parities, and (ii) transfers and writes back the $n-k$ newly computed parities at the corresponding storage nodes, and likewise transfers and writes the new value of the x updated systematic piece(s). All in all, this requires $k+n-k+x = n+x$ data transfers over the network, k blocks of

read and $n-k+x$ blocks of write operations (involving disk I/Os) at storage nodes, in addition to the computations for re-encoding (and the optional decoding). Re-computing each parity piece in step (i) involves k multiplication and $k-1$ addition operations.

For large volumes of changes, affecting multiple systematic blocks, this approach can amortize the costs of updates. However, for small updates, an incremental approach to compute the parities, which we discuss next, is more efficient.

2) INCREMENTAL COMPUTATION OF INDIVIDUAL PARITY UPDATES

Let $\mathbb{D}$ be a data object represented as a collection of the blocks $\mathcal{D} = \{d_1, \ldots, d_k\}$. Recall from (1) that the parities are computed based on the following linear combination of the systematic pieces: $c_j = \sum_{i=1}^{k} \xi^{j,i} d_i, j = 1, \ldots, n-k$.

Suppose $\mathbb{D}$ undergoes a change which only affects the block x, i.e., d_x changes, and its updated value is d'_x. Then all the parity pieces using d_x need updating.

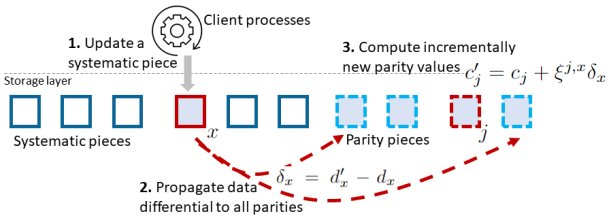
Observing that only the systematic piece d_x has changed [14], [15], [25], [26], [27], [31], one can recompute the new parity values incrementally without the need to carry out a full computation, as $c'_j = c_j + \xi^{j,x}\delta_x$ where $\delta_x = d'_x - d_x$. This involves a one time computation of δ_x which is effectively a single addition operation, and the new value for each parity can be computed using a single multiplication and addition operation, as opposed to the k multiplications and $k-1$ additions required above to recompute a parity from scratch.

The volume of data to be read and transferred is also substantially lower than the approach of overall re-computation, when a single systematic piece is updated, however the exact volume further depends on how the incremental update is actually orchestrated: either in a distributed manner by delegating computations to active storage nodes if available, or in a centralized manner at the writer client (see Figure 2 for an illustration of these variants).

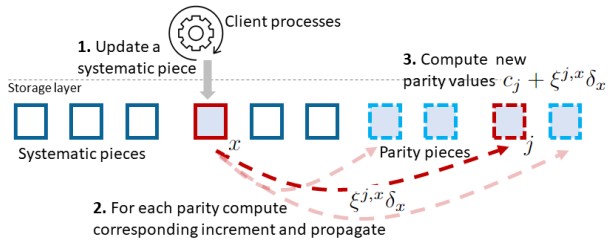
a: DISTRIBUTED VARIANT 1

The affected storage node x, or alternatively, the write client (not shown in Figure 2a) determines and transfers δ_x to the $n-k$ parity nodes, which then compute their respective $\xi^{j,x}\delta_x$ values locally, carry out a read of the locally stored c_j, and replace it with the updated c'_j. This approach has been used in e.g. [15], where one subvariant uses the client, while another subvariant uses a systematic storage node to disseminate the differential to the parity nodes. The works [25], [26], [27] also use this approach, but they are ambivalent regarding the entity which initially disseminates the differential among (some of) the parity nodes, and they additionally apply a gossip mechanism among parity nodes to continue the dissemination.

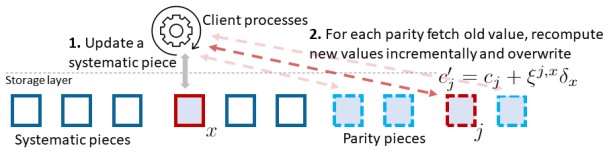
The write client needs to read and transfer the old value of the systematic piece from the corresponding storage node, and transfer back the new value (or alternatively, δ_x). In addition to these two data transfers, for MDS codes, there are a



(a) **Distributed variant 1**: $\delta_x = d'_x - d_x$ is sent to every parity node.



(b) **Distributed variant 2**: $\xi^{j,x}\delta_x$ is sent to parity node j.



(c) **Centralized variant**: $c_j + \xi^{j,x}\delta_x$ is sent to parity node j.

FIGURE 2. Various ways for incremental parity updates.

further $n-k$ transfers of δ_x to the $n-k$ parity nodes (note that δ_x is of the same size as a single block of data), which read their local (parity) data, compute and write back new updated values. This involves reading (as well as writing) $n-k+1$ blocks of data, and transferring $n-k+2$ data blocks over the network.

b: DISTRIBUTED VARIANT 2

Node x (or alternatively, not shown in Figure 2b, the client, e.g., [14]) computes individual $\xi^{j,x}\delta_x$ for each of the affected parity nodes and then transfers them to the respective nodes, which in turn use them to compute the new parity values.

Instead of conveying δ_x to all parity nodes, individualized $\xi^{j,x}\delta_x$ (whose size is that of a single block of data) are being sent. The above cost analysis thus remains the same.

This variant requires active nodes, which may not always be an option. A centralized variant circumvents this.

c: CENTRALIZED VARIANT

The write client computes the differential, fetches the parity pieces, recomputes the individual new values incrementally, and then repopulates them back at the storage nodes (see Figure 2c), in addition to repopulating the updated systematic piece. For MDS codes, this involves $n-k+1$ data reads and data writes, and $2(n-k+1)$ blocks of data transfer. While the centralized variant has data transfer overheads almost double with respect to the distributed variants, it not only accommodates passive storage nodes but also is much simpler

TABLE 2. Costs in terms of blocks of data transfer and read/writes, when a single systematic piece is updated.

	transfer	I/O (read+write)
Overall re-computation	$n+1$	$n-k+1$
Incremental (distributed)	$n-k+1$	$n-k+2$
Incremental (centralized)	$2(n-k+1)$	$2(n-k+1)$

from system design perspectives, see e.g., [32] for a non-MDS code.

In many practical deployments, $n-k$ is often (significantly) smaller than n. For instance, in Facebook's f4 system [4], discounting cross-geographic redundancy, an MDS Reed-Solomon code with $k = 10$ and $n = 14$ (so, $n - k = 4$) is used within a single data-center. For small writes affecting one or few systematic blocks, the incremental approaches result in substantial savings in terms of data read/write operations, network transfers and computations; as opposed to the approach entailing an overall re-computation, which is most beneficial when an update involves changes to multiple systematic pieces. In Table 2 we summarize the costs in terms of network traffic and disk I/O operations when an update modifies a single systematic piece.

E. SPECTRUM OF SYSTEM MODELS

The underlying assumptions on how nodes interact among each other, and how their actions might depart from the required behavior play crucial roles in determining the design as well as efficacy of any given technique. We describe and categorize below available system models, depending on their coordination techniques, fault models, synchronicity assumptions, and their client-storage interactions.

A common abstraction we adopt is that the clients doing read/write are logically independent from nodes providing storage services, even if they are sometimes co-located.

1) ARCHITECTURE: COMPUTATION & COORDINATION

Computation and coordination of actions among client, storage nodes and any other auxiliary system components could follow different strategies. We categorize them by introducing a taxonomy of architectures which disentangles the data compute and control planes.

Some nodes could carry out all the computations for an operation. For an update, this means determining the values of the updated encoded pieces (possibly by first fetching the previous values from various nodes) and populating the affected nodes with this information, as described in Subsection II-D. Coordination may then be realized in two ways:

(i) Centralized computations and coordination **(CC)**: Within a given time period (always, or for a time window), all operations are centralized at a unique node, e.g., one of the storage nodes, client or another extrinsic entity, which determines the ordering of read/write operations. In replicated systems, having a primary replica which actively manages the concurrency and consistency issues, while the other replicas copy its data as is, fits this approach. For erasure coded systems, the same idea can be used, even though bit-wise copying of the data is no longer an option, and instead, the centralized node has to re-encode the pieces to be stored.

(ii) Centralized computations, but distributed coordination **(CD)**: Within the same time period, different operations may be centralized, possibly at different entities. The affected entities may coordinate among themselves using a distributed protocol, e.g., a consensus algorithm to establish an ordering of operations to achieve consistency.

Alternatively, together with distributed computations, coordination may be realized either in a centralized or decentralized way, called respectively centralized coordination **(DC)**, or distributed coordination **(DD)**.

In the latter, nodes compute information in a distributed fashion, and coordinate among themselves using distributed algorithms, e.g., to establish a sequence of operations via a consensus algorithm, or to rely on a quorum of nodes to finalize decisions. The two distributed variants for update computation of Subsection II-D fall within the DD category, since storage nodes are computing their updates, even though in distributed variant 2, a central node also does perform some computation. As long as any aspect of either computation or coordination is distributed, even if there are some other aspects that rely on centralization, we will designate such hybrid approaches as distributed.

2) FAULT MODELS

Participating nodes, clients and/or storage nodes, may fail. Communication among (a subset of) storage nodes may be disrupted, as could communication between the overlying processes and the storage layer.

We distinguish two kinds of failures: (i) non-byzantine failures, comprising **crash failures** which refer to an entity not responding, and include the particular case of **fail-stop** failures, where it is further assumed that properly functioning entities are able to detect that a particular entity has failed, versus (ii) **byzantine** failures, which encompass arbitrary failures from an entity, including adversarial ones, where the entity not only does not do what it is supposed to do, but also has behaviors that it is not supposed to have. For a more nuanced (but general) discussion on various failure modes in distributed systems, we refer to [33].

The content of a storage node becoming inaccessible is considered a **fail-stop storage node** scenario. For erasure coded data, such failures lead to 'erasures' of some of the pieces. A storage node ceasing some computations and/or coordination activities also fits the fail-stop setup. In either case, the storage system may decide to repopulate a different node with the corresponding data. Alternatively, the original node may return back online, and continue its usual functions. This is a 'repair problem' [9], [10], which, unless specifically addressed by a technique for achieving consistency, will not be further discussed.

In contrast, a storage node deviating from its expected behaviour in any manner (other than simply stopping to respond), e.g., returning a wrong value, or introducing errors

while carrying out any computation, is a **byzantine storage node** fault. Error-correcting codes could be considered, though extrinsic meta-data based integrity mechanisms, e.g. hashes, can check which pieces are intact; erroneous pieces are then discarded, thus reducing faults to erasures.

A **byzantine client** may have arbitrary behaviors. Examples include populate the system with data which is inherently erroneous, e.g., the pretended systematic and encoded pieces do not conform with the underlying coding scheme, or with erroneous meta-data, e.g., tampered versioning information to confound the legitimate system participants.

3) A/SYNCHRONOUS SYSTEMS

Another crucial characteristic is the synchronicity of nodes and communication. Roughly speaking (see e.g. [33] for formal definitions), if there is a known bound on the difference of the clock times at individual nodes, and likewise, if the communication delay is bounded, then the system is considered to be **synchronous**. In the absence of any such bounds, the system is considered **asynchronous**.

In an asynchronous system, even determining whether a given node has failed and stopped is difficult. Since the synchronous setup is infeasible in practice, while it is often impractically complex to deal with a fully asynchronous setup, many pragmatic solutions operate under an 'in-between' model, called the **timed asynchronous** model [34] which uses time-outs as a proxy for failure detection. For instance, Redis Labs' describe their distributed locking service [35] as one which "...relies on the assumption that while there is no synchronized clock across the processes, still the local time in every process flows approximately at the same rate, with an error which is small ...".

4) CLIENT-STORAGE INTERACTION

Based on the interactions required among clients, or between clients and the storage layer, for the storage system to function, we classify architectures into three categories. The two extremes are: (i) decoupled (**dC**), where clients treat the storage as a totally independent service, thus supporting 'thin clients', and (ii) strongly coupled (**sC**), where the clients direct involvement in complex tasks and interactions with the storage layer and/or other clients are necessary to the overall system. An in-between situation is (iii) a weak-coupling (**wC**), such that, (reasonably) thin clients can still be deployed, while a 'proxy' middleware or a supplementary service may orchestrate some moderately complex tasks. These distinctions are arguably subjective, and for individual works, we will elaborate our designation choices.

The treatment of byzantine clients is an example of the role played by the client-storage interaction. Some works explicitly address this issue, while others follow a principle of separation of concerns, viewing byzantine clients as an application layer issue, disentangled from the storage layer. While the former approach has the apparent advantage of a holistic solution encompassing all system layers, by codifying the behaviour of benign clients, it also imposes an additional burden every application developer would have to adhere to. The latter has practical advantages of modularity and portability, i.e., by being disentangled from other systems, the storage system provides functionalities and guarantees unconditionally, enabling diverse overlying applications.

F. MULTI-VERSIONED DATA

Instead of storing only the latest version of the data, one may also want to retain all its versions over time. This could be particularly important for archival storage and provenance oriented applications. Sparse sampling based techniques [36], [37] to encode the differentials across adjacent versions in a storage efficient manner have been explored. However [36], [37] do not explore the issues of consistency, concurrency control and compression of multiple versions in conjunction, and that remains an open issue. Thus, further discussions on these works are out of the current scope.

Some other works e.g., [16], ORCAS-B in [19] or [20] happen to store multiple (but not necessarily all) versions, as a means and by-product of establishing consistency, however they do not provide any guarantees in terms of retaining all versions or a deterministic period of version history and are thus not really applicable if the purpose is multi-versioning.

III. APPROACHES WITH WEAK CONSISTENCY

We start by reviewing three works that achieve weak consistency guarantees, namely eventual consistency and regular semantics. In subsequent sections, we discuss approaches achieving strong consistency. An overall taxonomy of the different works reviewed is shown in Figure 3.

A. EVENTUAL CONSISTENCY

Eventual consistency specifies as guarantee that if a data update is made, then eventually it will be reflected at all the nodes where the information is stored redundantly. This means that a read query before eventual consistency is arrived at can yield an arbitrarily outdated result or even something completely random that was never written by any operation [38]. At the granularity of $\mathbb{D}$ as a unit, this guarantee is subject to accessing any appropriate subset of nodes, so that the data can be reconstructed. This is an optimistic approach, where availability is prioritized over consistency, since, like in the replicated case, different contemporaneous read operations may yield different values for the data object in the transition period until eventual consistency is established.

In [13], the default configuration considers a single writer which stores data dispersed in a multi-cloud setup, i.e., the storage system is a collection of cloud services (where each service is treated as a 'node'). Because of their heterogeneity, cloud services are considered as **passive storage** and a single writer centrally computes and coordinates (**CC**) all the actions. Operations are done at the granularity of ensembles of blocks, i.e., $\mathbb{D}$ **is the unit**, every encoded piece is stored in a different cloud, and any update leads to changes in the whole of $\mathcal{E}$. A single writer means there is no write concurrency, and a quorum of cloud services determines whether an update has

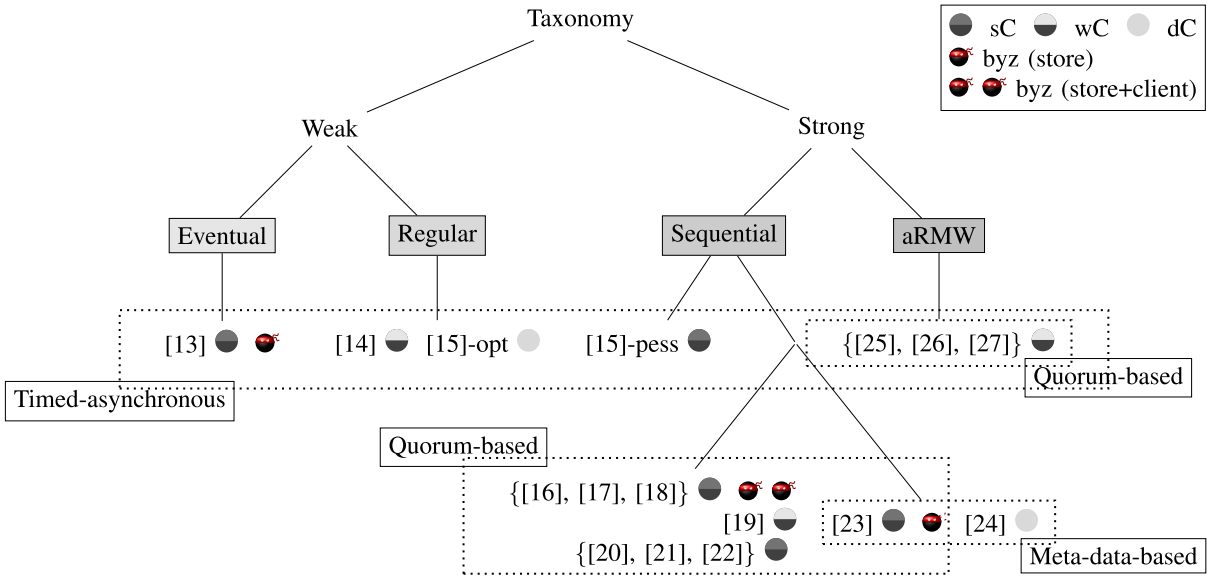


FIGURE 3. A taxonomy of the works reviewed. Abbreviations are those of Table 1.

propagated adequately: if enough cloud services acknowledge their update is complete, the writer considers the overall update done. This achieves the weak form of **eventual consistency**, even if individual storage nodes undergo **byzantine failures**. **Fail-stop failures of the writer** are considered. Note that each cloud service may internally maintain multiple copies of the stored data, and serve stale data until their internal redundancy is up-to-date, a process that is transparent to the mechanisms in [13]. Since cloud services often provide only eventual consistency guarantee, [13] cannot guarantee any stronger consistency.

In order to accommodate multiple writers, [13] uses time leases by effectuating **locks**, and thus, **blocks** different writers to write simultaneously, while, given the weak consistency requirement, reads are **not blocked**. The writers coordinate the leases among themselves using a single cloud storage service with regular storage semantics, to acquire and release leases. Several other issues such as data confidentiality are also addressed in [13], and all in all, the clients need to carry out many complex tasks, explicitly interacting with the storage elements. As such, we designate the client-storage architecture as strongly coupled (**sC**).

Both single-writer byzantine fault tolerant quorums and write locks based on leases assume synchronized clocks and bounded communication delays, which is realized using a **timed asynchronous setup**, i.e., using timeouts.

B. REGULAR SEMANTICS

Regular semantics for multiple readers is a generalization of Lamport's regular semantics for a single reader. For a single writer multiple reader setup, it is achieved, if (i) a read operation never returns a value that was never written, or a value that has been overwritten by a write operation; however, (ii) in the event of concurrent read and write operations, the read might return the value of the contemporaneous write or the immediately previously written value, while (iii) the writes themselves follow a total-ordering consistent with real-time ordering at the given writer. The second clause can be further generalized to accommodate multiple concurrent writes, in which case, contemporaneous read operations may return the value from any of the writes, or the previously written value.

1)

In [14], operations **at the granularity of individual systematic pieces** are carried out. For update computations, writer clients follow the distributed variant 2 strategy from Subsection II-D which requires **active storage nodes**, and compute **differentials of parity updates**. Knowing the coding strategy, clients use an atomic 'swap' operation at the corresponding storage node to replace a systematic piece to be updated (i.e., an **in-place** data replacement), while computing centrally the differentials for each parity with respect to this systematic piece.

To support multiple writer concurrency, the nodes storing systematic pieces create a local version identifier for each write operation, and during the swap process, this information is returned to the writer. For concurrent operations on the same systematic piece by two processes, the storage node determines an ordering locally, identified by a version id (or 'operation id', as named in the paper). The writers send the version id with parity differentials. A parity node then allows an 'add' operation only if it has already encountered the update for the immediately preceding version for a given systematic piece, ensuring that 'add' operations corresponding to concurrent updates on the same systematic piece are carried

out in the same order at every parity node. Since writes at all parity nodes are handled by the clients themselves, clients may have to repeatedly attempt to write the update, until the corresponding 'add' request is accepted. Writes are thus lock free in the sense that multiple clients can write distinct data blocks simultaneously, and there is no guarantee of a globally unique sequence for those operations; hence, at the granularity of $\mathbb{D}$, at best, **regular semantics** can be achieved. However, if different clients try to write concurrently on the same block, the atomic swap ensures an ordering at the nodes storing systematic pieces themselves, while the local version ids created by the systematic storage node lead to the imposition of the same ordering for 'add' operations at the parity nodes. Coordination being thus mostly distributed (in addition to the 'add' computations being distributed), we deem this approach to have distributed computation and distributed coordination (**DD**), even though it also contains central computations. While the clients compute the differentials, interact with storage nodes for 'swap' and 'add' operations, in principle, a proxy client or middleware could carry out these tasks on behalf of end-user clients, we thus designate this system to be weakly coupled (**wC**).

To support data recovery from **fail-stop** failures, each write is associated with a further unique identifier determined by the client, which is retained as meta-information at all parity nodes during the 'add' operations. This enables any client to carry out a recovery of the data at new replacement nodes, should some storage node fails. This is done through a multi-phase protocol, which first locks and blocks all writes, then tries to obtain a quorum of up-to-date nodes identified by their write operation identifiers. If a quorum cannot be achieved, because the lock triggered by the recovery process prevented some updates to be propagated to an adequate numbers of nodes, then the protocol only allows these interrupted but ongoing update writes to finish by partial release of locks. These computations are carried out centrally. Any such recovery from failure is registered as an epoch, and all unfinished updates from previous epochs are discarded, allowing for only new operations to be executed after a repair. In [15], which we discuss next, the commutativity of the increment operations for parity computation is exploited, simplifying the mechanisms. In particular, [14] tries to ensure that incremental updates due to changes to the same systematic piece are executed in the same order at every parity node; yet, it does not enforce an ordering when parities are updated because of changes to different systematic pieces. Instead, a separate mechanism ensures all pending updates of parities because of changes to different systematic pieces are finished within a quorum of nodes used for data recovery. Only the latter could have sufficed to update parities (as is the case in [15]), without differentiating whether the updates were for some or different systematic pieces.

Failure detection, locking and unlocking mechanisms rely on an implicit **timed asynchronous** model assumption.

In the absence of storage node failures, the protocol executes in a **non-blocking** manner, even though node failure recoveries require blocking write operations. Furthermore, depending on how the atomic swap capability at each storage is effectuated in practice, it may implicitly result in blocking.

No proof of liveness is given, but experimental results are provided to benchmark the system's performance, and no correctness issues were reported.

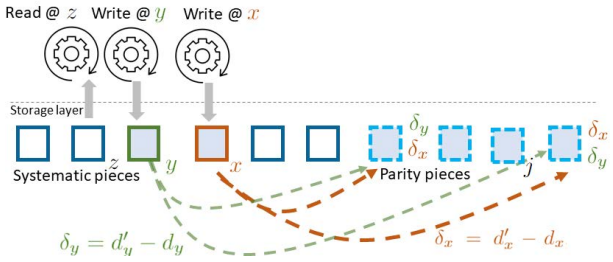


FIGURE 4. An optimistic update mechanism from [15].

2)

In [15], a so-called 'optimistic' variant allows concurrent writes to achieve consistency akin to regular semantics at the granularity of the overall data object, which we discuss next. A variant achieving sequential consistency and named 'pessimistic' will be discussed in Subsection IV-A. In both the 'optimistic' and 'pessimistic' variants, operations are at the **granularity of individual pieces**, and updates are performed using the distributed variant 1 of Subsection II-D.

In the optimistic variant (see Figure 4), writer clients interact with a node storing a systematic piece, e.g., nodes x and y in the figure. Similar to [14], the node storing a systematic piece determines the ordering for concurrent write requests for that specific piece. Such a node is also responsible for communicating the differential of the updated systematic piece (for example, node x computes and transmits $\delta_x = d'_x - d_x$) to all the nodes storing parities, which in turn compute the differential in their respective parities and incorporate them locally. They then acknowledge the completion back to the node with the systematic piece, which in turn aggregates such acknowledgments to finalize and confirm the update to the writer client (the flow of acknowledgements is omitted from the figure to keep it uncluttered). The client's interaction with the storage layer is straightforward, and the orchestration complexities are within the storage layer itself. As such, we designate this as decoupled (**dC**).

Multiple writes at distinct systematic pieces may occur concurrently, e.g., at nodes x and y in Figure 4, and the corresponding differentials are incorporated at the parity nodes. Since addition of all such differentials at the parity nodes commute, no specific ordering on how they are imparted is imposed, i.e., for the example in the figure, one parity node may carry out the update corresponding to x before that of y, while another parity might do so in the converse sequence. Read operations of other systematic pieces, e.g., at node z, can be carried out in parallel, unhindered by concurrent write operations at x or y. The optimistic variant in [15] thus

uses distributed computing and coordination (**DD**) relying on **active** storage nodes, it is **non-blocking** in nature and carries out **in-place** replacement of data.

In the absence of any failures, if all reads are directly from the systematic pieces, this approach satisfies sequential consistency over the individual pieces since the ordering of operations are logically centrally determined by the corresponding storage node. However, across the overall abstract data object, comprising all the systematic pieces, a unique view of global ordering of operations is not enforced. As such, at the granularity of $\mathbb{D}$ as a unit, a **regular semantics** results in the absence of any failures.

Failure recovery, particularly when systematic nodes experience **fail-stop** faults, to repopulate missing data (or carry out degraded reads) depends on how many given updates have propagated. Accordingly, first, either a roll-back of incomplete operations, or otherwise, a carry ahead to complete those operations, is recommended. Arguably, all the pending updates at a parity node need to be imparted (or alternatively, all unfinished updates would have to be rolled back) before using the parity nodes for reconstruction of a valid systematic piece (possibly, of an older version, essentially effectuating an abortion of latest operations). This assumes a **timed-asynchronous** system, to enable timeout based failure detection. Many combinations of failures across clients, systematic nodes and parity nodes during the write process and subsequent roll-back or carry-forward processes may occur, which may imperil the correctness of the reconstruction or degraded read operations. This issue is left to be explored.

IV. APPROACHES WITH SEQUENTIAL CONSISTENCY

Sequential consistency is a stronger form of consistency than regular semantics and thus also than eventual consistency because it requires that across all processes, there is a unique history of read/write events which is equivalent to a sequential execution, where the local ordering of events at each node is respected. This is not the strongest consistency, because, while events are sequential and consistent at every node, sequential consistency allows operations to appear out of real-time order. Sequential consistency applies to the atomic multi-reader multi-writer (MRMW) abstraction, which is the predominant focus of the existing literature on consistency over coded data. This notion generalizes that of atomic single-reader single-writer. Consider a variable V with the value V_t at time t. When several processes (say, f and g) carry out writes to modify its value concurrently at time t_2, and in case the new value is determined based on the previous value V_{t_1}, then different candidate values $V_{t_2} = f(V_{t_1})$ and $V_{t_2'} = g(V_{t_1})$ could be proposed by processes f and g. Then, either one of $[V_{t_1}, f(V_{t_1}), g(V_{t_1})]$ or $[V_{t_1}, g(V_{t_1}), f(V_{t_1})]$ is a valid global ordering, satisfying the sequential consistency criterion. In this example, both the second as well as the third values depend only on the first value, and the third value is independent of the second value. While this semantics is strict and caters to a variety of applications, it cannot support transactional semantics [39] essential to databases.

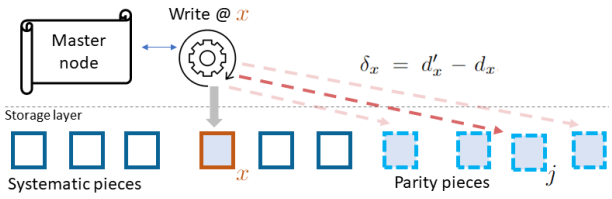


FIGURE 5. A pessimistic update mechanism from [15]: An extrinsic master node is used for locking and versioning.

The works in this section all provide sequential consistency achieving atomic Multi-Reader Multi-Writer (MRMW) abstraction. They are grouped into two groups, one that assumes a timed-asynchronous model, and one for asynchronous environments. Some works from the latter category furthermore stand-out in their use of an extrinsic entity for meta-data management, thus disentangling the control plane for consistency from the data plane for redundancy.

A. TIMED ASYNCHRONOUS

In [15], a so-called 'pessimistic variant' relies on a centralized master node for versioning and locking out concurrent writes (Figure 5). Similar to the optimistic variant discussed in Section III-B2, the operations are still at the granularity of **individual systematic pieces**. A client trying to update a piece of data contacts in parallel the node storing the piece and the master node to obtain information on the latest version. Accordingly, it determines whether the latest version of the data is available, in which case, it obtains a lock from the master node, so that no other concurrent write is feasible. This architecture thus, at a logical level, resembles the lease mechanism to accommodate multiple writers in [13]. The client then updates the systematic piece, and sends the difference object to nodes storing parity pieces, which in turn locally compute and store the updated values for the parity pieces, and then send acknowledgements (not shown in the figure) to the write process. Upon receiving all acknowledgements, the write process releases the lock. The mechanism in [15] relies heavily on centralized coordination, even though some computations are distributed (**DC**), assuming **active nodes**. A master node maintains versioning information and locks concurrent writes, yielding a **blocking** algorithm which achieves **sequential consistency**. The end-user clients manage the write locks at the master node, and determine the conclusion of updates at the storage nodes to do so. This leads to a strongly coupled (**sC**) architecture.

Failure of the master node requires the election of a new master, which then needs to reconstruct the information about latest versions by polling all storage nodes. Naturally, no write operation is feasible until a master node is reinstated. If a node storing a systematic piece experiences a **fail-stop** failure, all other operations are stopped, until the corresponding data is reinstated using a decoding process, provided that enough other nodes with the latest version of the data are available. Likewise, in case of a parity node fail-stop

failure, the data is recreated by carrying out re-encoding. If a client fails during the write process (detected by the master node using a timeout, which requires the use of a **timed asynchronous** model), then, depending on the progress of the write process, it is either rolled back, or completed by the master node. For every kind of crash recovery, further information (epoch) is maintained across the system, in order to prevent the use of stale information. Latest version of data replaces older versions **in-place**.

B. ASYNCHRONOUS AND QUORUM-BASED

We next review a series of works which share the following commonalities: (i) they assume an **asynchronous** environment, (ii) where the granularity of operations is $\mathbb{D}$ **as a unit**, (iii) and parity update is done with **overall re-computation**. They (iv) realize **non-blocking** mechanisms, (v) relying on quorums of storage nodes, but in a manner (vi) efficient only for **MDS codes**.

We recall that in the context of replicated systems, a quorum system is 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 the quorum system. Usually quorums are established to read and/or write data. The term quorum stems from the fact that read or write operations need to be performed at an adequate number of nodes for them to be successful and persist. Symmetric quorums are determined irrespective of the type of operations, while asymmetric quorums distinguish write and read quorums. Furthermore 'locks' may be attributed to each node within a quorum in order to carry out an operation, if there is a need to conditionally block other operations. Moreover, failed nodes might not participate in a quorum until not only they join the system anew, but also are repopulated with up-to-date data. Thus the quorum membership may need adequate redundancy so that intersection is guaranteed despite a number (chosen as per a fault-tolerance parameter) of failures.

Using a quorum of nodes to agree on a latest stored version, where nodes store encoded pieces from an MDS code in a manner agnostic of the structure of the code, leads to write processes not waiting for responses from all or some task specific nodes but only at a threshold of nodes. This has the following inherent drawbacks:

(i) A systematic piece affected by an update may not be populated with information for the latest version, and more generally, (ii) a given version of data may be stored with a redundancy lower than the nominal (n, k) code parameter would indicate. (iii) Since there is no guarantee that the systematic pieces are actually up-to-date for any given version, or even if they are, they may not be included in the quorum formed to read data, consequently, read operations are always carried out by invoking decoding, which in turn levies an additional burden in terms of I/O operations, data transfer as well as computation, which maybe deemed impractical in most real-world deployments. In practice, decoding based data access is in fact considered 'degraded read', which is done more as an exception than a norm. The works are distinguished based on the detailed algorithms and the choice of data structures involved in the coordination phases.

1)

In [16], before carrying out a write, the client queries a **quorum** of storage nodes, to determine the latest (logical) timestamp, and increments this timestamp to generate a new one. The client then computes a cryptographic hash $H(e_x)$ for each encoded piece e_x in $\mathcal{E} = \{e_1, \ldots, e_n\}$. These hashes are concatenated to create $H_{\mathcal{E}} = H(e_1)|\ldots|H(e_n)$, and a summary of this concatenation of hashes by computing a further hash $\hat{H}_{\mathcal{E}} = H(H_{\mathcal{E}})$ is generated. This summary hash $\hat{H}$ is appended with the logical timestamp to create a composite timestamp. The client then sends the individual encoded pieces to the storage nodes, along with the concatenation of hashes and the timestamp as version meta-data. **Active storage nodes** locally determine whether the data hash of the individual piece matches with the corresponding portion within the concatenation $H_{\mathcal{E}}$ and furthermore, check whether the hash of $H_{\mathcal{E}}$ matches the hash summary $\hat{H}$ embedded in the time-stamp. This ensures that data which is not encoded properly (poisonous write [40]) will not be written at well-behaved, i.e., non-byzantine, storage nodes. A storage node incorporates the write locally only if all these validations work. Then the storage node sends an acknowledgement to the writing client. The client needs to wait for a quorum of q acknowledgements (see (2) for relationships among q, the code parameters (n, k) and the number f of byzantine processes) to determine whether the write process is successful.

The approach in [16] thus stores **multiple data versions**, whose ordering is determined by logical time-stamps. It fits the setting of centralized computations with distributed coordination (**CD**), since the coordination is distributed, leveraging on locally stored data structures and quorums of storage nodes. Garbage collection of older versions is recommended, but is not an integral part of the proposal.

Similarly, read operations need to obtain a quorum of encoded blocks, foremost to determine the latest timestamp, then to ensure that there are adequate data blocks to decode the original data, while the integrity of the individual pieces is validated using the hash digests. If not enough pieces with the latest timestamp are obtained, it indicates an incomplete write, and the client is then expected to fetch and read a previous version (determined by logical timestamp) for which the write process had been completed.

Both **byzantine** storage nodes and write processes are accounted for by using additional (cryptographic hash based) meta-data, which is replicated at every storage node. While the mechanism provides very strong safety guarantees, in particular, a consistency guarantee which meets the stringency of at least **sequential consistency** (the authors claim linearizability), proof of liveness is not given. In particular, if multiple processes attempt to write concurrently, they may inadvertently use the same new logical timestamp. As such, it may so happen that none of them acquires a sufficiently large quorum to finish their write operations. This may recur indefinitely,

an issue not considered in [16], which suggests that if a write remains incomplete, then a read operation may have to read one of the older versions where the write operation was properly completed. However, simple heuristics can be used to address this issue, e.g., to abort write attempts after a timeout accompanied with back-off for a random period of time by the clients to initiate a new write operation.

Overall, to tolerate f byzantine (or fail-stop) failures, $n \geq 4f+1$ storage nodes (per version) are needed, while the code parameter k should be such that

$$k < q - f, \text{ for } 2f + 1 \leq q \leq n - 2f, \tag{2}$$

where q is the threshold of benign nodes (which may still crash-fail subsequently) to which a write operation writes to.[1] Multiple versions need to be stored, resulting in a high storage overhead, even though both concurrency control and consistency, as well as byzantine storage nodes, and more crucially, clients, are addressed.

[16] implicitly assumes that all encoded pieces carry equivalent amounts of information, and thus, liveliness in an asynchronous set-up only depends on how many nodes respond, not which nodes. This assumption only holds for MDS codes. The mechanism is thus inapplicable to LRCs.

A refinement of [16] is proposed in [17]. Before committing a write, storage nodes are required to determine that enough other correctly functioning storage nodes have fragments for the same write operation, and otherwise, the write operation is discarded. The quantum for 'enough' depends on the code parameters, so that the corresponding version can be reconstructed. A homomorphic fingerprinting technique is deployed, such that the fingerprint of an encoded piece yields the same output as the erasure coding of the systematic piece fingerprint. The storage nodes then determine more easily whether an ongoing update is completed (they do not need the content of the other pieces, but only the much smaller fingerprints), and accordingly discard older versions more aggressively, without the need for a separate garbage collection mechanism. The updates in [17] can thus be viewed as **in-place**, as opposed to [16], however, uncommitted data still needs to be buffered, which creates implicit though temporary storage overhead.

Both [16], [17] codify the behaviour of the (well-behaved) end-user clients and require them to carry out multiple tasks in coordination with the storage layer elements, resulting in a strongly coupled (**sC**) architecture.

Among other optimizations from [17], we note: (i) instead of populating all nodes with encoded pieces, the writer encodes and populates only $n-f$, where f is the targeted fault-tolerance threshold, which reduces both the computation and the storage cost compared to [16]; (ii) the use of message authentication codes instead of public key cryptography for client identification mitigates **byzantine** clients.

[1]The paper counts crash and byzantine failures separately, even though their impacts turn out to be identical, because of extrinsic meta-information. Here, we thus presented the bound in a slightly simplified manner.

2)

In [18], variations of the ideas from [16] are used. The logical timestamp is appended with a (unique) processor identifier, creating a strict ordering (lexicographic ordering is used over the extended time-stamp) even if the actual logical time-stamp is identical. This may result in a specific client's writes always get precedence over another's, and the practical implications to overlying applications are not explored.

A reliable atomic broadcast primitive [41] is assumed, such that the data pieces are either dispersed to all the (well-functioning) nodes in the broadcast group, or to none. Thus [18] avoids storing multiple versions explicitly, since either enough encoded pieces are stored at all the (well-functioning) nodes receiving the pieces only after ascertaining that enough other nodes also have corresponding pieces, or else the pieces are discarded. The reliable broadcast has a high communication complexity, and individual broadcasts would also utilize (temporary buffer) storage space before deciding to either deliver the data to the storage nodes, or reject these (thus masking the cost of storing multiple versions at the storage layer, while incurring storage in maintaining buffers of data in the communication layer). Similar to [16], a hash digest of the fragments determines that a correct fragment is being obtained by (and during data reads, from) storage nodes.

To prevent denial of service attacks by **byzantine** clients which might generate arbitrarily large timestamps, [18] enforces non-skipping timestamps through threshold cryptography. In the initial response to a writer with current timestamp, the servers also provide a threshold signature. A subsequent write operation is accepted by servers only if the threshold signature matches with the expected new timestamp, in which case, servers share among themselves the threshold signature for the next expected timestamp. End-user clients coordinate with the storage elements for atomic broadcast as well as adhering to the protocol enforcing non-skipping timestamps, leading to a strongly coupled (**sC**) architecture.

3)

In [19], two approaches, called ORCAS-A and ORCAS-B, are proposed. In ORCAS-A (see Figure 6), a write client first (Step 0, not shown) tries to contact all the n storage nodes, and waits for $n-f$ responses (where f is the target level of crash or **fail-stop** failures to be tolerated) to determine the latest (logical) timestamp for the data, and increment it to obtain a new timestamp. Then (Step 1) the whole data along with the associated timestamp is sent by the writer to all the (available) storage nodes. Each storage node provisionally stores the data by replication at first, by **in-place** replacement of the the previous value (be it the whole data, or an encoded piece). It is however possible that some of the nodes are offline when the data is sent out, or otherwise, due to communication outage, do not receive this new data, and as such, they may retain an older value, which in turn could

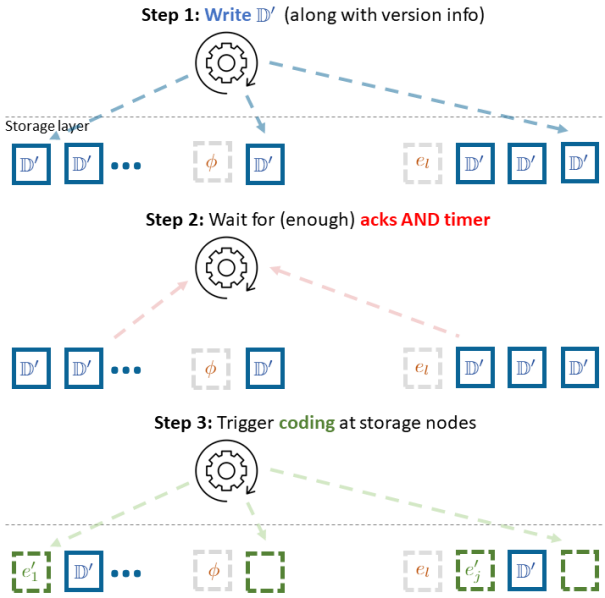


FIGURE 6. ORCAS-A from [19].

be either an encoded piece, e.g., e_l shown in the figure, or a replica of a whole object (case not shown), or if no previous value was held at the node, then it might continue to store nothing, indicated as ϕ in the figure. The writer waits (Step 2) for acknowledgements from at least $n-f$ nodes (and the expiry of a timer) to determine that such a node **quorum** has the latest value. It waits for the timer expiry, to allow all live nodes to respond when communication delays are bounded. Then, as a last phase (Step 3), the writer instructs these **active nodes** to carry out encoding locally and replace the whole replicated data with an encoded piece, using a code with parameters $(n, k = q-f)$ where f is the targeted fault-tolerance, and $q \geq n-f$ is the actual (dynamic) number of acknowledgements that the client received. The nodes need to record the associated coding parameters as meta-data, on top of the timestamp. Depending on the status of the nodes, and unreliability of communication, some nodes might miss the trigger to code, and may retain the replicated data (coding might be (re-)triggered for unencoded data during a subsequent access to the data). The coordination is centralized (at the writer client[2]) but computations are distributed (**DC**).

For a read operation, a client needs to contact the storage nodes and wait for responses from at least $n-f$ nodes, in order to identify the latest timestamp. Then it obtains sufficient fragments of the latest version to reconstruct the value. If the writer had failed before triggering the coding of the latest version at the storage nodes, then the (first) reader would in fact read one of the replicas, and would be the one which would instruct the storage nodes to replace the replicated data by the encoded pieces (i.e., complete the last phase of the write phase by proxy). ORCAS-A entails the transfer of the whole unencoded data to all the storage nodes. Furthermore, if the client crashes before triggering and completing the encoding of data at the storage nodes, then the storage nodes continue to store the replicated data instead of (smaller) encoded pieces. To address these concerns, ORCAS-B first determines the latest timestamp and the (estimate of the) number of live storage nodes to be at least q. Then, it carries out the encoding at the client itself using a (nz, x) encoding, where $z = \frac{x}{n-2f}$ and $x = LCM(q-f, n-2f)$ where LCM is the least common multiple. Then, instead of sending the whole object to all storage nodes (as in ORCAS-A), the client sends z encoded fragments to each node as part of the write operation, along with essential meta-information, such as the new timestamp and coding parameters. Upon receiving at least q acknowledgements, it triggers a trimming process, whereby each **passive** storage node discards some of the encoded fragments, in particular, retaining only $\frac{x}{q-f}$ of the original z fragments. Doing so ensures that even if f of the q nodes experience a **fail-stop** failure, enough unique fragments remain in the system, so that the original data can be reconstructed. Since the fragments of the older versions cannot be replaced in-place, instead, fragments of **multiple versions** need to all be retained, until a separate extrinsic process (assumed, but not detailed in [19]) ensures that there are no pending concurrent write processes, following which, the fragments of the older versions could be discarded. The coordination is centralized, as for ORCAS-A, however computations are also centralized (**CC**). The multi-phased mechanisms, and the end-user client's involvement in both the variants in triggering encoding and trimming of data respectively lead to a strongly coupled (**sC**) architecture.

By reducing the number of rounds of messages between a client and storage nodes, the following detrimental side effects are created:

(i) A high data transfer cost: in one variant, the whole data is transferred to every storage node, in the other, (possibly several factors) more coded pieces are.

(ii) A high storage overhead cost: in one variant, replication is used, in the other, multiple versions are maintained, with (possibly several factors) more coded pieces until update processes are complete.

(iii) This work considers the number of nodes available at the time of carrying out a specific write, and ensures a further fault-tolerance within even this set of live nodes. While that is an interesting perspective of fault-tolerance, it leads to added implementation and operational complexity arising from the dynamic choice of code parameters.

To justify the larger volume of storage (either using replication, or by storing much larger number of encoded pieces) when the data is undergoing updates, it is claimed that when/if there is no further changes to the data, the system would benefit from the redundancy being in storage efficient erasure coded manner. Furthermore, it is argued that when the system operates in a synchronous mode, the transitionary periods

[2]In case the original writer client fails to trigger the finalization phase, the next client accessing the data is delegated to carry out the task, in that sense, there is an element of distribution of coordination.

with higher storage overheads will be shorter. However, for said advocated scenario, a two-tier system, where a replicated layer is used as a 'cache' for concurrency control, disentangling it from long term storage which can be used to store the latest version erasure coded might be much simpler to realize, and also more storage efficient. [19] does not benchmark their approach against any such baseline.

4)

In [20], a set-up with a known and static set of **active** nodes is considered, where a write operation is decomposed into two phases, a preliminary phase and one for finalization. The data is associated with both a logical version number, and a binary label $\{pre, fin\}$ to indicate whether the data has been written preliminarily or finalized. Before a new write, a client queries all the storage nodes to determine (based on a **quorum** of responses) the highest value of the version corresponding to finalized writes. The new data to be written is assigned a version number by incrementing the current one, then the client creates erasure coded fragments and pre-writes these to the nodes, with the new version number and the label '*pre*'. When a quorum of acknowledgments from storage nodes is obtained to indicate that the pre-write fragments are stored, the client sends a message to all the clients to change the label to '*fin*'. If some nodes had missed out on the encoded pieces from the pre-write process, they still store the version and label information, along with a *NULL* data in lieu of the encoded piece. The label '*fin*' indicates that a quorum q of nodes still have the encoded pieces. Thus, if the size of the quorum q is chosen such that even if further f of these nodes fail, the data is still reconstructable, i.e., $q - f \geq k$, then one can guarantee that the latest data version remains available. Read operations query all the storage nodes, and then obtain a quorum of k pieces corresponding to the latest version with '*fin*' tag, to decode and reconstruct the latest value. Since coded fragments are centrally computed at a client, while coordination is a mix of centralized and distributed mechanisms, the mechanism falls into the **CD** category.

To tolerate f arbitrary **fail-stop** failures during any phase of the algorithm, the worst case scenario is that f other nodes among the n nodes failed to respond during the pre-write phase, suggesting a quorum size of $q = n - f$, thus the overall constraint $n - 2f \geq k$.

Similar to ORCAS-B [19], the mechanism of [20] does a preliminary write, ascertains that enough nodes have the latest version, and then has a finalization phase. However, because it considers the worst case scenario, there is no trimming of additional redundancy in the finalization phase. As such, even though the fault-tolerance of an (n, k) MDS erasure code is $n - k$, this approach guarantees a fault tolerance bounded by $\frac{n-k}{2}$, with the benefit of using a static code parameter. This contrasts with [19] which optimizes the storage space used, adjusted to the actual number of live nodes encountered in the first phase. This mechanism requires the storage of all the versions, which leads to a prohibitive storage cost. A garbage collection mechanism to retain only $\Delta+1$ latest versions is proposed, for a pre-determined parameter Δ, which guarantees liveness of writes if the number of concurrent write operations does not exceed Δ. Practicalities of how to choose Δ, or adapting Δ as per workload instead of using a hard-coded configuration are not explored. Depending on the degree of concurrency to be supported, the requirement of storing multiple versions to guarantee consistency may defeat the original purpose of using erasure codes, namely, to reduce storage footprint.

In [21], the ideas from [20] are extended, with as primary differentiator reconfiguration of nodes when the pool of storage nodes is dynamic. This is in essence achieved by assigning sequence numbers to each configuration, which is incremented with a change of configuration. Contiguity of sequences is achieved using quorums of intersecting nodes across adjacent configurations, which also ensures that the latest version of the data persists across such adjacent configurations. In [22], an implementation of these ideas is presented by integration with the Cassandra [42] key-value store, accompanied with certain purported storage optimizations. Given the need for the end-user clients to be involved in a multi-phased interaction with the storage nodes, to manipulate the labels and finalize writes, the architecture in these works are considered to be strongly coupled (**sC**).

C. SEPARATE META-DATA AIDED

1)

In [23], a meta-data service is used with **passive** storage nodes which decouples the storage and coordination of meta-information from the storage of the actual data. The meta-data service is a directory service, which, for each client, provides an exclusive space to write information, which in turn can be read by all the clients, i.e. the service comprises of atomic single-writer multi-reader (SWMR) registers for each client. Thus the coordination complexity is offloaded (distributed) to the clients themselves, while relying on a centralized external meta-data service.

A sequence number-client identifier pair is considered as a logical timestamp, where the sequence number is incremented for every new write. Before a write, a client scans the aforementioned directory service and reads the timestamp recorded by every client, to determine the highest timestamp, and increments the corresponding sequence number to generate a new timestamp paired with its own client identifier. This ensures a globally unique sequence of write operations[3] in a **non-blocking** manner, where multiple writers (and readers, detailed below) can concurrently operate. The data to be stored is encoded, and fragments are sent to the storage nodes. [23] thus fits the centralized computation but distributed coordination (**CD**) framework, even though the coordination relies on a centralized meta-data service. When acknowledgements from $k + t$ **quorum** of nodes out of

[3]If multiple clients try to write concurrently and choose the same new sequential number, a prior agreed tie-breaker, such as lexical ordering of the paired client identifier, could be applied, e.g., as in [18] or [23].

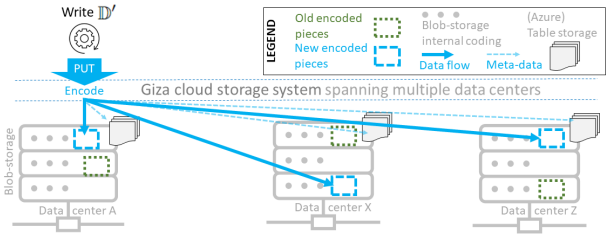


FIGURE 7. Giza [24] architecture.

$n \geq k + 2t$ storage nodes are received (where t is the number of storage node **fail-stop** failures to be tolerated), then the identity of the storage nodes, with the hashes of the fragments and the timestamp, are written into the directory service.

Each client wanting to perform a read starts by registering its intent on the meta-data directory service. Thus once a writer updates the meta-data at the directory, it also determines whether and which concurrent read operations might be ongoing. Since the reader might be reading the values prior to the write operation, corresponding fragments at other storage nodes should not be garbage collected before the reader finishes. Accordingly, the writer determines whether to garbage collect the previous value. For each writer-and-reader (concurrently active) pair, it is shown that preserving only two values can ensure atomicity of operations. Thus, while the approach does not support in-place updates, it needs to retain only a **bounded number of versions** (a maximum of two for each concurrently active writer-reader pair). The idea of explicit coordination among concurrent write and read processes is similar in purpose to [19], leading again to a strongly coupled (**sC**) architecture, even though here, it is realized via the meta-data service.

The reliable meta-data service also ensures that the latest version can be easily identified, along with the location of the storage nodes that carry the corresponding fragments. The hashes of fragments stored at the meta-data service furthermore ensure that **byzantine failures** of storage nodes can be identified. The corresponding fragments can thus be ignored during the decoding process, that is, treated as 'erasures' instead of 'errors'. The atomicity of operations in the meta-data service is exploited to coordinate all the other actions, and in particular, in guaranteeing the atomic operations over the encoded data. The issue of the meta-data service itself becoming a single point of failure or bottleneck is not discussed. Presumably, this can be addressed through replication, particularly since the volume of meta-data is significantly smaller, and there are mature techniques to ensure consistency of single-writer multi-reader replicated data.

2)

In Giza [24], similar to [23], data and meta-data are stored separately, and the meta-data plays a key role in achieving consistency. The environment considered and the mechanisms deployed are however fundamentally different. [24] is targeted at a multi-data center environment (see Figure 7), and is engineered to leverage on, but also subject to the constraints of the Azure infrastructure, specifically Azure blob and table storage.[4] For each data object, individual encoded pieces are stored across different data centers, using Azure blob storage. These coded pieces are treated as immutable data with unique identifiers. A new version of the data then leads to the introduction of new coded pieces, treated to be independent and agnostic of other pieces from other versions. Consequently, unlike the other works, there is no need nor notion of a specific predetermined storage nodes where the new encoded pieces need to reside. As such, it is assumed that all n encoded pieces are successfully populated in (some) distinct data centers and storage devices. **Passive** storage nodes can be used for this.

Version control is achieved solely through meta-data management. Meta-data about each data object is stored as a row of an Azure table within each data center, and is replicated across the data-centers over which the encoded pieces are dispersed. Paxos [43] is adapted to keep the replicated meta-data consistent across data-centers. For each version of a data object, three columns of the row of the table are used. Two of these columns carry information relevant to Paxos itself, while the third carries the information related to the new version of the data object, namely, (blob storage unique) identifiers of the encoded pieces, its corresponding data-center locations, and the committed version number.

A client carrying out a write operation first reads the meta-data from the local data center to figure out the highest committed version number, to determine (by incrementing) the tentative next version number. When it tries to commit this new version to the meta-data stored at other data centers, if it discovers that another operation had already committed said version, then, its own commit attempt would fail, and it will retry with a new version number based on the discovery of the latest version. This mechanism of updating the meta-data across different data centers is atomic (achieved through Paxos), ensuring that either the new version is committed at all the (live) data centers, or it is not, resulting in a global view of the ordering of versions. Computations of encoded pieces are centralized within the local data-center instance of Giza where the write process inserts the data using a PUT API, while coordination is distributed (in logic, storage of meta-data that enables coordination, and the Giza middleware which spans across multiple data-centers) thus making [24] fit the **CD** setting.

Reads need to go through the meta-data to determine the latest version, and access the corresponding fragments. Since the actual encoded pieces are written independently of version relationships and before updating the meta-data; furthermore since the meta-data is essentially replicated data, Giza cleverly simplifies the problem of concurrency and consistency of erasure coded data by transforming it into the issue of consistency of replicated meta-data, essentially decoupling

[4] https://docs.microsoft.com/en-us/azure/storage/common/storage-introduction

the actual versioned data from the (concurrency and consistency) control. It retains all the versions of the data, and treats a delete operation as a (special) update operation, which is tied to garbage collection of the encoded fragments of all the previous versions; while the meta-data itself is retained longer, ensuring that no new write on the same object can be carried out at any data-center, before garbage collecting the meta-data as well.

[24] explicitly uses the Giza middleware to isolate the end-user client, which only carries out a PUT operation with the data, while the rest of the orchestration is in-fact carried out in proxy, by Giza, yielding a decoupled (**dC**) architecture.

We note that the underlying Azure blob storage applies an internal erasure coding within individual data centers, for further fault-tolerance, but since this is done over the 'immutable' data, there is no need for any further version control. As such, in Giza, there are two layers of coding - a version controlled coding with $n = k + 1$ for dispersal of data across multiple data centers, and a further coding of these immutable pieces within individual data centers.

Thus, [24] assumes a **non-Byzantine asynchronous** set-up to achieve a **non-blocking** mechanism, reliant on storing **multiple versions** of the encoded data, treating these individual pieces as immutable, but aided by **active** nodes to maintain a replicated table, supporting atomic conditional updates for executing Paxos and atomic updates of information over the separately stored meta-data which is used to enforce concurrency control and consistency.

V. APPROACHES WITH ATOMIC READ-MODIFY-WRITE

Stronger than Multi-Reader Multi-Writer (MRMW), the atomic Read-Modify-Write (**aRMW**) semantics not only requires a globally agreed sequence of changes, but furthermore constrains the write operation behaviour, as if the data is read and overwritten simultaneously, with a new value or some function of the previous value. Thus, if there are concurrent write operations, say processes f and g concurrently write at t_2 to modify the value of a variable V based on the previous value V_{t_1}, and the different candidate values are $V_{t_2} = f(V_{t_1})$ and $V_{t_2'} = g(V_{t_1})$, then the resulting global ordering should be such that the second value in the sequence determines what follows, i.e., only either of $[V_{t_1}, f(V_{t_1}), g(f(V_{t_1}))]$ or $[V_{t_1}, g(V_{t_1}), f(g(V_{t_1}))]$ would be an acceptable sequence; this is in contrast to MRMW, where $[V_{t_1}, f(V_{t_1}), g(V_{t_1})]$ or $[V_{t_1}, g(V_{t_1}), f(V_{t_1})]$ could be acceptable outcomes, as described previously in Section IV.

In [25], [26], and [27], atomic Read-Modify-Write is achieved. This necessitates a **blocking algorithm**, hence the assumption of a distributed locking service, e.g., [35]. Two kinds of locks are considered: (i) exclusive write locks, such that, if a client has a write lock on a resource, no other client is allowed to have a read or a write lock on the resource until the write lock is released, (ii) read locks, which can be issued to multiple clients simultaneously, however if a resource has a read lock, then a write lock for the same cannot be granted, and vice versa, until the read lock is released.

A fundamental differences distinguish [25], [26], [27] from works reviewed earlier: they explicitly take into consideration the structural properties of the code, in particular, that the systematic pieces can be considered independent of each other, while the parities are dependent from (a subset of) systematic pieces. This naturally makes the operations at the **granularity of individual pieces** and implies that quorum membership can be explicitly determined based on the code structure, where the quorums for reads and writes are dependent on the particular systematic piece(s) involved. Smaller quorums as compared to approaches in Subsection IV-B thus suffice. Since the atomic Read-Modify-Write consistency is derived from the locking mechanism, even quorums that are agnostic of code structure could have sufficed to that end. Conversely, these quorums exploiting the code structure, deployed without locks and by storing multiple versions of data instead of carrying out in-place replacements, might achieve sequential consistency, but without the stricter atomic Read-Modify-Write guarantee.

Two mechanisms guarantee atomic Read-Modify-Write consistency, given the dependency of parity pieces on (subsets of) the systematic pieces: (i) simultaneous writes on distinct parity pieces should not render the system in an inconsistent state, where parity pieces are modified out of order; this is particularly relevant for correct degraded reads, and (ii) all live parity nodes should ideally reflect their latest value, updated correspondingly to the latest changes to relevant systematic pieces. This second clause is not strictly necessary, but desirable to make optimal use of storage overhead, and this is thus a pragmatic choice adopted in [25], [26], and [27], in contrast to other quorum based approaches in Subsection IV-B. There, not all parities are updated, instead additional redundancy is deployed up-front (e.g., many such mechanisms tolerate f failures where f is lower bounded by $\frac{n-k}{2}$, even though an MDS (n, k) code can inherently tolerate $n - k$ failures) and a decoding reliant on a majority of the nodes capturing latest value is performed, subject to an assumed bound in the number of failures in the system.

A. READ AND WRITE QUORUMS

In [25], in addition to taking into account the code structure, quorums are defined in an operation specific manner. Distinct definitions for read, degraded reads and write quorums over data encoded using an (n, k) code are proposed. Consider the systematic block d_i, for some $1 \leq i \leq k$. A write quorum $Q_w(d_i)$ to update d_i is defined to be a subset of nodes comprising one node storing d_i and at least $\lfloor \frac{n-k}{2} \rfloor + 1$ nodes storing parities. A read quorum $Q_r(d_i)$ is one node storing d_i, and a (computed) read quorum $Q_{cr}(d_i)$ is a subset of nodes comprising parities so that at least $\lfloor \frac{n-k}{2} \rfloor + 1$ parities are present and any further necessary (coded) blocks to compute d_i. Computed read quorums are useful in the context of degraded reads. A read quorum has size 1, a computed read quorum has size $\max(k, \frac{n-k}{2} + 2)$ and a write quorum has size $\frac{n-k}{2} + 2$. The value $n - k$ is typically smaller than k

for practical deployments. Quorum based works in Subsection IV-B require quorums which are strictly larger than k for both read and write operations (since full encoding/decoding is required), and often, significantly more, e.g, $\frac{n+k}{2}$.

When a write lock is acquired over a write quorum, which includes the affected systematic piece, it ensures that no other write or even read operations on the same block can be carried out simultaneously. In fact, since more than half the parity nodes are involved in a write quorum, any quorum for a future write operation is guaranteed to intersect with the set of nodes involved in previous writes, which is essential for establishing ordering of operations. This **blocking** of other operations is key to ensuring atomic Read-Modify-Write (**aRMW**) semantics. One of the problems with lock based mechanisms is that a node which has acquired certain locks may fail before releasing the locks, thus stalling the whole system. Practically, this is dealt with by releasing locks by default after a time-out [35], even if they are not released explicitly. This implies that [25] assumes a **timed asynchronous** model. Even when a systematic piece is under a write lock, other systematic piece(s) can be read simultaneously in parallel, if the corresponding storage node(s) is/are available, because the de facto read quorum comprises only the corresponding systematic pieces.

[25] adopts a proactive approach to update dissemination, where all the nodes storing parity pieces gossip among each other, to propagate the update differential. Since the differential of the updated systematic piece $\delta_x = d'_x - d_x$ itself is of the same size as a fragment, the gossip is about the meta-data, e.g., logical time-stamp, and the actual differential is pulled by the nodes which do not yet have it. The write operation concludes after ensuring that the parity nodes included in the write quorum receive the differential and associated meta-information. **Active** storage nodes locally compute the new parity values and replace them **in-place**. They cache the differential itself, pending garbage collection. Once the update is propagated to all parities (again determined through gossiping), the local caches can be garbage collected. The computation of parities is distributed, and coordination is done among the storage nodes using a gossip algorithm, which leads to a distributed computation and coordination (**DD**) based architecture.

Since a write operation acquires an exclusive lock over more than half the parity pieces, two different processes cannot simultaneously carry out writes, even over distinct systematic pieces. This is a bottleneck of [25]. A single write process can however write to multiple systematic pieces simultaneously (by acquiring write locks for the corresponding systematic pieces, but without the need to acquire any additional locks on parities, since the existing quorum suffices), and furthermore, as mentioned earlier, systematic pieces that are not being written into can in the meanwhile still be read in parallel. Orchestration of locks is assumed and delegated to a distributed lock service, resulting in a weakly coupled (**wC**) architecture. The family of works [26], [27] inherit the same system design, and focus on the code-structure specific choice of quorums, and as such, they are too are **wC**.

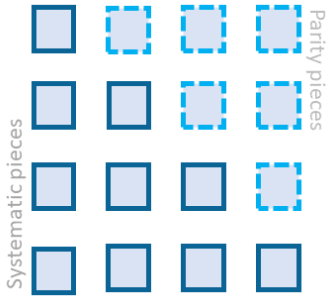


FIGURE 8. A Grid layout for an $(n, \frac{\sqrt{n}(1+\sqrt{n})}{2})$ code [26], with $\sqrt{n}$ an integer.

Byzantine behavior is not considered in [25], and the mechanism is effective for **fail-stop** set-ups, for which resource availability and contention estimations are provided for MDS codes, though the rest of the described mechanisms apply for generic codes.

B. GRID QUORUMS

The quorum technique used in [25] is asymmetric in nature, meaning that read and write quorums are different. While the resulting write quorum has a size already significantly smaller than other approaches, it nevertheless necessitates more than half of the parity nodes to be involved. In [26], two symmetric grid quorum variants (called Grid and B-Grid quorums in the context of replicated systems) are adapted to erasure coded data, to reduce the size of write quorums, but with certain constraints on the choice of code parameters. [26] demonstrates how the relationship among systematic and parity fragments can be exploited to achieve quorums involving smaller number of nodes, and affording some extent of parallelism of operations within a system of coded blocks.

As the name suggests, grid quorums assume a logical grid, meaning that while the physical layout of the storage nodes can be arbitrary, the grid layout is a logical abstraction used for algorithmic reasoning, in fact a square grid, on which n storage nodes are positioned. The grid thus has dimensions $\sqrt{n} \times \sqrt{n}$, assuming $\sqrt{n}$ is an integer. The applicability of the mechanism using Grid quorum is restricted to $(n, \frac{\sqrt{n}(1+\sqrt{n})}{2})$ codes, which also implies codes with a storage overhead marginally lower than two. For such codes, the logical grid layout shown in Figure 8 is considered, with the systematic pieces populating the lower triangle including the diagonal, while parities populate the upper triangle. With this set-up, a baseline strategy is to consider all the nodes in row i and column i to be part of a quorum, and thus for any read or write access to a systematic piece in row i, to invoke this quorum. This baseline set-up yields quorums of size $2\sqrt{n} - 1$. However, leveraging the observation from [25], these base-line quorums can be trimmed, so that for a systematic piece in row i, only a quorum involving itself, and only the parities

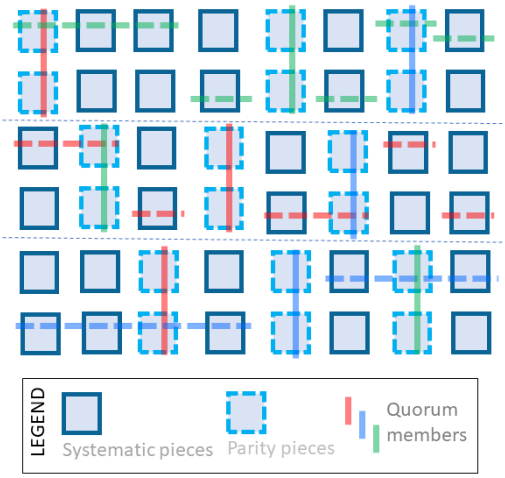


FIGURE 9. Quorums for a (48, 30) code, a B-Grid layout with $c = 8$, $b = 3$ and $r = 2$. [26].

in row i and column i can be considered. The size of these quorums are strictly upper bounded by $2\sqrt{n} - 1$.

B-Grid quorums assume a rectangular grid instead. Now $n = cbr$ nodes are arranged into a logical rectangular grid of br rows and c columns, where rows are grouped into b bands of r rows, as shown on Figure 9. The intersection of column c and band j is called a mini-column, denoted by $[\![j, c]\!]$. We first specify a set $C = \{C_1, \ldots, C_b\}$ of mini-columns in which the rb^2 parities are placed. We have b sets C_i of mini-columns, $i = 1, \ldots, b$, that is a set of mini-columns for each band in the grid, each set C_i contains b mini-columns in band i, given by $C_i = \{[\![i, c(i, \beta)]\!],\ \beta \in \{1, \ldots, b\}\}$, where $c(i, \beta)$ shows the dependency of each column in i and β, e.g. in the figure, $C_1 = \{[\![1, 1]\!], [\![2, 4]\!], [\![3, 3]\!]\}$. The $k = n - rb^2$ data blocks are placed elsewhere. Using this configuration, a quorum Q_{ij} comprises the b mini-columns $[\![1, c(1, i)]\!], \ldots [\![b, c(b, i)]\!]$, one per band, and of a choice of $c - 1$ elements in band i, such that there is exactly one element per mini-column. The index j of the quorum refers to the jth choice out of the $r(c - 1)$ choices to choose one element from a mini-column of r elements, for each of the $c - 1$ columns.

The B-Grid based quorum is available for $(n, k) = (cbr, n - rb^2)$ codes, the range of parameters is thus more flexible than the earlier Grid version. The size of a B-Grid quorum is $c - b$ data nodes plus $br + b - 1$ parity nodes, for a total of $br + c - 1$.

The focus of [26] was in designing the quorums themselves, particularly reducing the size of the quorums while ensuring intersection of any two quorums. Rest of the mechanisms, e.g., locking, propagation and computation of incremental updates, etc., are inherited from [25], and as such, they share the same general characteristics - **active** nodes, **timed-asynchronous** environment, **blocking** algorithm, using a **DD** framework and assumes **fail-stop** failures, achieving **in-place** data updates. Both the Grid and B-Grid based quorums in [26] are limited to **MDS** codes, subject to further constraints on the choice of code parameters.

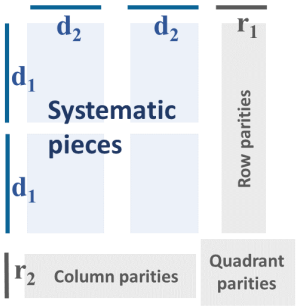


FIGURE 10. The layout of the LRC codes from [27].

C. GRID QUORUMS FOR LRC CODES

[27] exposes how an explicit treatment of the code property can be exploited to design quorums for a family of locally repairable codes (**LRC** codes). This is of particular practical relevance owing to the facts that (i) practical storage systems often and increasingly deploy LRC codes, rather than MDS codes, while (ii) a majority of approaches rely on MDS codes, and their code-structure agnostic treatment render them inapplicable (or impractical) to LRCs. This is because a quorum threshold is set, which depends on the fault tolerance of the code, which itself is independent from the set of encoded pieces for MDS codes (any k pieces are always enough), a property that is not true for LRC codes. To have a threshold that does not depend on the encoded pieces, one would need to consider a worst case scenario, making the quorum highly inefficient. Similar to [26], [27] again inherits the overall mechanisms and thus shares the characteristics of [25]. The focus of [27] is thus in designing a quorum mechanism in tandem with an LRC code design.

The specific family of LRC proposed in [27] is similar to [3] and [32], where the code locality naturally maps to the (logical) grid layout, and ideas from [26] on Grid quorum construction for erasure coded data are adapted by leveraging the structural nuances of the proposed LRC code family.

The LRC code is an $(n, k) = (4d_1d_2 + r, 4d_1d_2)$ code. Using $k = 4d_1d_2$ allows to place the systematic pieces into a grid layout formed by 4 quadrants as shown in Figure 10, each with d_1 rows and d_2 columns. The r parities are grouped as $2d_1r_1$ row parities (as the name suggests, these are computed using the systematic pieces from the same row), similarly, $2d_2r_2$ column parities, and $6r_3$ quadrant parities where r_3 parities are computed using all systematic pieces from a given combination of two quadrants. r_1, r_2 and r_3 are code design parameters, as are d_1 and d_2.

Two quorum variants are proposed: if $r_3 = 0$, the code only has row and column parities. Then the quorum $Q_{i,j}$ to access the systematic piece located on row i and column j comprises the nodes in position $\{(i, j), (i, 2d_2+l), (2d_1+k, j)\}$ for $l = \{1, \ldots, r_1\}$ and $k = \{1, \ldots, r_2\}$, i.e., the data symbol and the parities sharing its row and column in the grid layout. This quorum technically does not satisfy the formal definition of quorum since $Q_{i,j} \cap Q_{k,l} = \varnothing$ if both $i \neq k$ and $j \neq l$ hold,

even though it is adequate in this current setting to guarantee consistency. 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 for ensuring consistency and the proposed quorums are adequate to achieve sequential consistency at the granularity of individual data pieces.

Now, if $r_3 > 0$, the code has row, column as well as quadrant parities. The quorum $Q_{i,j}$ to access the systematic piece located on row i and column j includes the nodes from the quorum for $r_3 = 0$, and additionally the $3r_3$ quadrant parities that are determined according to the quadrant to which the systematic piece belongs. Consider the quorums $Q_{i,j}$ and $Q_{k,l}$ with $r_3 > 0$. 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 data object $\mathbb{D}$. 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.

VI. CONCLUDING REMARKS

A. FACTS AND ARTEFACTS

At the beginning of this paper, we provided Table 1, containing an overview of the works considered. Two groups of properties were highlighted: first, those that are playing a role in the update processes, including the nature of the storage nodes (active or passive), of the architecture (in terms of centralization or distribution of computation/coordination), whether the updates are in-place. We also listed properties related to the system model and design goals - in terms of accommodating (a)synchronicity, realization of non/blocking algorithms, and the nature of failures tolerated.

Now that the considered works are understood, we extract below interactions among some of their key characteristics across the dimensions discussed above.

1) ARCHITECTURE VS STORAGE

We notice that systems with passive servers often use centralized computations and coordination (**CC**), since they do not have the ability to perform these tasks themselves, while if the architecture assumes distributed computations and coordination (**DD**), then the storage nodes are typically **active**; though [23], [24] are two notable exceptions, which can be explained by the presence of extrinsic meta-data which is leveraged to disentangle the storage (relegated to passive nodes) from the control logic. In [23] the clients interact among each other via a collection of Single-Writer, Multi-Reader registers used as the extrinsic meta-data store, while in [24] the meta-data is stored in Azure tables, which provides sophisticated primitives like conditional atomic swap which can be deemed as active, and the control logic is realized through the Giza middlware which is distributed and runs across multiple data-centers.

2) ATOMIC READ-MODIFY-WRITE (aRMW) VS BLOCKING AND IN-PLACE

aRMW operations need to ensure that two concurrent write operations reading and using the same previous value of the data should not happen since any ordering of these operations will not satisfy the atomicity property; hence they necessarily rely on blocking. In contrast, non-blocking algorithms generate multiple and divergent versions optimistically, on which a global ordering may be established a posteriori. Thus, they may require the retention of multiple versions (particularly when coordination is distributed), at least until the ordering is established, after which, older versions may be discarded (garbage collected). The approaches reliant on multiple versions whose ordering is resolved subsequently are naturally incongruent to in-place updates.

3) DESIGN CONSTRAINT UNDER ASYNCHRONICITY

In an asynchronous set-up, a mechanism contingent on responses from a specific entity is undesirable, because of its adverse impact on liveness. As such, we notice that (almost) all the mechanisms that operate under an assumption of an asynchronous environment rely on quorum based algorithms, where a threshold of responses, but irrespective of, which particular individual entities these responses are from, is used. This in turn limits their applicability to MDS codes, an issue and its implications have been elaborated in Section IV-B. An apparent exception is [24]. In [24], new write operations are however not tied to any specific individual node, or even specific data-center, and as such, the mechanism can wait for 'all' responses and still progress in an asynchronous environment, since the responders in each iteration can be any out of a much larger set of entities. In contrast, [25], [26], [27] explore code structure specific quorums, which (in part) requires responses from some specific nodes. These works relied on a timed-synchronous assumption.

B. BYZANTINE FAULTS

This brings us to an issue which is peripheral to our core focus. Within the storage layer, byzantine faults typically happen because of two primary reasons: software artifacts or bugs [44] and corruption of the actual data [45] stored in the storage hardware. e.g., [44] identified that approximately 5% of the bugs affect consistency. Such bugs however would result in an unbounded number of byzantine faults, while all the existing algorithmic solutions that we discuss are efficacious under the assumption of a limited number of byzantine faults. As such, the root cause of such software and firmware byzantine faults need to be addressed, which require fundamentally different tools, e.g., formal verification [46], which is outside our scope. In a study of data corruption in the storage stack [45], silent data corruptions are characterized

as checksum/parity inconsistencies, as well as (file-system block) identity discrepancies. File-systems address the block identity discrepancy issues. Likewise, local RAID system of modern storage systems are equipped to deal with silent data corruption such as checksum and parity inconsistencies. Often, erasure codes furthermore have inherent error correcting capability. As such, the algorithmic approaches of byzantine fault-tolerance surveyed here may have limited practicality, both because there are off-the-shelf solutions to deal with random errors, while systematic errors (e.g., arising from software bugs) may lead to arbitrary number of faults, which the algorithmic solutions cannot cope with. As such, the scarcity of works dealing with byzantine nodes may be explained both by the significantly more complex system designs they entail, as well as the marginal practical benefit such a design might provide in the specific context.

C. FROM REPLICATION TO ERASURE CODING

Given the antecedence of the literature on concurrency control and consistency for replicated data, which inspired many of the reviewed works, our final remarks are on the subtleties of the transition from replicated to erasure coded redundancy.

When using replication, as long as one full copy of a version, old or new, is present in the system, that version is usable, and it can be further propagated to make more copies. This is particularly so if no data corruption errors (byzantine faults at storage) happen. A quorum is then used to ensure that (at least one instance of) the latest version of the data is identified. Thus, in replicated quorum system designs, older data may be replaced in-place at the storage nodes. In contrast, for erasure coded redundancy, no single storage node carries enough information to reconstruct the whole data object $\mathbb{D}$, and the code parameters determine another threshold, which needs to be met, so that the storage nodes among them carry enough information to recompute the latest version of overall $\mathbb{D}$. If data at individual nodes were to be replaced in-place and older instances were discarded immediately before ascertaining that sufficient number of other nodes have also stored other encoded pieces, then there would be a risk of insufficient information for reconstructing the latest data while also losing the older version(s). This trade-off is explicitly exposed and juxtaposed in the two variants of [19], where, ORCAS-A achieves in-place updates with replicas, before triggering coding at each storage node; while in ORCAS-B, where coded data is populated directly at storage nodes, multiple versions need to be maintained.

Consequently, the works in IV-B predominantly need to rely on storing multiple versions. Exceptions like [17] and [18] use extrinsic mechanisms to ensure that enough encoded pieces are written together, achieving in-place data writes by relegating the overhead of storing the multiple versions from the storage system to 'elsewhere'. E.g., [17] stores new data in a temporary buffer until it verifies that enough storage nodes have pieces from the same write operations, before committing the writes to the storage layer, while [18] relies on an atomic broadcast primitive, which itself again would need to buffer the data until delivery of the data. Likewise, the works from Section V, by operating at the granularity of individual pieces, achieve in-place replacement, but they store the differentials with previous versions temporarily, and garbage collect these after ascertaining that the update has propagated across the rest of the storage nodes which were not part of the original quorum used to carry out the write.

D. OUTLOOK

The works [25], [26], [27] adapt techniques for replication, namely Gifford's quorum in [25] and grid quorums in [26] to erasure codes, by explicitly accounting for the structural relationship among the codewords, allowing for significantly smaller quorum sizes as compared to the ones used in many of the works in IV-B. Three promising directions of future investigation are: (i) explore and adapt other quorums that have been studied previously for replicated data, and more crucially (ii) explore the design of quorums applicable to more families of LRCs, and (iii) determine whether and how such quorums can be applied in an asynchronous instead of a timed-asynchronous environment and in realizing non-blocking mechanisms.

In a spirit similar as above, the structure of error correcting codes may be exploited to mitigate byzantine faults. Yet, none of the reviewed works do so. This comprises another natural frontier to explore.

Finally, the current body of work has very heterogeneous treatments, often impeding their valorization for real-world deployments. Real-world usage and setups data informed workloads, along with a proper set of metrics of pragmatic interest to establish a common and comprehensive set of benchmarks that can be used to evaluate and compare various approaches are thus necessary for translating research in this space into wide-scale adoption. This work, while qualitative in nature, serves as a first step in that direction. We have systematized the knowledge, not only in terms of the design elements of the algorithms, but also by identifying the spectrum of models and assumptions, which may serve as a guide to identify and establish quantitative metrics.

ACKNOWLEDGMENT

The authors would like thank Dr. V. R. Cadambe for his patient explanations clarifying the author's queries on some of his coauthored papers [20], [21], [22].

REFERENCES

[1] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan, "Does erasure coding have a role to play in my data center?" Microsoft Res., Redmond, WA, USA, Tech. Rep. MSR-TR-2010-52, 2010.

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

[3] 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.

[4] 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. USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2014, pp. 383–398.

[5] 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 *Proc. Symp. Mass Storage Syst. Technol. (MSST)*, 2015, pp. 1–14.
[6] B. Charron-Bost, F. Pedone, and A. Schiper, *Replication* (Lecture Notes in Computer Science), vol. 5959. Berlin, Germany: Springer, 2010.
[7] O. T. Lee, S. D. M. Kumar, and P. Chandran, "Erasure coded storage systems for cloud storage—Challenges and opportunities," in *Proc. Int. Conf. Data Sci. Eng. (ICDSE)*, 2016, pp. 1–7.
[8] W. Lin, D. Chiu, and Y. Lee, "Erasure code replication revisited," in *Proc. Int. Conf. Peer-to-Peer Comput.*, 2004, pp. 90–97.
[9] F. Oggier and A. Datta, "Coding techniques for repairability in networked distributed storage systems," *Found. Trends Commun. Inf. Theory*, vol. 9, no. 4, pp. 383–466, 2013.
[10] S. Liu and F. Oggier, "An overview of coding for distributed storage systems," in *Network Coding and Subspace Designs*. Berlin, Germany: Springer, 2018.
[11] Y. Perry. (2020). *What is Block Storage?* [Online]. Available: https://cloud.netapp.com/blog/cvo-blg-what-is-block-storage-pros-cons-and-comparisons
[12] RedHat. (2018). *File Storage, Block Storage, or Object Storage*. [Online]. Available: https://www.redhat.com/en/topics/data-storage/file-block-object-storage
[13] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and secure storage in a cloud-of-clouds," *ACM Trans. Storage*, vol. 9, no. 4, pp. 1–33, 2013.
[14] 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.
[15] K. Peter and A. Reinefeld, "Consistency and fault tolerance for erasure-coded distributed storage systems," in *Proc. 5th Int. Workshop Data-Intensive Distrib. Comput. Date*, 2012, pp. 23–32.
[16] 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.*, 2004, pp. 135–144.
[17] J. Hendricks, G. R. Ganger, and M. K. Reiter, "Low-overhead Byzantine fault-tolerant storage," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 73–86, Oct. 2007.
[18] C. Cachin and S. Tessaro, "Optimal resilience for erasure-coded Byzantine distributed storage," in *Proc. Int. Conf. Dependable Syst. Netw. (DSN)*, 2006, pp. 115–124.
[19] P. Dutta, R. Guerraoui, and R. R. Levy, "Optimistic erasure-coded distributed storage," in *Proc. Int. Symp. Distrib. Comput.* Berlin, Germany: Springer, 2008, pp. 182–196.
[20] 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.
[21] 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.
[22] V. R. Cadambe, K. M. Konwar, M. Medard, H. Pan, L. Tseng, and Y. Wu, "CassandrEAS: Highly available and storage-efficient distributed key-value store with erasure coding," in *Proc. IEEE 19th Int. Symp. Netw. Comput. Appl. (NCA)*, Nov. 2020, pp. 1–8.
[23] E. Androulaki, C. Cachin, D. Dobre, and M. Vukolić, "Erasure-coded Byzantine storage with separate metadata," in *Proc. Int. Conf. Princ. Distrib. Syst.*, 2014, pp. 76–90.
[24] 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. USENIX Annu. Tech. Conf. (ATC)*, 2017, pp. 539–551.
[25] A. Datta and F. Oggier, "Quorums over codes," *J. Parallel Distrib. Comput.*, vol. 161, pp. 1–19, Mar. 2022.
[26] F. Oggier and A. Datta, "On grid quorums for erasure coded data," *Entropy*, vol. 23, no. 2, p. 177, Jan. 2021.
[27] A. Datta, A. A. Fahreza, and F. Oggier, "QLOC: Quorums with local reconstruction codes," *IEEE Access*, vol. 9, pp. 93298–93314, 2021.
[28] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proc. VLDB Endowment*, vol. 7, no. 3, pp. 181–192, Nov. 2013.
[29] P. Viotti and M. Vukolić, "Consistency in non-transactional distributed storage systems," *ACM Comput. Surv.*, vol. 49, no. 1, pp. 1–34, 2016.
[30] L. Lamport, "On interprocess communication," Microsoft Res., Tech. Rep., 1985. [Online]. Available: https://lamport.azurewebsites.net/pubs/interprocess.pdf
[31] K. S. Esmaili, A. Chiniah, and A. Datta, "Efficient updates in cross-object erasure-coded storage systems," in *Proc. Int. Conf. Big Data*, 2013, pp. 28–32.
[32] 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.
[33] S. Poledna, *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*, vol. 345. Berlin, Germany: Springer, 2007, ch. 3.
[34] F. Cristian and C. Fetzer, "The timed asynchronous distributed system model," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, pp. 642–657, Jun. 1999.
[35] Redis Labs. *Distributed Locks With Redis*. Accessed: Feb. 11, 2021. [Online]. Available: https://redis.io/topics/distlock
[36] J. Harshan, F. Oggier, and A. Datta, "Sparsity exploiting erasure coding for resilient storage and efficient I/O access in delta based versioning systems," in *Proc. IEEE 35th Int. Conf. Distrib. Comput. Syst.*, Jun. 2015, pp. 798–799.
[37] J. Harshan, A. Datta, and F. Oggier, "DiVers: An erasure code based storage architecture for versioning exploiting sparsity," *Future Gener. Comput. Syst.*, vol. 59, pp. 47–62, Jun. 2016.
[38] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *Commun. ACM*, vol. 56, no. 5, pp. 55–63, May 2013.
[39] 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, 2007.
[40] J.-P. Martin, L. Alvisi, and M. Dahlin, "Minimal Byzantine storage," in *Proc. Int. Symp. Distrib. Comput.* Berlin, Germany: Springer, 2002, pp. 311–325.
[41] G. Bracha, "An asynchronous [(n-1)/3]-resilient consensus protocol," in *Proc. 3rd Annu. ACM Symp. Princ. Distrib. Comput.*, 1984, pp. 154–162.
[42] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.
[43] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 51–58, Dec. 2001.
[44] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-Anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What bugs live in the cloud? A study of 3000+ issues in cloud systems," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2014, pp. 1–14.
[45] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, "An analysis of data corruption in the storage stack," *ACM Trans. Storage*, vol. 4, no. 3, pp. 1–28, Nov. 2008.
[46] P. Deligiannis, M. McCutchen, P. Thomson, S. Chen, A. F. Donaldson, J. Erickson, C. Huang, A. Lal, R. Mudduluru, S. Qadeer, and W. Schulte, "Uncovering bugs in distributed storage systems during testing (not in production!)," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 249–262.

ANWITAMAN DATTA is currently an Associate Professor with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. His core research interests include the topics of large-scale resilient distributed systems, information security, and applications of data analytics. Presently, he is exploring topics at the intersection of computer science, 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.

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