# ARES: Adaptive, Reconfigurable, Erasure coded, atomic Storage

NICOLAS NICOLAOU, Algolysis Ltd, Limassol, Cyprus
VIVECK CADAMBE, EE Department, Penn. State University, University Park, PA, US
N. PRAKASH, KISHORI M. KONWAR, MURIEL MEDARD, and NANCY LYNCH,
Massachusetts Institute of Technology, Cambridge MA, USA

Emulating a shared *atomic*, read/write storage system is a fundamental problem in distributed computing. Replicating atomic objects among a set of data hosts was the norm for traditional implementations (e.g., [6]) in order to guarantee the availability and accessibility of the data despite host failures. As replication is highly storage demanding, recent approaches suggested the use of erasure-codes to offer the same fault-tolerance while optimizing storage usage at the hosts. Initial works focused on a fix set of data hosts. To guarantee longevity and scalability, a storage service should be able to dynamically mask hosts failures by allowing new hosts to join, and failed host to be removed without service interruptions. This work presents the first erasure-code based atomic algorithm, called ARES, which allows the set of hosts to be modified in the course of an execution. ARES is composed of three main components: (i) a *reconfiguration protocol*, (ii) a *read/write protocol*, and (iii) a set of *data access primitives*. The design of ARES is modular and is such to accommodate the usage of various erasure-code parameters on a per-configuration basis. We provide bounds on the latency of read/write operations, and analyze the storage and communication costs of the ARES algorithm.

**ACM Reference Format:**
Nicolas Nicolaou, Viveck Cadambe, N. Prakash, Kishori M. Konwar, Muriel Medard, and Nancy Lynch. 2020. ARES: Adaptive, Reconfigurable, Erasure coded, atomic Storage . 1, 1 (July 2020), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Distributed Storage Systems (DSS) store large amounts of data in an affordable manner. Cloud vendors deploy hundreds to thousands of commodity machines, networked together to act as a single giant storage system. Component failures of commodity devices, and network delays are the norm, therefore, ensuring consistent data-access and availability at the same time is challenging. Vendors often solve availability by replicating data across multiple servers. These services use carefully constructed algorithms that ensure that these copies are consistent, especially when they can be accessed concurrently by different operations. The problem of keeping copies consistent becomes even more challenging when failed servers need to be replaced or new servers are added, without

This work was partially funded by the Center for Science of Information NSF Award CCF-0939370, NSF Award CCF-1461559, AFOSR Contract Number: FA9550-14-1-0403, NSF CCF-1553248 and RPF/POST-DOC/0916/0090.
Authors' addresses: Nicolas Nicolaou, nicolas@algolysis.comAlgolysis Ltd, Limassol, Cyprus; Viveck Cadambe, vxc12@engr.psu.eduEE Department, Penn. State University, University Park, PA, US; N. Prakash, prakashn@mit.edu; Kishori M. Konwar, kishori@csail.mit.edu; Muriel Medard, medard@mit.edu; Nancy Lynch, lynch@csail.mit.eduMassachusetts Institute of Technology, Cambridge MA, USA.


XXXX-XXXX/2020/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

interrupting the service. Any type of service interruption in a heavily used DSS usually translates to immense revenue loss.

The goal of this work is to provide an algorithm for implementing strongly consistent (i.e., atomic/linearizable), fault-tolerant distributed read/write storage, with low storage and communication footprint, and the ability to reconfigure the set of data hosts without service interruptions.

**Replication-based Atomic Storage.** A long stream of work used replication of data across multiple servers to implement atomic (linearizable) read/write objects in message-passing, asynchronous environments where servers (data hosts) may crash fail [5, 6, 15–17, 19, 20, 32]. A notable replication-based algorithm appears in the work by Attiya, Bar-Noy and Dolev [6] (we refer to as the ABD algorithm) which implemented non-blocking atomic read/write data storage via logical timestamps paired with values to order read/write operations. Replication based strategies, however, incur high storage and communication costs; for example, to store 1,000,000 objects each of size 1MB (a total size of 1TB) across a 3 server system, the ABD algorithm replicates the objects in all the 3 servers, which blows up the worst-case *storage cost* to 3TB. Additionally, every write or read operation may need to transmit up to 3MB of data (while retrieving an object value of size 1MB), incurring high *communication cost*.

**Erasure Code-based Atomic Storage.** Erasure Coded-based DSS are extremely beneficial to save storage and communication costs while maintaining similar fault-tolerance levels as in replication based DSS [11]. Mechanisms using an $[n, k]$ erasure code splits a value $v$ of size, say 1 unit, into $k$ elements, each of size $\frac{1}{k}$ units, creates $n$ *coded elements* of the same size, and stores one coded element per server, for a total storage cost of $\frac{n}{k}$ units. So the $[n = 3, k = 2]$ code in the previous example will reduce the storage cost to 1.5TB and the communication cost to 1.5MB (improving also operation latency). Maximum Distance Separable (MDS) codes have the property that value $v$ can be reconstructed from any $k$ out of these $n$ coded elements; note that replication is a special case of MDS codes with $k = 1$. In addition to the potential cost-savings, the suitability of erasure-codes for DSS is amplified with the emergence of highly optimized erasure coding libraries, that reduce encoding/decoding overheads [1, 7, 36]. In fact, an exciting recent body of systems and optimization works [2, 26, 36, 39–42, 44] have demonstrated that for several data stores, the use of erasure coding results in lower latencies than replication based approaches. This is achieved by allowing the system to carefully tune erasure coding parameters, data placement strategies, and other system parameters that improve workload characteristics – such as load and spatial distribution. A complementary body of work has proposed novel non-blocking algorithms that use erasure coding to provide an atomic storage over asynchronous message passing models [8, 10, 11, 14, 27, 28, 43]. Since erasure code-based algorithms, unlike their replication-based counter parts, incur the additional burden of synchronizing the access of multiple pieces of coded-elements from the *same version* of the data object, these algorithms are quite complex.

**Reconfigurable Atomic Storage.** *Configuration* refers to the set of storage servers that are collectively used to host the data and implement the DSS. *Reconfiguration* is the process of adding or removing servers in a DSS. In practice, reconfigurations are often desirable by system administrators [4], for a wide range of purposes, especially during system maintenance. As the set of storage servers becomes older and unreliable they are replaced with new ones to ensure data-durability. Furthermore, to scale the storage service to increased or decreased load, larger (or smaller) configurations may be needed to be deployed. Therefore, in order to carry out such reconfiguration steps, in addition to the usual read and write operations, an operation called reconfig is invoked by reconfiguration clients. Performing reconfiguration of a system, without service interruption, is a very challenging task and an active area of research. RAMBO [31] and DynaStore [3] are two of the

handful of algorithms [12, 18, 21, 25, 37, 38] that allows reconfiguration on live systems; all these algorithms are replication-based.

Despite the attractive prospects of creating strongly consistent DSS with low storage and communication costs, so far, no algorithmic framework for reconfigurable atomic DSS employed erasure coding for fault-tolerance, or provided any analysis of bandwidth and storage costs. Our paper fills this vital gap in algorithms literature, through the development of novel reconfigurable approach for atomic storage that use *erasure codes* for fault-tolerance. From a practical viewpoint, our work may be interpreted as a bridge between the systems optimization works [2, 26, 36, 39–42, 44] and non-blocking erasure coded based consistent storage [8, 10, 11, 14, 27, 28, 43]. Specifically, the use of our *reconfigurable* algorithms would potentially enable a data storage service to dynamically shift between different erasure coding based parameters and placement strategies, as the demand characteristics (such as load and spatial distribution) change, without service interruption.

**Our Contributions.** We develop a *reconfigurable*, *erasure-coded*, *atomic* or *strongly consistent* [23, 30] read/write storage algorithm, called ARES. Motivated by many practical systems, ARES assumes clients and servers are separate processes * that communicate via logical point-to-point channels.

In contrast to the, replication-based reconfigurable algorithms [3, 12, 18, 21, 25, 31, 37, 38], where a configuration essentially corresponds to the set of servers that stores the data, the same concept for erasure coding need to be much more involved. In particular, in erasure coding, even if the same set of $n$ servers are used, a change in the value of $k$ defines a new configuration. Furthermore, several erasure coding based algorithms [10, 14] have additional parameters that tune how many older versions each server store, which in turn influences the concurrency level allowed. Tuning of such parameters can also fall under the purview of reconfiguration.

To accommodate these various reconfiguration requirements, ARES takes a modular approach. In particular, ARES uses a set of primitives, called *data-access primitives* (DAPs). A different implementation of the DAP primitives may be specified in each configuration. ARES uses DAPs as a "black box" to: (i) transfer the object state from one configuration to the next during reconfig operations, and (ii) invoke read/write operations on a single configuration. Given the DAP implementation for each configuration we show that ARES correctly implements a *reconfigurable*, *atomic* read/write storage.

| Algorithm | #rounds /write | #rounds /read | Reconfig. | Repl. or EC | Storage cost | read bandwidth | write bandwidth |
|---|---|---|---|---|---|---|---|
| CASGC [9] | 3 | 2 | No | EC | $(\delta+1)\frac{n}{k}$ | $\frac{n}{k}$ | $\frac{n}{k}$ |
| SODA [27] | 2 | 2 | No | EC | $\frac{n}{k}$ | $(\delta+1)\frac{n}{k}$ | $\frac{n^2}{k}$ |
| ORCAS-A [14] | 3 | $\geq 2$ | No | EC | $n$ | $n$ | $n$ |
| ORCAS-B [14] | 3 | 3 | No | EC | $\infty$ | $\infty$ | $\infty$ |
| ABD [6] | 2 | 2 | No | Repl. | $n$ | $2n$ | $n$ |
| RAMBO [31] | 2 | 2 | Yes | Repl. | $\geq n$ | $\geq n$ | $\geq n$ |
| DYNASTORE [3] | $\geq 4$ | $\geq 4$ | Yes | Repl. | $\geq n$ | $\geq n$ | $\geq n$ |
| SMARTMERGE [25] | 2 | 2 | Yes | Repl. | $\geq n$ | $\geq n$ | $\geq n$ |
| ARES (this paper) | 2 | 2 | Yes | EC | $(\delta+1)\frac{n}{k}$ | $(\delta+1)\frac{n}{k}$ | $\frac{n}{k}$ |

Table 1. Comparison of ARES with previous algorithms emulating atomic Read/Write Memory for replication (Repl.) and erasure-code based (EC) algorithms. $\delta$ is the maximum number of concurrent writes with any read during the course of an execution of the algorithm. In practice, $\delta < 4$ [11].

The DAP primitives provide ARES a much broader view of the notion of a configuration as compared to replication-based algorithms. Specifically, the DAP primitives may be parameterized, following the parameters of protocols used for their implementation (e.g., erasure coding parameters,

*In practice, these processes can be on the same node or different nodes.

set of servers, quorum design, concurrency level, etc.). While transitioning from one configuration to another, our modular construction, allows ARES to reconfigure between different sets of servers, quorum configurations, and erasure coding parameters. In principle, ARES even allows to reconfigure between completely different protocols as long as they can be interpreted/expressed in terms of the primitives; though in this paper, we only present one implementation of the DAP primitives to keep the scope of the paper reasonable. From a technical point of view, our modular structure makes the atomicity proof of a complex algorithm (like ARES) easier.

An important consideration in the design choice of ARES, is to ensure that we gain/retain the advantages that come with erasure codes – cost of data storage and communication is low – while having the flexibility to reconfigure the system. Towards this end, we present an erasure-coded implementation of DAPs which satisfy the necessary properties, and are used by ARES to yield the first reconfigurable, *erasure-coded*, read/write atomic storage implementation, where read and write operations complete in *two-rounds*. We provide the atomicity property and latency analysis for any operation in ARES, along with the storage and communication costs resulting from the erasure-coded DAP implementation. In particular, we specify lower and upper bounds on the communication latency between the service participants, and we provide the necessary conditions to guarantee the termination of each read/write operation while concurrent with reconfig operations.

Table 1 compares ARES with a few well-known erasure-coded and replication-based (static and reconfigurable) atomic memory algorithms. From the table we observe that ARES is the only algorithm to combine a dynamic behavior with the use of erasure codes, while reducing the storage and communcation costs associated with the read or write operations. Moreover, in ARES the number of rounds per write and read is at least as good as in any of the remaining algorithms.

**Document Structure.** Section 2, presents the model assumptions and Section 3, the DAP primitives. In Section 4, we present the implementation of the reconfiguration and read/write protocols in ARES using the DAPs. In Section 5, we present an erasure-coded implementation of a set of DAPs, which can be used in every configuration of the ARES algorithm. Section **??** provides operation latency and cost analysis, and Section 8 the DAP flexibility. We conclude our work in Section 9. Due to lack of space omitted proofs can be found in [34].

## 2 MODEL AND DEFINITIONS

A shared atomic storage, consisting of any number of individual objects, can be emulated by composing individual atomic memory objects. Therefore, herein we aim in implementing a single atomic *read/write* memory object. A read/write object takes a value from a set $\mathcal{V}$. We assume a system consisting of four distinct sets of processes: a set $\mathcal{W}$ of writers, a set $\mathcal{R}$ of readers, a set $\mathcal{G}$ of reconfiguration clients, and a set $\mathcal{S}$ of servers. Let $\mathcal{I} = \mathcal{W} \cup \mathcal{R} \cup \mathcal{G}$ be the set of clients. Servers host data elements (replicas or encoded data fragments). Each writer is allowed to modify the value of a shared object, and each reader is allowed to obtain the value of that object. Reconfiguration clients attempt to introduce new configuration of servers to the system in order to mask transient errors and to ensure the longevity of the service. Processes communicate via *messages* through *asynchronous*, and *reliable* channels.

**Configurations.** A *configuration*, with a unique identifier from a set $C$, is a data type that describes the finite set of servers that are used to implement the atomic storage service. In our setting, each configuration is also used to describe the way the servers are grouped into intersecting sets, called *quorums*, the consensus instance that is used as an external service to determine the next configuration, and a set of data access primitives that specify the interaction of the clients and servers in the configuration (see Section 3). More formally, a configuration, $c \in C$, consists of: $(i)$ $c.Servers \subseteq \mathcal{S}$: a set of server identifiers; $(ii)$ $c.Quorums$: the set of quorums on $c.Servers$, s.t.

$\forall Q_1, Q_2 \in c.Quorums, Q_1, Q_2 \subseteq c.Servers$ and $Q_1 \cap Q_2 \neq \emptyset$; $(iii)$ $DAP(c)$: the set of primitives (operations at level lower than reads or writes) that clients in $\mathcal{I}$ may invoke on $c.Servers$; and $(iv)$ $c.Con$: a consensus instance with the values from $C$, implemented and running on top of the servers in $c.Servers$. We refer to a server $s \in c.Servers$ as a *member* of configuration $c$. The consensus instance $c.Con$ in each configuration $c$ is used as a service that returns the identifier of the configuration that follows $c$.

**Executions.** An algorithm $A$ is a collection of processes, where process $A_p$ is assigned to process $p \in \mathcal{I} \cup \mathcal{S}$. The *state*, of a process $A_p$ is determined over a set of state variables, and the state $\sigma$ of $A$ is a vector that contains the state of each process. Each process $A_p$ implements a set of actions. When an action $\alpha$ occurs it causes the state of $A_p$ to change, say from some state $\sigma_p$ to some different state $\sigma'_p$. We call the triple $\langle \sigma_p, \alpha, \sigma'_p \rangle$ a *step* of $A_p$. Algorithm $A$ performs a step, when some process $A_p$ performs a step. An action $\alpha$ is *enabled* in a state $\sigma$ if $\exists$ a step $\langle \sigma, \alpha, \sigma' \rangle$ to some state $\sigma'$. An *execution* is an alternating sequence of states and actions of $A$ starting with the initial state and ending in a state. An execution $\xi$ *fair* if enabled actions perform a step infinitely often. In the rest of the paper we consider executions that are fair and well-formed. A process $p$ *crashes* in an execution if it stops taking steps; otherwise $p$ is *correct* or *non-faulty*. We assume a function $c.\mathcal{F}$ to describe the failure model of a configuration $c$.

**Reconfigurable Atomic Read/Write Objects.** A reconfigurable atomic object supports three operations: read(), write($v$) and reconfig($c$). A read() operation returns the value of the atomic object, write($v$) attempts to modify the value of the object to $v \in \mathcal{V}$, and the reconfig($c$) that attempts to install a new configuration $c \in C$. We assume *well-formed* executions where each client may invoke one operation (read(), write($v$) or reconfig($c$)) at a time.

An implementation of a read/write or a reconfig operation contains an *invocation* action (such as a call to a procedure) and a *response* action (such as a return from the procedure). An operation $\pi$ is *complete* in an execution, if it contains both the invocation and the *matching* response actions for $\pi$; otherwise $\pi$ is *incomplete*. We say that an operation $\pi$ *precedes* an operation $\pi'$ in an execution $\xi$, denoted by $\pi \rightarrow \pi'$, if the response step of $\pi$ appears before the invocation step of $\pi'$ in $\xi$. Two operations are *concurrent* if neither precedes the other. An implementation $A$ of a read/write object satisfies the atomicity (linearizability [23]) property if the following holds [30]. Let the set $\Pi$ contain all complete read/write operations in any well-formed execution of $A$. Then there exists an irreflexive partial ordering $\prec$ satisfying the following:

- **A1.** For any operations $\pi_1$ and $\pi_2$ in $\Pi$, if $\pi_1 \rightarrow \pi_2$, then it cannot be the case that $\pi_2 \prec \pi_1$.
- **A2.** If $\pi \in \Pi$ is a write operation and $\pi' \in \Pi$ is any read/write operation, then either $\pi \prec \pi'$ or $\pi' \prec \pi$.
- **A3.** The value returned by a read operation is the value written by the last preceding write operation according to $\prec$ (or the initial value if there is no such write).

**Storage and Communication Costs.** We are interested in the *complexity* of each read and write operation. The complexity of each operation $\pi$ invoked by a process $p$, is measured with respect to three metrics, during the interval between the invocation and the response of $\pi$: $(i)$ *number of communication round*, accounting the number of messages exchanged during $\pi$, $(ii)$ *storage efficiency* (storage cost), accounting the maximum storage requirements for any single object at the servers during $\pi$, and $(iii)$ *message bit complexity* (communication cost) which measures the size of the messages used during $\pi$.

We define the total storage cost as the size of the data stored across all servers, at any point during the execution of the algorithm. The communication cost associated with a read or write operation is the size of the total data that gets transmitted in the messages sent as part of the operation. We assume that metadata, such as version number, process ID, etc. used by various operations is of

negligible size, and is hence ignored in the calculation of storage and communication cost. Further, we normalize both costs with respect to the size of the value $v$; in other words, we compute the costs under the assumption that $v$ has size 1 unit.

**Erasure Codes.** We use an $[n, k]$ linear MDS code [24] over a finite field $\mathbb{F}_q$ to encode and store the value $v$ among the $n$ servers. An $[n, k]$ MDS code has the property that any $k$ out of the $n$ coded elements can be used to recover (decode) the value $v$. For encoding, $v$ is divided into $k$ elements $v_1, v_2, \ldots v_k$ with each element having size $\frac{1}{k}$ (assuming size of $v$ is 1). The encoder takes the $k$ elements as input and produces $n$ coded elements $e_1, e_2, \ldots, e_n$ as output, i.e., $[e_1, \ldots, e_n] = \Phi([v_1, \ldots, v_k])$, where $\Phi$ denotes the encoder. For ease of notation, we simply write $\Phi(v)$ to mean $[e_1, \ldots, e_n]$. The vector $[e_1, \ldots, e_n]$ is referred to as the codeword corresponding to the value $v$. Each coded element $c_i$ also has size $\frac{1}{k}$. In our scheme we store one coded element per server. We use $\Phi_i$ to denote the projection of $\Phi$ on to the $i^{\text{th}}$ output component, i.e., $e_i = \Phi_i(v)$. Without loss of generality, we associate the coded element $e_i$ with server $i$, $1 \leq i \leq n$.

**Tags.** We use logical tags to order operations. A tag $\tau$ is defined as a pair $(z, w)$, where $z \in \mathbb{N}$ and $w \in \mathcal{W}$, an ID of a writer. Let $\mathcal{T}$ be the set of all tags. Notice that tags could be defined in any totally ordered domain and given that this domain is countably infinite, then there can be a direct mapping to the domain we assume. For any $\tau_1, \tau_2 \in \mathcal{T}$ we define $\tau_2 > \tau_1$ if $(i)$ $\tau_2.z > \tau_1.z$ or $(ii)$ $\tau_2.z = \tau_1.z$ and $\tau_2.w > \tau_1.w$.

## 3 DATA ACCESS PRIMITIVES

In this section we introduce a set of primitives, we refer to as *data access primitives (DAP)*, which are invoked by the clients during read/write/reconfig operations and are defined for any configuration $c$ in ARES. The DAPs allow us: $(i)$ to describe ARES in a *modular* manner, and $(ii)$ a cleaner reasoning about the correctness of ARES.

We define three data access primitives for each $c \in \mathcal{C}$: $c$.put-data($\langle \tau, v \rangle$), via which a client can ingest the tag value pair $\langle \tau, v \rangle$ in to the configuration $c$; $(ii)$ $c$.get-data(), used to retrieve the most up to date tag and vlaue pair stored in the configuration $c$; and $(iii)$ $c$.get-tag(), used to retrieve the most up to date tag for an object stored in a configuration $c$. More formally, assuming a tag $\tau$ from a set of totally ordered tags $\mathcal{T}$, a value $v$ from a domain $\mathcal{V}$, and a configuration $c$ from a set of identifiers $\mathcal{C}$, the three primitives are defined as follows:

DEFINITION 1 (DATA ACCESS PRIMITIVES). *Given a configuration identifier $c \in \mathcal{C}$, any non-faulty client process $p$ may invoke the following data access primitives during an execution $\xi$, where $c$ is added to specify the configuration specific implementation of the primitives:*

*D1:* $c$.get-tag() *that returns a tag $\tau \in \mathcal{T}$;*
*D2:* $c$.get-data() *that returns a tag-value pair $(\tau, v) \in \mathcal{T} \times \mathcal{V}$,*
*D3:* $c$.put-data($\langle \tau, v \rangle$) *which accepts the tag-value pair $(\tau, v) \in \mathcal{T} \times \mathcal{V}$ as argument.*

In order for the DAPs to be useful in designing the ARES algorithm we further require the following consistency properties. As we see later in Section 7, the safety property of ARES holds, given that these properties hold for the DAPs in each configuration.

PROPERTY 1 (DAP CONSISTENCY PROPERTIES). *In an execution $\xi$ we say that a DAP operation in an execution $\xi$ is complete if both the invocation and the matching response step appear in $\xi$. If $\Pi$ is the set of complete DAP operations in execution $\xi$ then for any $\phi, \pi \in \Pi$:*

*C1* *If $\phi$ is* $c$.put-data($\langle \tau_\phi, v_\phi \rangle$)*, for $c \in \mathcal{C}$, $\langle \tau_\phi, v_\phi \rangle \in \mathcal{T} \times \mathcal{V}$, and $\pi$ is* $c$.get-tag() *(or* $c$.get-data()*) that returns $\tau_\pi \in \mathcal{T}$ (or $\langle \tau_\pi, v_\pi \rangle \in \mathcal{T} \times \mathcal{V}$) and $\phi$ completes before $\pi$ is invoked in $\xi$, then $\tau_\pi \geq \tau_\phi$.*

*C2* *If $\phi$ is a $c$.get-data() that returns $\langle \tau_\pi, v_\pi \rangle \in \mathcal{T} \times \mathcal{V}$, then there exists $\pi$ such that $\pi$ is $c$.put-data($\langle \tau_\pi, v_\pi \rangle$) and $\phi$ did not complete before the invocation of $\pi$. If no such $\pi$ exists in $\xi$, then $(\tau_\pi, v_\pi)$ is equal to $(t_0, v_0)$.*

In Section 5 we show how to implement a set of DAPs, where erasure-codes are used to reduce storage and communication costs. Our DAP implementation satisfies Property 1.

As noted earlier, expressing ARES in terms of the DAPs allows one to achieve a modular design. Modularity enables the usage of different DAP implementation per configuration, during any execution of ARES, as long as the DAPs implemented in each configuration satisfy Property 1. For example, the DAPs in a configuration $c$ may be implemented using replication, while the DAPs in the next configuration say $c'$, may be implemented using erasure-codes. Thus, a system may use a scheme that offers higher fault tolerance (e.g. replication) when storage is not an issue, while switching to a more storage efficient (less fault-tolerant) scheme when storage gets limited.

In Section 8, we show that the presented DAPs are not only suitable for algorithm ARES, but can also be used to implement a large family of atomic read/write storage implementations. By describing an algorithm $A$ according to a simple algorithmic template (see Alg. 7), we show that $A$ preserves safety (atomicity) if the used DAPs satisfy Property 1, and $A$ preserves liveness (termination), if every invocation of the used DAPs terminate, under the failure model assumed.

## 4 ARES PROTOCOL

In this section, we describe ARES. In the presentation of ARES algorithm we decouple the reconfiguration service from the shared memory emulation, by utilizing the DAPs presented in Section 3. This allows ARES, to handle both the reorganization of the servers that host the data, as well as utilize a different atomic memory implementation per configuration. It is also important to note that ARES adopts a client-server architecture and separates the reader, writer and reconfiguration processes from the server processes that host the object data. More precisely, ARES algorithm comprises of three major components: $(i)$ The reconfiguration protocol which consists of invoking, and subsequently installing new configuration via the reconfig operation by recon clients. $(ii)$ The read/write protocol for executing the read and write operations invoked by readers and writers. $(iii)$ The implementation of the DAPs for each installed configuration that respect Property 1 and which are used by the reconfig, read and write operations.

### 4.1 Implementation of the Reconfiguration Service.

In this section, we describe the reconfiguration service in ARES. The service relies on an underlying sequence of configurations (already proposed or installed by reconfig operations), in the from of a "distributed list", which we refer to as the *global configuration sequence (or list)* $\mathcal{G}_L$. Conceptually, $\mathcal{G}_L$ represents an ordered list of pairs $\langle c, status \rangle$, where $c$ is a configuration identifier ($c \in C$), and a binary state variable $status \in \{F, P\}$ that denotes whether $c$ is *finalized* ($F$) or is still *pending* ($P$). Initially, $\mathcal{G}_L$ contains a single element, say $\langle c_0, F \rangle$, which is known to every participant in the service.

To facilitate the creation of $\mathcal{G}_L$, each server in $c.Servers$ maintains a local variable $nextC \in \{C \cup \{\bot\}\} \times \{P, F\}$, which is used to point to the configuration that follows $c$ in $\mathcal{G}_L$. Initially, at any server $nextC = \langle \bot, F \rangle$. Once $nextC$ it is set to a value it is never altered. As we show below, at any point in the execution of ARES and in any configuration $c$, the $nextC$ variables of the non-faulty servers in $c$ that are not equal to $\bot$ agree, i.e., $\{s.nextC : s \in c.Servers \wedge s.nextC \neq \bot\}$ is either empty of has only one element.

Clients discover the configuration that follows a $\langle c, * \rangle$ in the sequence by contacting a subset of servers in $c.Servers$ and collecting their $nextC$ variables. Every client in $\mathcal{I}$ maintains a local variable $cseq$ that is expected to be some subsequence of $\mathcal{G}_L$. Initially, at every client the value of $cseq$ is

$\langle c_0, F\rangle$. We use the notation $\widehat{x}$ (a caret over some name) to denote state variables that assumes values from the domain $\{C \cup \{\perp\}\} \times \{P, F\}$.

Reconfiguration clients may introduce new configurations, each associated with a unique configuration identifier from $C$. Multiple clients may concurrently attempt to introduce different configurations for same next link in $\mathcal{G}_L$. ARES uses consensus to resolve such conflicts: a subset of servers in $c.Servers$, in each configuration $c$, implements a distributed consensus service (such as Paxos [29], RAFT [35]) , denoted by $c.Con$.

The reconfiguration service consists of two major components: $(i)$ *sequence traversal*, responsible of discovering a recent configuration in $\mathcal{G}_L$, and $(ii)$ *reconfiguration operation* that installs new configurations in $\mathcal{G}_L$.

**Algorithm 1** Sequence traversal at each process $p \in \mathcal{I}$ of algorithm ARES.

```
    procedure read-config(seq)
2:    μ = max({j : seq[j].status = F})
      ĉ ← seq[μ]
4:    while ĉ ≠ ⊥ do
        ĉ' ←get-next-config(ĉ.cfg)
6:      if ĉ' ≠ ⊥ then
          μ ← μ + 1
8:        seq[μ] ← ĉ'
          put-config(seq[μ − 1].cfg, seq[μ])
10:       ĉ ← seq[μ]
      end while
12:   return seq
    end procedure

14: procedure get-next-config(c)
      send (READ-CONFIG) to each s ∈ c.Servers
16:   until ∃Q, Q ∈ c.Quorums s.t. rec_i receives
    nextC_s from ∀s ∈ Q
      if ∃s ∈ Q s.t. nextC_s.status = F then
18:     return nextC_s
      else if ∃s ∈ Q s.t. nextC_s.status = P then
20:     return nextC_s
      else
22:     return ⊥
    end procedure

24: procedure put-config(c, nextC)
      send (WRITE-CONFIG, nextC) to each s ∈
    c.Servers
26:   until ∃Q, Q ∈ c.Quorums s.t. rec_i receives ACK
    from ∀s ∈ Q
    end procedure
```

**Sequence Traversal.** Any read/write/reconfig operation $\pi$ utilizes the sequence traversal mechanism to discover the latest state of the global configuration sequence, as well as to ensure that such a state is discoverable by any subsequent operation $\pi'$. See Fig. 1 for an example execution in the case of a reconfig operation. In a high level, a client starts by collecting the $nextC$ variables from a quorum of servers in a configuration $c$, such that $\langle c, F\rangle$ is the last finalized configuration in that client's local $cseq$ variable (or $c_0$ if no other finalized configuration exists). If any server $s$ returns a $nextC$ variable such that $nextC.cfg \neq \perp$, then the client $(i)$ adds $nextC$ in its local $cseq$, $(ii)$ propagates $nextC$ in a quorum of servers in $c.Servers$, and $(iii)$ repeats this process in the configuration $nextC.cfg$. The client terminates when all servers reply with $nextC.cfg = \perp$. More precisely, the sequence parsing consists of three actions (see Alg. 1):

get-next-config$(c)$: The action get-next-config returns the configuration that follows $c$ in $\mathcal{G}_L$. During get-next-config$(c)$, a client sends READ-CONFIG messages to all the servers in $c.Servers$, and waits for replies containing $nextC$ from a quorum in $c.Quorums$. If there exists a reply with $nextC.cfg \neq \perp$ the action returns $nextC$; otherwise it returns $\perp$.

put-config$(c, c')$: The put-config$(c, c')$ action propagates $c'$ to a quorum of servers in $c.Servers$. During the action, the client sends (WRITE-CONFIG, $c'$) messages, to the servers in $c.Servers$ and waits for each server $s$ in some quorum $Q \in c.Quorums$ to respond.

read-config($seq$): A read-config($seq$) sequentially traverses the installed configurations in order to discover the latest state of the sequence $\mathcal{G}_L$. At invocation, the client starts with the last finalized configuration $\langle c, F\rangle$ in the given $seq$ (Line A1:2), and enters a loop to traverse $\mathcal{G}_L$ by invoking get-next-config($c$), which returns the next configuration, say $\widehat{c}'$. While $\widehat{c}' \neq \bot$, then: (a) $\widehat{c}'$ is appended at the end of the sequence $seq$; (b) a put-config($c, \widehat{c}'$) is invoked to inform a quorum of servers in $c.Servers$ to update the value of their $nextC$ variable to $\widehat{c}'$; and (c) variable $c$ is set to $\widehat{c}'.cfg$. If $\widehat{c}' = \bot$ the loop terminates and the action read-config returns $seq$.

**Algorithm 2** Reconfiguration protocol of algorithm ARES.

```
   at each reconfigurer rec_i
2: State Variables:
   cseq[]s.t.cseq[j] ∈ C × {F, P} with members:
4: Initialization:
   cseq[0] = ⟨c_0, F⟩

6: operation reconfig(c)
     if c ≠ ⊥ then
8:     cseq ←read-config(cseq)
       cseq ← add-config(cseq, c)
10:    update-config(cseq)
       cseq ← finalize-config(cseq)
12: end operation

   procedure add-config(seq, c)
14:  ν ← |seq|
     c' ← seq[ν].cfg
16:  d ← c'.Con.propose(c)
     seq[ν + 1] ← ⟨d, P⟩
18:  put-config(c', ⟨d, P⟩)
     return seq
20: end procedure

   procedure update-config(seq)
22:  μ ← max({j : seq[j].status = F})
     ν ← |seq|
24:  M ← ∅
     for i = μ : ν do
26:    ⟨t, v⟩ ← seq[i].cfg.get-data()
       M ← M ∪ {⟨τ, v⟩}
28:  ⟨τ, v⟩ ← max_t{⟨t, v⟩ : ⟨t, v⟩ ∈ M}
     seq[ν].put-data(⟨τ, v⟩)
30: end procedure

   procedure finalize-config(seq)
32:  ν = |seq|
     seq[ν].status ← F
34:  put-config(seq[ν − 1].cfg, seq[ν])
     return seq
36: end procedure
```

**Algorithm 3** Server protocol of algorithm ARES.

```
   at each server s_i in configuration c_k
2: State Variables:
   τ ∈ ℕ × 𝒲, initially, ⟨0, ⊥⟩
4: v ∈ V, intially, ⊥
   nextC ∈ C × {P, F}, initially ⟨⊥, P⟩

6: Upon receive (READ-CONFIG) s_i, c_k from q
     send nextC to q
8: end receive

   Upon receive (WRITE-CONFIG, cfgT_in) s_i, c_k from
   q
10:  if nextC.cfg = ⊥ ∨ nextC.status = P then
       nextC ← cfgT_in
12:  send ACK to q
   end receive
```

**Reconfiguration operation.** A reconfiguration operation reconfig($c$), $c \in C$, invoked by any reconfiguration client $rec_i$, attempts to append $c$ to $\mathcal{G}_L$. The set of server processes in $c$ are not a part of any other configuration different from $c$. In a high-level, $rec_i$ first executes a sequence traversal to discover the latest state of $\mathcal{G}_L$. Then it attempts to add the new configuration $c$, at the end of the discovered sequence by proposing $c$ in the consensus instance of the last configuration in the

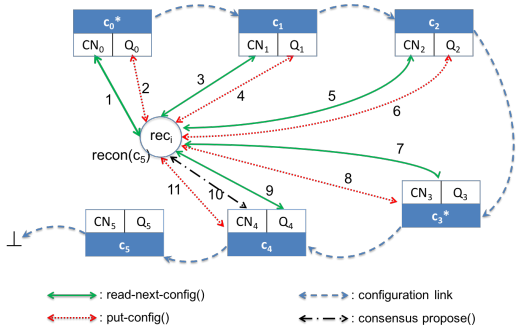


Fig. 1. Illustration of an execution of the reconfiguration steps.

sequence. The client accepts and appends the decision of the consensus instance (that might be different than $c$). Then it attempts to transfer the latest value of the read/write object to the latest installed configuration. Once the information is transferred, $rec_i$ finalizes the last configuration in its local sequence and propagates the finalized tuple to a quorum of servers in that configuration. The operation consists of four phases, executed consecutively by $rec_i$ (see Alg. 2):

read-config($seq$): The phase read-config($seq$) at $rec_i$, reads the recent global configuration sequence as described in the sequence traversal.

add-config($seq, c$): The add-config($seq, c$) attempts to append a new configuration $c$ to the end of $seq$ (client's view of $\mathcal{G}_L$). Suppose the last configuration in $seq$ is $c'$ (with status either $F$ or $P$), then in order to decide the most recent configuration, $rec_i$ invokes $c'.Con.propose(c)$, on the consensus object associated with configuration $c'$. Let $d \in \mathcal{C}$ be the configuration identifier decided by the consensus service. If $d \neq c$, this implies that another (possibly concurrent) reconfiguration operation, invoked by a reconfigurer $rec_j \neq rec_i$, proposed and succeeded $d$ as the configuration to follow $c'$. In this case, $rec_i$ adopts $d$ as it own propose configuration, by adding $\langle d, P\rangle$ to the end of its local $cseq$ (entirely ignoring $c$), using the operation put-config($c', \langle d, P\rangle$), and returns the extended configuration $seq$.

update-config($seq$): Let us denote by $\mu$ the index of the last configuration in the local sequence $cseq$, at $rec_i$, such that its corresponding status is $F$; and $\nu$ denote the last index of $cseq$. Next $rec_i$ invokes update-config($cseq$), which gathers the tag-value pair corresponding to the maximum tag in each of the configurations in $\widehat{cseq[i]}$ for $\mu \leq i \leq \nu$, and transfers that pair to the configuration that was added by the add-config action. The get-data and put-data DAPs are used to transfer the value of the object to the new configuration, and they are implemented with respect to the configuration that is accessed. Suppose $\langle t_{max}, v_{max}\rangle$ is the tag value pair corresponding to the highest tag among the responses from all the $\nu - \mu + 1$ configurations. Then, $\langle t_{max}, v_{max}\rangle$ is written to the configuration $d$ via the invocation of $\widehat{cseq[\nu]}.cfg$.put-data($\langle \tau_{max}, v_{max}\rangle$).

finalize-config($cseq$): Once the tag-value pair is transferred, in the last phase of the reconfiguration operation, $rec_i$ executes finalize-config($cseq$), to update the status of the last configuration in $cseq$, say $d = \widehat{cseq[\nu]}.cfg$, to $F$. The reconfigurer $rec_i$ informs a quorum of servers in the previous configuration $c = \widehat{cseq[\nu-1]}.cfg$, i.e. in some $Q \in c.Quorums$, about the change of status, by executing the put-config($c, \langle d, F\rangle$) action.

**Server Protocol.** Each server responds to requests from clients (Alg. 3). A server waits for two types of messages: READ-CONFIG and WRITE-CONFIG. When a READ-CONFIG message is received for a particular configuration $c_k$, then the server returns $nextC$ variables of the servers in $c_k.Servers$. A WRITE-CONFIG message attempts to update the $nextC$ variable of the server with a particular tuple

$cfgT_{in}$. A server changes the value of its local $nextC.cfg$ in two cases: (i) $nextC.cfg = \perp$, or (ii) $nextC.status = P$.

Fig. 1 illustrates an example execution of a reconfiguration operation recon($c_5$). In this example, the reconfigurer $rec_i$ goes through a number of configuration queries (read-next-config) before it reaches configuration $c_4$ in which a quorum of servers replies with $nextC.cfg = \perp$. There it proposes $c_5$ to the consensus object of $c_4$ ($c_4.Con.propose(c_5)$ on arrow 10), and once $c_5$ is decided, recon($c_5$) completes after executing finalize-config($c_5$).

**Algorithm 4** Write and Read protocols at the clients for ARES.

```
     Write Operation:
     at each writer w_i
     State Variables:
     cseq[]s.t.cseq[j] ∈ C × {F, P} with members:
     Initialization:
     cseq[0] = ⟨c_0, F⟩

     operation write(val), val ∈ V
       cseq ←read-config(cseq)
       μ ← max({i : cseq[i].status = F})
       ν ← |cseq|
       for i = μ : ν do
         τ_max ← max(cseq[i].cfg.get-tag(), τ_max)
       ⟨τ, v⟩ ← ⟨⟨τ_max.ts + 1, ω_i⟩, val⟩
       done ← false
       while not done do
         cseq[ν].cfg.put-data(⟨τ, v⟩)
         cseq ←read-config(cseq)
         if |cseq| = ν then
           done ← true
         else
           ν ← |cseq|
       end while
     end operation

     Read Operation:
     at each reader r_i
     State Variables:
     cseq[]s.t.cseq[j] ∈ C × {F, P} with members:
     Initialization:
     cseq[0] = ⟨c_0, F⟩

     operation read( )
       cseq ←read-config(cseq)
       μ ← max({j : cseq[j].status = F})
       ν ← |cseq|
       for i = μ : ν do
         ⟨τ, v⟩ ← max(cseq[i].cfg.get-data(), ⟨τ, v⟩)

       done ← false
       while not done do
         cseq[ν].cfg.put-data(⟨τ, v⟩)
         cseq ←read-config(cseq)
         if |cseq| = ν then
           done ← true
         else
           ν ← |cseq|
       end while
       return v
     end operation
```

## 4.2 Implementation of Read and Write operations.

The read and write operations in ARES are expressed in terms of the DAP primitives (see Section 3). This provides the flexibility to ARES to use different implementation of DAP primitives in different configurations, without changing the basic structure of ARES. At a high-level, a write (or read) operation is executed where the client: ($i$) obtains the *latest configuration sequence* by using the read-config action of the reconfiguration service, ($ii$) queries the configurations, in *cseq*, starting from the last finalized configuration to the end of the discovered configuration sequence, for the latest $\langle tag, value\rangle$ pair with a help of get-tag (or get-data) operation as specified for each configuration, and ($iii$) repeatedly propagates a new $\langle tag', value'\rangle$ pair (the largest $\langle tag, value\rangle$ pair) with put-data in the last configuration of its local sequence, until no additional configuration is observed. In more detail, the algorithm of a read or write operation $\pi$ is as follows (see Alg. 4):

A write (or read) operation is invoked at a client $p$ when line Alg. 4:8 (resp. line Alg. 4:31) is executed. At first, $p$ issues a read-config action to obtain the latest introduced configuration in $\mathcal{G}_L$, in both operations.

If $\pi$ is a write $p$ detects the last finalized entry in $cseq$, say $\mu$, and performs a $cseq[j].conf$.get-tag() action, for $\mu \leq j \leq |cseq|$ (line Alg. 4:9). Then $p$ discovers the *maximum tag* among all the returned tags ($\tau_{max}$), and it increments the maximum tag discovered (by incrementing the integer part of $\tau_{max}$), generating a new tag, say $\tau_{new}$. It assigns $\langle \tau, v \rangle$ to $\langle \tau_{new}, val \rangle$, where $val$ is the value he wants to write (Line Alg. 4:13).

if $\pi$ is a read, $p$ detects the last finalized entry in $cseq$, say $\mu$, and performs a $cseq[j].conf$.get-data() action, for $\mu \leq j \leq |cseq|$ (line Alg. 4:32). Then $p$ discovers the *maximum tag-value* pair ($\langle \tau_{max}, v_{max} \rangle$) among the replies, and assigns $\langle \tau, v \rangle$ to $\langle \tau_{max}, v_{max} \rangle$.

Once specifying the $\langle \tau, v \rangle$ to be propagated, both reads and writes execute the $cseq[\nu].cfg$.put-data($\langle \tau, v \rangle$) action, where $\nu = |cseq|$, followed by executing read-config action, to examine whether new configurations were introduced in $\mathcal{G}_L$. The repeat these steps until no new configuration is discovered (lines Alg. 4:15–21, or lines Alg. 4:37–43). Let $cseq'$ be the sequence returned by the read-config action. If $|cseq'| = |cseq|$ then no new configuration is introduced, and the read/write operation terminates; otherwise, $p$ sets $cseq$ to $cseq'$ and repeats the two actions. Note, in an execution of ARES, two consecutive read-config operations that return $cseq'$ and $cseq''$ respectively must hold that $cseq'$ is a prefix of $cseq''$, and hence $|cseq'| = |cseq''|$ only if $cseq' = cseq''$. Finally, if $\pi$ is a read operation the value with the highest tag discovered is returned to the client.

**Discussion** ARES shares similarities with previous algorithms like RAMBO [22] and the framework in [38]. The reconfiguration technique used in ARES ensures the prefix property on the configuration sequence (resembling a blockchain data structure [33]) while the array structure in RAMBO allowed nodes to maintain an incomplete reconfiguration history. On the other hand, the DAP usage, exploits a different viewpoint compared to [38], allowing implementations of atomic read/write registers without relying on strong objects, like ranked registers [13].

## 5 IMPLEMENTATION OF THE DAPS

In this section, we present an implementation of the DAPs, that satisfies the properties in Property 1, for a configuration $c$, with $n$ servers using a $[n, k]$ MDS coding scheme for storage. We implement an instance of the algorithm in a configuration of $n$ server processes. We store each coded element $c_i$, corresponding to an object at server $s_i$, where $i = 1, \cdots, n$. The implementations of DAP primitives used in ARES are shown in Alg. 5, and the servers' responses in Alg. 6.

Each server $s_i$ stores one state variable, $List$, which is a set of up to $(\delta + 1)$ (tag, coded-element) pairs. Initially the set at $s_i$ contains a single element, $List = \{(t_0, \Phi_i(v_0)\}$. Below we describe the implementation of the DAPs.

$c$.get-tag(): A client, during the execution of a $c$.get-tag() primitive, queries all the servers in $c.Servers$ for the highest tags in their $Lists$, and awaits responses from $\left\lceil \frac{n+k}{2} \right\rceil$ servers. A server upon receiving the GET-TAG request, responds to the client with the highest tag, as $\tau_{max} \equiv \max_{(t,c) \in List} t$. Once the client receives the tags from $\left\lceil \frac{n+k}{2} \right\rceil$ servers, it selects the highest tag $t$ and returns it .

$c$.put-data($\langle t_w, v \rangle$): During the execution of the primitive $c$.put-data($\langle t_w, v \rangle$), a client sends the pair $(t_w, \Phi_i(v))$ to each server $s_i \in c.Servers$. When a server $s_i$ receives a message (PUT-DATA, $t_w, c_i$) , it adds the pair in its local $List$, trims the pairs with the smallest tags exceeding the length $(\delta + 1)$ of the $List$ , and replies with an ack to the client. In particular, $s_i$ replaces the coded-elements of the older tags with $\perp$, and maintains only the coded-elements associated with the $(\delta + 1)$ highest tags in the $List$ (see Line Alg. 6:16). The client completes the primitive operation after getting acks from $\left\lceil \frac{n+k}{2} \right\rceil$ servers.

**Algorithm 5** DAP implementation for ARES.

at each process $p_i \in \mathcal{I}$

2: **procedure** c.get-tag()
  **send** (QUERY-TAG) to each $s \in c.Servers$
4: **until** $p_i$ receives $\langle t_s, e_s \rangle$ from $\lceil \frac{n+k}{2} \rceil$ servers in $c.Servers$
  $t_{max} \leftarrow \max(\{t_s : \text{received } \langle t_s, v_s \rangle \text{ from } s\})$
6: **return** $t_{max}$
**end procedure**

8: **procedure** c.get-data()
  **send** (QUERY-LIST) to each $s \in c.Servers$
10: **until** $p_i$ receives $List_s$ from each server $s \in \mathcal{S}_g$ s.t. $|\mathcal{S}_g| = \lceil \frac{n+k}{2} \rceil$ and $\mathcal{S}_g \subset c.Servers$
  $Tags^{\geq k}_{*}$ = set of tags that appears in $k$ lists
12: $Tags^{\geq k}_{dec}$ = set of tags that appears in $k$ lists with values
  $t^{*}_{max} \leftarrow \max Tags^{\geq k}_{*}$
14: $t^{dec}_{max} \leftarrow \max Tags^{\geq k}_{dec}$
  **if** $t^{dec}_{max} = t^{*}_{max}$ **then**
16: $v \leftarrow$ decode value for $t^{dec}_{max}$
  **return** $\langle t^{dec}_{max}, v \rangle$
18: **end procedure**

**procedure** c.put-data($\langle \tau, v \rangle$))
20: $code\text{-}elems = [(\tau, e_1), \ldots, (\tau, e_n)]$, $e_i = \Phi_i(v)$
  **send** (WRITE, $\langle \tau, e_i \rangle$) to each $s_i \in c.Servers$
22: **until** $p_i$ receives ACK from $\lceil \frac{n+k}{2} \rceil$ servers in $c.Servers$
**end procedure**

**Algorithm 6** The response protocols at any server $s_i \in \mathcal{S}$ in ARES for client requests.

at each server $s_i \in \mathcal{S}$ in configuration $c_k$

2: **State Variables:**
$List \subseteq \mathcal{T} \times C_s$, initially $\{(t_0, \Phi_i(v_0))\}$

**Upon receive** (QUERY-TAG) $_{s_i,\, c_k}$ **from** $q$
4: $\tau_{max} = \max_{(t,c) \in List} t$
  Send $\tau_{max}$ to $q$
6: **end receive**

**Upon receive** (QUERY-LIST) $_{s_i,\, c_k}$ **from** $q$
8: Send $List$ to $q$
**end receive**
10:
**Upon receive** (PUT-DATA, $\langle \tau, e_i \rangle$) $_{s_i,\, c_k}$ **from** $q$
12: $List \leftarrow List \cup \{\langle \tau, e_i \rangle\}$
  **if** $|List| > \delta + 1$ **then**
14: $\tau_{min} \leftarrow \min\{t : \langle t, * \rangle \in List\}$
/* remove the coded value and retain the tag */
  $List \leftarrow List \setminus \{\langle \tau, e \rangle : \tau = \tau_{min} \wedge \langle \tau, e \rangle \in List\}$
16: $List \leftarrow List \cup \{(\tau_{min}, \perp)\}$
  Send ACK to $q$
18: **end receive**

$c$.get-data(): A client, during the execution of a $c$.get-data() primitive, queries all the servers in $c.Servers$ for their local variable $List$, and awaits responses from $\lceil \frac{n+k}{2} \rceil$ servers. Once the client receives $Lists$ from $\lceil \frac{n+k}{2} \rceil$ servers, it selects the highest tag $t$, such that: $(i)$ its corresponding value $v$ is decodable from the coded elements in the lists; and $(ii)$ $t$ is the highest tag seen from the responses of at least $k$ $Lists$ (see lines Alg. 5:11-14) and returns the pair $(t, v)$. Note that in the case where anyone of the above conditions is not satisfied the corresponding read operation does not complete.

## 5.1 Safety (Property 1) proof of the DAPs

**Correctness.** In this section we are concerned with only one configuration $c$, consisting of a set of servers $c.Servers$. We assume that at most $f \leq \frac{n-k}{2}$ servers from $c.Servers$ may crash. Lemma 9 states that the DAP implementation satisfies the consistency properties Property 1 which will be used to imply the atomicity of the ARES algorithm.

THEOREM 2 (SAFETY). *Let* $\Pi$ *a set of complete DAP operations of Algorithm 5 in a configuration* $c \in C$, c.get-tag, c.get-data *and* c.put-data*, of an execution* $\xi$. *Then, every pair of operations* $\phi, \pi \in \Pi$ *satisfy Property 1.*

**Liveness.** To reason about the liveness of the proposed DAPs, we define a concurrency parameter $\delta$ which captures all the put-data operations that overlap with the get-data, until the time the client has all data needed to attempt decoding a value. However, we ignore those put-data operations that might have started in the past, and never completed yet, if their tags are less than that of any put-data that completed before the get-data started. This allows us to ignore put-data operations due to failed clients, while counting concurrency, as long as the failed put-data operations are followed by a successful put-data that completed before the get-data started. In order to define the amount of concurrency in our specific implementation of the DAPs presented in this section the following definition captures the put-data operations that overlap with the get-data, until the client has all data required to decode the value.

DEFINITION 3 (VALID get-data OPERATIONS). *A* get-data *operation* $\pi$ *from a process* $p$ *is valid if* $p$ *does not crash until the reception of* $\left\lceil \frac{n+k}{2} \right\rceil$ *responses during the get-data phase.*

DEFINITION 4 (put-data CONCURRENT WITH A VALID get-data). *Consider a valid* get-data *operation* $\pi$ *from a process* $p$. *Let* $T_1$ *denote the point of initiation of* $\pi$. *For* $\pi$, *let* $T_2$ *denote the earliest point of time during the execution when* $p$ *receives all the* $\left\lceil \frac{n+k}{2} \right\rceil$ *responses. Consider the set* $\Sigma = \{\phi : \phi$ *is any* put-data *operation that completes before* $\pi$ *is initiated*$\}$, *and let* $\phi^* = \arg\max_{\phi \in \Sigma} tag(\phi)$. *Next, consider the set* $\Lambda = \{\lambda : \lambda$ *is any* put-data *operation that starts before* $T_2$ *such that* $tag(\lambda) > tag(\phi^*)\}$. *We define the number of* put-data *concurrent with the valid* get-data $\pi$ *to be the cardinality of the set* $\Lambda$.

Termination (and hence liveness) of the DAPs is guaranteed in an execution $\xi$, provided that a process no more than $f$ servers in $c.Servers$ crash, and no more that $\delta$ put-data may be concurrent at any point in $\xi$. If the failure model is satisfied, then any operation invoked by a non-faulty client will collect the necessary replies independently of the progress of any other client process in the system. Preserving $\delta$ on the other hand, ensures that any operation will be able to decode a written value. These are captured in the following theorem:

THEOREM 5 (LIVENESS). *Let* $\xi$ *be well-formed and fair execution of DAPs, with an* $[n, k]$ *MDS code, where* $n$ *is the number of servers out of which no more than* $\frac{n-k}{2}$ *may crash, and* $\delta$ *be the maximum number of* put-data *operations concurrent with any valid* get-data *operation. Then any* get-data *and* put-data *operation* $\pi$ *invoked by a process* $p$ *terminates in* $\xi$ *if* $p$ *does not crash between the invocation and response steps of* $\pi$.

## 6 CORRECTNESS OF ARES

In this section, we prove that ARES correctly implements an atomic, read/write, shared storage service. In particular, we show that ARES ensures atomicity iff the DAP implementation in each configuration $c_i$ satisfies Property 1. The correctness of ARES highly depends on the way the configuration sequence is constructed at each client process. We begin by showing some critical properties preserved by the reconfiguration service proposed in ARES in subsection 6.2. Given those properties and the fact that the DAP

Therefore, we start by proving, in subsection 6.1, that Property 1 holds for the DAP implementation in Section 5. Based on this, in subsection 7, we prove the atomicty of ARES. Next, in sub-section 7.1, we derive the storage and communication costs of read and write operations, and in sub-section 7.2, we derive the latency of reads and writes in terms of the minimum and maximum delays of any point-to-point messages of the underlying network.

We proceed by first introducing some definitions and notation, that we will use in the proofs that follow.

**Notations and definitions.** For a server $s$, we use the notation $s.var|_\sigma$ to refer to the value of the state variable $var$, in $s$, at a state $\sigma$ of an execution $\xi$. If server $s$ crashes at a state $\sigma_f$ in an execution $\xi$ then $s.var|_\sigma \triangleq s.var|_{\sigma_f}$ for any state variable $var$ and for any state $\sigma$ that appears after $\sigma_f$ in $\xi$.

DEFINITION 6 (TAG OF A CONFIGURATION). *Let $c \in C$ be a configuration, $\sigma$ be a state in some execution $\xi$ then we define the tag of $c$ at state $\sigma$ as $tag(c)|_\sigma \triangleq \min_{Q \in c.Quorums} \max_{s \in Q} (s.tag|_\sigma)$. We drop the suffix $|_\sigma$, and simply denote as $tag(c)$, when the state is clear from the context.*

DEFINITION 7. *Let $\sigma$ be any point in an execution of* ARES *and suppose we use the notation $\mathbf{c}^p_\sigma$ for $p.cseq|_\sigma$, i.e., the cseq variable at process $p$ at the state $\sigma$. Then we define as $\mu(\mathbf{c}^p_\sigma) \triangleq \max\{i : \mathbf{c}^p_\sigma[i].status = F\}$ and $\nu(\mathbf{c}^p_\sigma) \triangleq |\mathbf{c}^p_\sigma|$, where $|\mathbf{c}^p_\sigma|$ is the length of the configuration vector $\mathbf{c}^p_\sigma$.*

DEFINITION 8 (PREFIX ORDER). *Let $\mathbf{x}$ and $\mathbf{y}$ be any two configuration sequences. We say that $\mathbf{x}$ is a prefix of $\mathbf{y}$, denoted by $\mathbf{x} \preceq_p \mathbf{y}$, if $\mathbf{x}[j].cfg = \mathbf{y}[j].cfg$, for all $j$ such that $\mathbf{x}[j] \neq \perp$.*

## 6.1 Safety (Property 1) proof of the DAPs

**Correctness.** In this section we are concerned with only one configuration $c$, consisting of a set of servers $c.Servers$. We assume that at most $f \leq \frac{n-k}{2}$ servers from $c.Servers$ may crash. Lemma 9 states that the DAP implementation satisfies the consistency properties Property 1 which will be used to imply the atomicity of the ARES algorithm.

THEOREM 9 (SAFETY). *Let $\Pi$ a set of complete DAP operations of Algorithm 5 in a configuration $c \in C$,* c.get-tag*,* c.get-data *and* c.put-data*, of an execution $\xi$. Then, every pair of operations $\phi, \pi \in \Pi$ satisfy Property 1.*

**Liveness.** To reason about the liveness of the proposed DAPs, we define a concurrency parameter $\delta$ which captures all the put-data operations that overlap with the get-data, until the time the client has all data needed to attempt decoding a value. However, we ignore those put-data operations that might have started in the past, and never completed yet, if their tags are less than that of any put-data that completed before the get-data started. This allows us to ignore put-data operations due to failed clients, while counting concurrency, as long as the failed put-data operations are followed by a successful put-data that completed before the get-data started. In order to define the amount of concurrency in our specific implementation of the DAPs presented in this section the following definition captures the put-data operations that overlap with the get-data, until the client has all data required to decode the value.

DEFINITION 10 (VALID get-data OPERATIONS). *A* get-data *operation $\pi$ from a process $p$ is valid if $p$ does not crash until the reception of $\left\lceil \frac{n+k}{2} \right\rceil$ responses during the get-data phase.*

DEFINITION 11 (put-data CONCURRENT WITH A VALID get-data). *Consider a valid* get-data *operation $\pi$ from a process $p$. Let $T_1$ denote the point of initiation of $\pi$. For $\pi$, let $T_2$ denote the earliest point of time during the execution when $p$ receives all the $\left\lceil \frac{n+k}{2} \right\rceil$ responses. Consider the set $\Sigma = \{\phi : \phi$ is any* put-data *operation that completes before $\pi$ is initiated$\}$, and let $\phi^* = \arg\max_{\phi \in \Sigma} tag(\phi)$. Next, consider the set $\Lambda = \{\lambda : \lambda$ is any* put-data *operation that starts before $T_2$ such that $tag(\lambda) > tag(\phi^*)\}$. We define the number of* put-data *concurrent with the valid* get-data *$\pi$ to be the cardinality of the set $\Lambda$.*

Termination (and hence liveness) of the DAPs is guaranteed in an execution $\xi$, provided that a process no more than $f$ servers in $c.Servers$ crash, and no more that $\delta$ put-data may be concurrent

at any point in $\xi$. If the failure model is satisfied, then any operation invoked by a non-faulty client will collect the necessary replies independently of the progress of any other client process in the system. Preserving $\delta$ on the other hand, ensures that any operation will be able to decode a written value. These are captured in the following theorem:

THEOREM 12 (LIVENESS). *Let $\xi$ be well-formed and fair execution of DAPs, with an $[n, k]$ MDS code, where n is the number of servers out of which no more than $\frac{n-k}{2}$ may crash, and $\delta$ be the maximum number of* put-data *operations concurrent with any valid* get-data *operation. Then any* get-data *and* put-data *operation $\pi$ invoked by a process p terminates in $\xi$ if p does not crash between the invocation and response steps of $\pi$.*

### 6.2 Reconfiguration Protocol Properties

In this section we analyze the properties that we can achieve through our reconfiguration algorithm. The first lemma shows that any two configuration sequences have the same configuration identifiers in the same indexes.

LEMMA 13. *For any reconfigurer r that invokes an* reconfig($c$) *action in an execution $\xi$ of the algorithm, If r chooses to install c in index k of its local r.cseq vector, then r invokes the $Cons[k-1].propose(c)$ instance over configuration $r.cseq[k-1].cfg$.*

PROOF. It follows directly from the algorithm. □

LEMMA 14. *If a server s sets $s.nextC$ to $\langle c, F\rangle$ at some state $\sigma$ in an execution $\xi$ of the algorithm, then $s.nextC = \langle c, F\rangle$ for any state $\sigma'$ that appears after $\sigma$ in $\xi$.*

PROOF. Notice that a server $s$ updates the $s.nextC$ variable for some specific configuration $c_k$ in a state $st$ if: (i) $s$ did not receive any value for $c_k$ before (and thus $nextC = \perp$), or (ii) $s$ received a tuple $\langle c, P\rangle$ and before $\sigma$ received the tuple $\langle c', F\rangle$. By Observation **??** $c = c'$ as $s$ updates the $s.nextC$ of the same configuration $c_k$. Once the tuple becomes equal to $\langle c, F\rangle$ then $s$ does not satisfy the update condition for $c_k$, and hence in any state $\sigma'$ after $\sigma$ it does not change $\langle c, F\rangle$. □

LEMMA 15 (CONFIGURATION UNIQUENESS). *For any processes $p, q \in \mathcal{I}$ and any states $\sigma_1, \sigma_2$ in an execution $\xi$, it must hold that $\mathbf{c}^p_{\sigma_1}[i].cfg = \mathbf{c}^q_{\sigma_2}[i].cfg$, $\forall i$ s.t. $\mathbf{c}^p_{\sigma_1}[i].cfg, \mathbf{c}^q_{\sigma_2}[i].cfg \neq \perp$.*

PROOF. The lemma holds trivially for $\mathbf{c}^p_{\sigma_1}[0].cfg = \mathbf{c}^q_{\sigma_2}[0].cfg = c_0$. So in the rest of the proof we focus in the case where $i > 0$. Let us assume w.l.o.g. that $\sigma_1$ appears before $\sigma_2$ in $\xi$.

According to our algorithm a process $p$ sets $p.cseq[i].cfg$ to a configuration identifier $c$ in two cases: (i) either it received $c$ as the result of the consensus instance in configuration $p.cseq[i-1].cfg$, or (ii) $p$ receives $s.nextC.cfg = c$ from a server $s \in p.cseq[i-1].cfg.Servers$. Note here that (i) is possible only when $p$ is a reconfigurer and attempts to install a new configuration. On the other hand (ii) may be executed by any process in any operation that invokes the read-config action. We are going to proof this lemma by induction on the configuration index.

*Base case:* The base case of the lemma is when $i = 1$. Let us first assume that $p$ and $q$ receive $c_p$ and $c_q$, as the result of the consensus instance at $p.cseq[0].cfg$ and $q.cseq[0].cfg$ respectively. By Lemma 13, since both processes want to install a configuration in $i = 1$, then they have to run $Cons[0]$ instance over the configuration stored in their local $cseq[0].cfg$ variable. Since $p.cseq[0].cfg = q.cseq[0].cfg = c_0$ then both $Cons[0]$ instances run over the same configuration $c_0$ and according to Observation **??** return the same value, say $c_1$. Hence $c_p = c_q = c_1$ and $p.cseq[1].cfg = q.cseq[1].cfg = c_1$.

Let us examine the case now where $p$ or $q$ assign a configuration $c$ they received from some server $s \in c_0.Servers$. According to the algorithm only the configuration that has been decided by the consensus instance on $c_0$ is propagated to the servers in $c_0.Servers$. If $c_1$ is the decided

configuration, then $\forall s \in c_0.Servers$ such that $s.nextC(c_0) \neq \bot$, it holds that $s.nextC(C_0) = \langle c_1, * \rangle$. So if $p$ or $q$ set $p.cseq[1].cfg$ or $q.cseq[1].cfg$ to some received configuration, then $p.cseq[1].cfg = q.cseq[1].cfg = c_1$ in this case as well.

*Hypothesis:* We assume that $\mathbf{c}^p_{\sigma_1}[k] = \mathbf{c}^q_{\sigma_2}[k]$ for some $k$, $k \geq 1$.

*Induction Step:* We need to show that the lemma holds for $i = k + 1$. If both processes retrieve $p.cseq[k+1].cfg$ and $q.cseq[k+1].cfg$ through consensus, then both $p$ and $q$ run consensus over the previous configuration. Since according to our hypothesis $\mathbf{c}^p_{\sigma_1}[k] = \mathbf{c}^q_{\sigma_2}[k]$ then both process will receive the same decided value, say $c_{k+1}$, and hence $p.cseq[k+1].cfg = q.cseq[k+1].cfg = c_{k+1}$. Similar to the base case, a server in $c_k.Servers$ only receives the configuration $c_{k+1}$ decided by the consensus instance run over $c_k$. So processes $p$ and $q$ can only receive $c_{k+1}$ from some server in $c_k.Servers$ so they can only assign $p.cseq[k+1].cfg = q.cseq[k+1].cfg = c_{k+1}$ at Line 2:8. That completes the proof. □

Lemma 15 showed that any two operations store the same configuration in any cell $k$ of their *cseq* variable. It is not known however if the two processes discover the same number of configuration ids. In the following lemmas we will show that if a process learns about a configuration in a cell $k$ then it also learns about all configuration ids for every index $i$, such that $0 \leq i \leq k-1$.

LEMMA 16. *In any execution $\xi$ of the algorithm , If for any process $p \in \mathcal{I}$, $\mathbf{c}^p_{\sigma}[i] \neq \bot$ in some state $\sigma$ in $\xi$, then $\mathbf{c}^p_{\sigma'}[i] \neq \bot$ in any state $\sigma'$ that appears after $\sigma$ in $\xi$.*

PROOF. A value is assigned to $\mathbf{c}^p_*[i]$ either after the invocation of a consensus instance, or while executing the read-config action. Since any configuration proposed for installation cannot be $\bot$ (A2:7), and since there is at least one configuration proposed in the consensus instance (the one from $p$), then by the validity of the consensus service the decision will be a configuration $c \neq \bot$. Thus, in this case $\mathbf{c}^p_*[i]$ cannot be $\bot$. Also in the read-config procedure, $\mathbf{c}^p_*[i]$ is assigned to a value different than $\bot$ according to Line A2:L8. Hence, if $\mathbf{c}^p_{\sigma}[i] \neq \bot$ at state $\sigma$ then it cannot become $\bot$ in any state $\sigma'$ after $\sigma$ in execution $\xi$. □

LEMMA 17. *Let $\sigma_1$ be some state in an execution $\xi$ of the algorithm. Then for any process p, if $k = max\{i : \mathbf{c}^p_{\sigma_1}[i] \neq \bot\}$, then $\mathbf{c}^p_{\sigma_1}[j] \neq \bot$, for $0 \leq j < k$.*

PROOF. Let us assume to derive contradiction that there exists $j < k$ such that $\mathbf{c}^p_{\sigma_1}[j] = \bot$ and $\mathbf{c}^p_{\sigma_1}[j+1] \neq \bot$. Suppose w.l.o.g. that $j = k-1$ and that $\sigma_1$ is the state immediately after the assignment of a value to $\mathbf{c}^p_{\sigma_1}[k]$, say $c_k$. Since $\mathbf{c}^p_{\sigma_1}[k] \neq \bot$, then $p$ assigned $c_k$ to $\mathbf{c}^p_{\sigma_1}[k]$ in one of the following cases: (i) $c_k$ was the result of the consensus instance, or (ii) $p$ received $c_k$ from a server during a read-config action. The first case is trivially impossible as according to Lemma 13 $p$ decides for $k$ when it runs consensus over configuration $\mathbf{c}^p_{\sigma_1}[k-1].cfg$. Since this is equal to $\bot$, then we cannot run consensus over a non-existent set of processes. In the second case, $p$ assigns $\mathbf{c}^p_{\sigma_1}[k] = c_k$ in Line A1:8. The value $c_k$ was however obtained when $p$ invoked get-next-config on configuration $\mathbf{c}^p_{\sigma_1}[k-1].cfg$. In that action, $p$ sends READ-CONFIG messages to the servers in $\mathbf{c}^p_{\sigma_1}[k-1].cfg.Servers$ and waits until a quorum of servers replies. Since we assigned $\mathbf{c}^p_{\sigma_1}[k] = c_k$ it means that get-next-config terminated at some state $\sigma'$ before $\sigma_1$ in $\xi$, and thus: (a) a quorum of servers in $\mathbf{c}^p_{\sigma'}[k-1].cfg.Servers$ replied, and (b) there exists a server $s$ among those that replied with $c_k$. According to our assumption however, $\mathbf{c}^p_{\sigma_1}[k-1] = \bot$ at $\sigma_1$. So if state $\sigma'$ is before $\sigma_1$ in $\xi$, then by Lemma 16, it follows that $\mathbf{c}^p_{\sigma'}[k-1] = \bot$. This however implies that $p$ communicated with an empty configuration, and thus no server replied to $p$. This however contradicts the assumption that a server replied with $c_k$ to $p$.

Since any process traverses the configuration sequence starting from the initial configuration $c_0$, then with a simple induction we can show that $\mathbf{c}^p_{\sigma_1}[j] \neq \bot$, for $0 \leq j \leq k$. □

We can now move to an important lemma that shows that any read-config action returns an extension of the configuration sequence returned by any previous read-config action. First, we show that the last finalized configuration observed by any read-config action is at least as recent as the finalized configuration observed by any subsequent read-config action.

LEMMA 18. *If at a state $\sigma$ of an execution $\xi$ of the algorithm, if $\mu(\mathbf{c}^p_\sigma) = k$ for some process $p$, then for any element $0 \le j < k$, $\exists Q \in \mathbf{c}^p_\sigma[j].cfg.Quorums$ such that $\forall s \in Q, s.nextC(\mathbf{c}^p_\sigma[j].cfg) = \mathbf{c}^p_\sigma[j+1]$.*

PROOF. This lemma follows directly from the algorithm. Notice that whenever a process assigns a value to an element of its local configuration (Lines A1:8 and A2:17), it then propagates this value to a quorum of the previous configuration (Lines A1:9 and A2:18). So if a process $p$ assigned $c_j$ to an element $\mathbf{c}^p_{\sigma'}[j]$ in some state $\sigma'$ in $\xi$, then $p$ may assign a value to the $j+1$ element of $\mathbf{c}^p_{\sigma''}[j+1]$ only after put-config$(\mathbf{c}^p_{\sigma'}[j-1].cfg, \mathbf{c}^p_{\sigma'}[j])$ occurs. During put-config action, $p$ propagates $\mathbf{c}^p_{\sigma'}[j]$ in a quorum $Q \in \mathbf{c}^p_{\sigma'}[j-1].cfg.Quorums$. Hence, if $\mathbf{c}^p_\sigma[k] \neq \perp$, then $p$ propagated each $\mathbf{c}^p_{\sigma'}[j]$, for $0 < j \le k$ to a quorum of servers $Q \in \mathbf{c}^p_{\sigma'}[j-1].cfg.Quorums$. And this completes the proof. □

LEMMA 19 (CONFIGURATION PREFIX). *Let $\pi_1$ and $\pi_2$ two completed* read-config *actions invoked by processes $p_1, p_2 \in \mathcal{I}$ respectively, such that $\pi_1 \to \pi_2$ in an execution $\xi$. Let $\sigma_1$ be the state after the response step of $\pi_1$ and $\sigma_2$ the state after the response step of $\pi_2$. Then $\mathbf{c}^{p_1}_{\sigma_1} \preceq_p \mathbf{c}^{p_2}_{\sigma_2}$.*

PROOF. Let $\nu_1 = \nu(\mathbf{c}^{p_1}_{\sigma_1})$ and $\nu_2 = \nu(\mathbf{c}^{p_2}_{\sigma_2})$. By Lemma 15 for any $i$ such that $\mathbf{c}^{p_1}_{\sigma_1}[i] \neq \perp$ and $\mathbf{c}^{p_2}_{\sigma_2}[i] \neq \perp$, then $\mathbf{c}^{p_1}_{\sigma_1}[i].cfg = \mathbf{c}^{p_2}_{\sigma_2}[i].cfg$. Also from Lemma 17 we know that for $0 \le j \le \nu_1$, $\mathbf{c}^{p_1}_{\sigma_1}[j] \neq \perp$, and $0 \le j \le \nu_2$, $\mathbf{c}^{p_2}_{\sigma_2}[j] \neq \perp$. So if we can show that $\nu_1 \le \nu_2$ then the lemma follows.

Let $\mu = \mu(\mathbf{c}^{p_2}_{\sigma'})$ be the last finalized element which $p_2$ established in the beginning of the read-config action $\pi_2$ (Line A2:2) at some state $\sigma'$ before $\sigma_2$. It is easy to see that $\mu \le \nu_2$. If $\nu_1 \le \mu$ then $\nu_1 \le \nu_2$ and the lemma follows. Thus, it remains to examine the case where $\mu < \nu_1$. Notice that since $\pi_1 \to \pi_2$ then $\sigma_1$ appears before $\sigma'$ in execution $\xi$. By Lemma 18, we know that by $\sigma_1$, $\exists Q \in \mathbf{c}^{p_1}_{\sigma_1}[j].cfg.Quorums$, for $0 \le j < \nu_1$, such that $\forall s \in Q, s.nextC = \mathbf{c}^{p_1}_{\sigma_1}[j+1]$. Since $\mu < \nu_1$, then it must be the case that $\exists Q \in \mathbf{c}^{p_1}_{\sigma_1}[\mu].cfg.Quorums$ such that $\forall s \in Q, s.nextC = \mathbf{c}^{p_1}_{\sigma_1}[\mu+1]$. But by Lemma 15, we know that $\mathbf{c}^{p_1}_{\sigma_1}[\mu].cfg = \mathbf{c}^{p_2}_{\sigma'}[\mu].cfg$. Let $Q'$ be the quorum that replies to the read-next-config occurred in $p_2$, on configuration $\mathbf{c}^{p_2}_{\sigma'}[\mu].cfg$. By definition $Q \cap Q' \neq \emptyset$, thus there is a server $s \in Q \cap Q'$ that sends $s.nextC = \mathbf{c}^{p_1}_{\sigma_1}[\mu+1]$ to $p_2$ during $\pi_2$. Since $\mathbf{c}^{p_1}_{\sigma_1}[\mu+1] \neq \perp$ then $p_2$ assigns $\mathbf{c}^{p_2}_*[\mu+1] = \mathbf{c}^{p_1}_{\sigma_1}[\mu+1]$, and repeats the process in the configuration $\mathbf{c}^{p_2}_*[\mu+1].cfg$. Since every configuration $\mathbf{c}^{p_1}_{\sigma_1}[j].cfg$, for $\mu \le j < \nu_1$, has a quorum of servers with $s.nextC$, then by a simple induction it can be shown that the process will be repeated for at least $\nu_1 - \mu$ iterations, and every configuration $\mathbf{c}^{p_2}_{\sigma''}[j] = \mathbf{c}^{p_1}_{\sigma_1}[j]$, at some state $\sigma''$ before $\sigma_2$. Thus, $\mathbf{c}^{p_2}_{\sigma_2}[j] = \mathbf{c}^{p_1}_{\sigma_1}[j]$, for $0 \le j \le \nu_1$. Hence $\nu_1 \le \nu_2$ and the lemma follows in this case as well. □

Thus far we focused on the configuration member of each element in *cseq*. As operations do get in account the *status* of a configuration, i.e. *P* or *F*, in the next lemma we will examine the relationship of the last finalized configuration as detected by two operations. First we present a lemma that shows the monotonicity of the finalized configurations.

LEMMA 20. *Let $\sigma$ and $\sigma'$ two states in an execution $\xi$ such that $\sigma$ appears before $\sigma'$ in $\xi$. Then for any process $p$ must hold that $\mu(\mathbf{c}^p_\sigma) \le \mu(\mathbf{c}^p_{\sigma'})$.*

PROOF. This lemma follows from the fact that if a configuration $k$ is such that $\mathbf{c}^p_\sigma[k].status = F$ at a state $\sigma$, then $p$ will start any future read-config action from a configuration $\mathbf{c}^p_{\sigma'}[j].cfg$ such that $j \ge k$. But $\mathbf{c}^p_{\sigma'}[j].cfg$ is the last finalized configuration at $\sigma'$ and hence $\mu(\mathbf{c}^p_{\sigma'}) \ge \mu(\mathbf{c}^p_\sigma)$. □

LEMMA 21 (CONFIGURATION PROGRESS). *Let $\pi_1$ and $\pi_2$ two completed* read-config *actions invoked by processes $p_1, p_2 \in \mathcal{I}$ respectively, such that $\pi_1 \rightarrow \pi_2$ in an execution $\xi$. Let $\sigma_1$ be the state after the response step of $\pi_1$ and $\sigma_2$ the state after the response step of $\pi_2$. Then $\mu(\mathbf{c}^{p_1}_{\sigma_1}) \leq \mu(\mathbf{c}^{p_2}_{\sigma_2})$.*

PROOF. By Lemma 19 it follows that $\mathbf{c}^{p_1}_{\sigma_1}$ is a prefix of $\mathbf{c}^{p_2}_{\sigma_2}$. Thus, if $\nu_1 = \nu(\mathbf{c}^{p_1}_{\sigma_1})$ and $\nu_2 = \nu(\mathbf{c}^{p_2}_{\sigma_2})$, $\nu_1 \leq \nu_2$. Let $\mu_1 = \mu(\mathbf{c}^{p_1}_{\sigma_1})$, such that $\mu_1 \leq \nu_1$, be the last element in $\mathbf{c}^{p_1}_{\sigma_1}$, where $\mathbf{c}^{p_1}_{\sigma_1}[\mu_1].status = F$. Let now $\mu_2 = \mu(\mathbf{c}^{p_2}_{\sigma'})$, be the last element which $p_2$ obtained in Line A1:2 during $\pi_2$ such that $\mathbf{c}^{p_2}_{\sigma'}[\mu_2].status = F$ in some state $\sigma'$ before $\sigma_2$. If $\mu_2 \geq \mu_1$, and since $\sigma_2$ is after $\sigma'$, then by Lemma 20 $\mu_2 \leq \mu(\mathbf{c}^{p_2}_{\sigma_2})$ and hence $\mu_1 \leq \mu(\mathbf{c}^{p_2}_{\sigma_2})$ as well.

It remains to examine the case where $\mu_2 < \mu_1$. Process $p_1$ sets the status of $\mathbf{c}^{p_1}_{\sigma_1}[\mu_1]$ to $F$ in two cases: (i) either when finalizing a reconfiguration, or (ii) when receiving an $s.nextC = \langle \mathbf{c}^{p_1}_{\sigma_1}[\mu_1].cfg, F\rangle$ from some server $s$ during a read-config action. In both cases $p_1$ propagates the $\langle \mathbf{c}^{p_1}_{\sigma_1}[\mu_1].cfg, F\rangle$ to a quorum of servers in $\mathbf{c}^{p_1}_{\sigma_1}[\mu_1 - 1].cfg$ before completing. We know by Lemma 19 that since $\pi_1 \rightarrow \pi_2$ then $\mathbf{c}^{p_1}_{\sigma_1}$ is a prefix in terms of configurations of the $\mathbf{c}^{p_2}_{\sigma_2}$. So it must be the case that $\mu_2 < \mu_1 \leq \nu(\mathbf{c}^{p_2}_{\sigma_2})$. Thus, during $\pi_2$, $p_2$ starts from the configuration at index $\mu_2$ and in some iteration performs get-next-config in configuration $\mathbf{c}^{p_2}_{\sigma_2}[\mu_1 - 1]$. According to Lemma 15, $\mathbf{c}^{p_1}_{\sigma_1}[\mu_1 - 1].cfg = \mathbf{c}^{p_2}_{\sigma_2}[\mu_1 - 1].cfg$. Since $\pi_1$ completed before $\pi_2$, then it must be the case that $\sigma_1$ appears before $\sigma'$ in $\xi$. However, $p_2$ invokes the get-next-config operation in a state $\sigma''$ which is either equal to $\sigma'$ or appears after $\sigma'$ in $\xi$. Thus, $\sigma''$ must appear after $\sigma_1$ in $\xi$. From that it follows that when the get-next-config is executed by $p_2$ there is already a quorum of servers in $\mathbf{c}^{p_2}_{\sigma_2}[\mu_1 - 1].cfg$, say $Q_1$, that received $\langle \mathbf{c}^{p_1}_{\sigma_1}[\mu_1].cfg, F\rangle$from $p_1$. Since, $p_2$ waits from replies from a quorum of servers from the same configuration, say $Q_2$, and since the *nextC* variable at each server is monotonic (Lemma 14), then there is a server $s \in Q_1 \cap Q_2$, such that $s$ replies to $p_2$ with $s.nextC = \langle \mathbf{c}^{p_1}_{\sigma_1}[\mu_1].cfg, F\rangle$. So, $\mathbf{c}^{p_2}_{\sigma_2}[\mu_1]$ gets $\langle \mathbf{c}^{p_1}_{\sigma_1}[\mu_1].cfg, F\rangle$, and hence $\mu(\mathbf{c}^{p_2}_{\sigma_2}) \geq \mu_1$ in this case as well. This completes our proof. □

THEOREM 22. *Let $\pi_1$ and $\pi_2$ two completed* read-config *actions invoked by processes $p_1, p_2 \in \mathcal{I}$ respectively, such that $\pi_1 \rightarrow \pi_2$ in an execution $\xi$. Let $\sigma_1$ be the state after the response step of $\pi_1$ and $\sigma_2$ the state after the response step of $\pi_2$. Then the following properties hold:*

(a) $\mathbf{c}^{p_2}_{\sigma_2}[i].cfg = \mathbf{c}^{p_1}_{\sigma_1}[i].cfg$, *for* $1 \leq i \leq \nu(\mathbf{c}^{p_1}_{\sigma_1})$,
(b) $\mathbf{c}^{p_1}_{\sigma_1} \preceq_p \mathbf{c}^{p_2}_{\sigma_2}$, *and*
(c) $\mu(\mathbf{c}^{p_1}_{\sigma_1}) \leq \mu(\mathbf{c}^{p_2}_{\sigma_2})$

PROOF. Statements (*a*), (*b*) and (*c*) follow from Lemmas 15, 19, and 20. □

### 6.3 Atomicity Property of ARES

The correctness of ARES highly depends on the way the configuration sequence is constructed at each client process. Let $\mathbf{c}^{p}_{\sigma}$ denote the configuration sequence *cseq* at process $p$ in a state $\sigma$ and $\mu(\mathbf{c}^{p}_{\sigma})$ the index of the last finalized configuration in $\mathbf{c}^{p}_{\sigma}$. Then the following properties are preserved by the reconfiguration service:

THEOREM 23. *Let $\pi_1$ and $\pi_2$ two completed* read-config *actions invoked by processes $p_1, p_2 \in \mathcal{I}$ respectively, such that $\pi_1 \rightarrow \pi_2$ in an execution $\xi$. Let $\sigma_1$ the state after the response step of $\pi_1$ and $\sigma_2$ the state after the response step of $\pi_2$. Then the following properties hold:* (*a*) ***Configuration Consistency****:* $\mathbf{c}^{p_2}_{\sigma_2}[i].cfg = \mathbf{c}^{p_1}_{\sigma_1}[i].cfg$, *for* $1 \leq i \leq |\mathbf{c}^{p_1}_{\sigma_1}|$, (*b*) ***Seq. Prefix****:* $\mathbf{c}^{p_1}_{\sigma_1} \preceq_p \mathbf{c}^{p_2}_{\sigma_2}$, *and* (*c*) ***Seq. Progress****:* $\mu(\mathbf{c}^{p_1}_{\sigma_1}) \leq \mu(\mathbf{c}^{p_2}_{\sigma_2})$

Given the properties satisfied by the reconfiguration algorithm of ARES and assuming that the DAP used satisfy Property 1, as presented in Section 3, then we have the following result.

THEOREM 24 (ATOMICITY). *In any execution $\xi$ of* ARES*, if in every configuration $c \in \mathcal{G}_L$, $c$.get-data(), $c$.put-data(), and $c$.get-tag() satisfy Property 1, then* ARES *satisfy atomicity.*

REMARK 25. *Algorithm* ARES *satisfies atomicity even when the implementaton of the DAPs in two different configurations $c_1$ and $c_2$ are not the same, given that the $c_i$.get-tag, $c_i$.get-data, and the $c_i$.put-data primitives in each $c_i$ satisfy Property 1.*

## 7 PERFORMANCE ANALYSIS OF ARES

In this section we provide an analysis of the storage and communication costs of ARES, and the latency of read and write operations.

### 7.1 Storage and Communication Costs for ARES.

We now briefly present the storage and communication costs associated with the presented DAPs. Recall that by our assumption, the storage cost counts the size (in bits) of the coded elements stored in variable *List* at each server. We ignore the storage cost due to meta-data. For communication cost we measure the bits sent on the wire between the nodes.

THEOREM 26. *The* ARES *algorithm has: (i) storage cost $(\delta + 1)\frac{n}{k}$, (ii) communication cost for each write at most to $\frac{n}{k}$, and (iii) communication cost for each read at most $(\delta + 2)\frac{n}{k}$.*

### 7.2 Latency Analysis for read and writes in ARES

Liveness properties cannot be specified for ARES, without restricting asynchrony or the rate of arrival of reconfig operations, or if the consensus protocol never terminates. Here, we provide some conditional performance analysis of the operation, based on latency bounds on the message delivery. We assume that local computations take negligible time and the latency of an operation is due to the delays in the messages exchanged during the execution. We measure delays in *time units* of some global clock, which is visible only to an external viewer. No process has access to the clock. Let $d$ and $D$ be the minimum and maximum durations taken by messages, sent during an execution of ARES, to reach their destinations. Also, let $T(\pi)$ denote the duration of an operation (or action) $\pi$. In the statements that follow, we consider any execution $\xi$ of ARES, which contains $k$ reconfig operations. For any configuration $c$ in an execution of ARES, we assume that any $c.Con$.propose operation, takes at least $T_{min}(CN)$ time units.

The following lemma shows the maximum latency of a read or a write operation, invoked by any non-faulty client. From ARES algorithm, the latency of a read/write operation depends on the delays of the DAPs operations. For our analysis we assume that all get-data, get-tag and put-data primitives use two phases of communication. Each phase consists of a communication between the client and the servers.

LEMMA 27. *Suppose $\pi$, $\phi$ and $\psi$ are operations of the type* put-data*,* get-tag *and* get-data*, respectively, invoked by some non-faulty reconfiguration clients, then the latency of these operations are bounded as follows: $(i)$ $2d \leq T(\pi) \leq 2D$; $(ii)$ $2d \leq T(\phi) \leq 2D$; and $(iii)$ $2d \leq T(\psi) \leq 2D$.*

In the following lemma, we estimate the time taken for a read or a write operation to complete, when it discovers $k$ configurations between its invocation and response steps.

LEMMA 28. *Consider any execution of* ARES *where at most k reconfiguration operations are invoked. Let $\sigma_s$ and $\sigma_e$ be the states before the invocation and after the completion step of a read/write operation $\pi$, in some fair execution $\xi$ of* ARES*. Then we have $T(\pi) \leq 6D\,(k + 2)$ to complete.*

It remains now to examine the conditions under which a read/write operation may âĂIJcatch upâĂİ with an infinite number of reconfiguration operations. For the sake of a worst case analysis

we will assume that reconfiguration operations suffer the minimum delay $d$, whereas read and write operations suffer the maximum delay $D$ in each message exchange. Also, we assume that any consensus operation takes the least amount of time to complete $T_{min}(CN)$. The following theorem is the main result of this section, in which we define the relation between $T_{min}(CN)$, $d$ and $D$ so to guarantee that any read or write issued by a non-faulty client always terminates.

THEOREM 29. *Suppose $T_{min}(CN) \geq 3(6D - d)$, then any read or write operation $\pi$ completes in any execution $\xi$ of* ARES *for any number of reconfiguration operations in $\xi$.*

## 8 FLEXIBILITY OF DAPS

In this section, we argue that various implementations of DAPs can be used in ARES. In fact, via reconfig operations, one can implement a highly adaptive atomic DSS: replication-based can be transformed into erasure-code based DSS; increase or decrease the number of storage servers; study the performance of the DSS under various code parameters, etc. The insight to implementing various DAPs comes from the observation that the simple algorithmic template $A$ (see Alg. 7) for reads and writes protocol combined with any implementation of DAPs, satisfying Property 1 gives rise to a MWMR atomic memory service. Moreover, the read and writes operations terminate as long as the implemented DAPs complete.

**Algorithm 7** Template $A$ for the client-side read/write steps.

```
   operation read()
2:   ⟨t, v⟩ ← c.get-data()
     c.put-data(⟨t, v⟩)
4:   return ⟨t, v⟩
   end operation

6: operation write(v)
     t ← c.get-tag()
8:   t_w ← inc(t)
     c.put-data(⟨t_w, v⟩)
10: end operation
```

A read operation in $A$ performs $c$.get-data() to retrieve a tag-value pair, $\langle \tau, v \rangle$ from a configuration $c$, and then it performs a $c$.put-data($\langle \tau, v \rangle$) to propagate that pair to the configuration $c$. A write operation is similar to the read but before performing the put-data action it generates a new tag which associates with the value to be written. The following result shows that $A$ is atomic and live, if the DAPs satisfy Property 1 and live.

THEOREM 30 (ATOMICITY OF TEMPLATE $A$). *Suppose the DAP implementation satisfies the consistency properties $C1$ and $C2$ of Property 1 for a configuration $c \in \mathcal{C}$. Then any execution $\xi$ of algorithm $A$ in configuration $c$ is atomic and live if each DAP invocation terminates in $\xi$ under the failure model $c.\mathcal{F}$.*

A number of known tag-based algorithms that implement atomic read/write objects (e.g., ABD [6], FAST[15] – see [34]), can be expressed in terms of DAP.

## 9 CONCLUSIONS

We presented an algorithmic framework suitable for reconfigurable, erasure code-based atomic memory service in asynchronous, message-passing environments. Future work will involve adding efficient repair and reconfiguration using regenerating codes.

## REFERENCES

[1] Intel storage acceleration library (open source version). https://goo.gl/zkVl4N.

[2] ABEBE, M., DAUDJEE, K., GLASBERGEN, B., AND TIAN, Y. Ec-store: Bridging the gap between storage and latency in distributed erasure coded systems. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)* (July 2018), pp. 255–266.

[3] AGUILERA, M. K., KEIDAR, I., MALKHI, D., AND SHRAER, A. Dynamic atomic storage without consensus. In *Proceedings of the 28th ACM symposium on Principles of distributed computing (PODC '09)* (New York, NY, USA, 2009), ACM, pp. 17–25.

[4] AGUILERA, M. K., KEIDARY, I., MALKHI, D., MARTIN, J.-P., AND SHRAERY, A. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS 102* (2010), 84–081.

[5] ANTA, A. F., NICOLAOU, N., AND POPA, A. Making "fast" atomic operations computationally tractable. In *International Conference on Principles Of Distributed Systems* (2015), OPODIS'15.

[6] ATTIYA, H., BAR-NOY, A., AND DOLEV, D. Sharing memory robustly in message passing systems. *Journal of the ACM 42(1)* (1996), 124–142.

[7] BURIHABWA, D., FELBER, P., MERCIER, H., AND SCHIAVONI, V. A performance evaluation of erasure coding libraries for cloud-based data stores. In *Distributed Applications and Interoperable Systems* (2016), Springer, pp. 160–173.

[8] CACHIN, C., AND TESSARO, S. Optimal resilience for erasure-coded byzantine distributed storage. IEEE Computer Society, pp. 115–124.

[9] CADAMBE, V. R., LYNCH, N., MÉDARD, M., AND MUSIAL, P. A coded shared atomic memory algorithm for message passing architectures. In *Network Computing and Applications (NCA), 2014 IEEE 13th International Symposium on* (Aug 2014), pp. 253–260.

[10] CADAMBE, V. R., LYNCH, N. A., MÉDARD, M., AND MUSIAL, P. M. A coded shared atomic memory algorithm for message passing architectures. *Distributed Computing 30*, 1 (2017), 49–73.

[11] CHEN, Y. L. C., MU, S., AND LI, J. Giza: Erasure coding objects across global data centers. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC âĂŹ17)* (2017), pp. 539–551.

[12] CHOCKLER, G., GILBERT, S., GRAMOLI, V., MUSIAL, P. M., AND SHVARTSMAN, A. A. Reconfigurable distributed storage for dynamic networks. *Journal of Parallel and Distributed Computing 69*, 1 (2009), 100–116.

[13] CHOCKLER, G., AND MALKHI, D. Active disk paxos with infinitely many processes. *Distributed Computing 18*, 1 (2005), 73–84.

[14] DUTTA, P., GUERRAOUI, R., AND LEVY, R. R. Optimistic erasure-coded distributed storage. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 182–196.

[15] DUTTA, P., GUERRAOUI, R., LEVY, R. R., AND CHAKRABORTY, A. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC)* (2004), pp. 236–245.

[16] FAN, R., AND LYNCH, N. Efficient replication of large data objects. In *Distributed algorithms* (2003), F. E. Fich, Ed., vol. 2848 of *Lecture Notes in Computer Science*, pp. 75–91.

[17] FERNÁNDEZ ANTA, A., HADJISTASI, T., AND NICOLAOU, N. Computationally light "multi-speed" atomic memory. In *International Conference on Principles Of Distributed Systems* (2016), OPODIS'16.

[18] GAFNI, E., AND MALKHI, D. Elastic Configuration Maintenance via a Parsimonious Speculating Snapshot Solution. In *International Symposium on Distributed Computing* (2015), Springer, pp. 140–153.

[19] GEORGIOU, C., NICOLAOU, N. C., AND SHVARTSMAN, A. A. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 289–304.

[20] GEORGIOU, C., NICOLAOU, N. C., AND SHVARTSMAN, A. A. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel and Distributed Computing 69*, 1 (2009), 62–79.

[21] GILBERT, S. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. Master's thesis, MIT, August 2003.

[22] GILBERT, S., LYNCH, N., AND SHVARTSMAN, A. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)* (2003), pp. 259–268.

[23] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12*, 3 (1990), 463–492.

[24] HUFFMAN, W. C., AND PLESS, V. *Fundamentals of error-correcting codes*. Cambridge university press, 2003.

[25] JEHL, L., VITENBERG, R., AND MELING, H. Smartmerge: A new approach to reconfiguration for atomic storage. In *International Symposium on Distributed Computing* (2015), Springer, pp. 154–169.

[26] JOSHI, G., SOLJANIN, E., AND WORNELL, G. Efficient redundancy techniques for latency reduction in cloud systems. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS) 2*, 2 (2017), 12.

[27] KONWAR, K. M., PRAKASH, N., KANTOR, E., LYNCH, N., MÉDARD, M., AND SCHWARZMANN, A. A. Storage-optimized data-atomic algorithms for handling erasures and errors in distributed storage systems. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2016), pp. 720–729.

[28] KONWAR, K. M., PRAKASH, N., LYNCH, N., AND MÉDARD, M. Radon: Repairable atomic data object in networks. In *The International Conference on Distributed Systems (OPODIS)* (2016).

[29] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (1998), 133–169.

[30] LYNCH, N. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[31] LYNCH, N., AND SHVARTSMAN, A. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of 16th International Symposium on Distributed Computing (DISC)* (2002), pp. 173–190.

[32] LYNCH, N. A., AND SHVARTSMAN, A. A. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of Symposium on Fault-Tolerant Computing* (1997), pp. 272–281.

[33] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system.

[34] NICOLAOU, N., CADAMBE, V., KONWAR, K., PRAKASH, N., LYNCH, N., AND MÉDARD, M. Ares: Adaptive, reconfigurable, erasure coded, atomic storage. *CoRR abs/1805.03727* (2018).

[35] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 305–320.

[36] RASHMI, K., CHOWDHURY, M., KOSAIAN, J., STOICA, I., AND RAMCHANDRAN, K. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *OSDI* (2016), pp. 401–417.

[37] SHRAER, A., MARTIN, J.-P., MALKHI, D., AND KEIDAR, I. Data-centric reconfiguration with network-attached disks. In *Proceedings of the 4th Int'l Workshop on Large Scale Distributed Sys. and Middleware (LADIS âĂŹ10)* (2010), p. 22âĂŞ26.

[38] SPIEGELMAN, A., KEIDAR, I., AND MALKHI, D. Dynamic Reconfiguration: Abstraction and Optimal Asynchronous Solution. In *31st International Symposium on Distributed Computing (DISC 2017)* (2017), vol. 91, pp. 40:1–40:15.

[39] WANG, S., HUANG, J., QIN, X., CAO, Q., AND XIE, C. Wps: A workload-aware placement scheme for erasure-coded in-memory stores. In *Networking, Architecture, and Storage (NAS), 2017 International Conference on* (2017), IEEE, pp. 1–10.

[40] XIANG, Y., LAN, T., AGGARWAL, V., AND CHEN, Y.-F. R. Multi-tenant latency optimization in erasure-coded storage with differentiated services. In *2015 IEEE 35th International Conference on Distributed Computing Systems (ICDCS)* (2015), IEEE, pp. 790–791.

[41] XIANG, Y., LAN, T., AGGARWAL, V., CHEN, Y.-F. R., XIANG, Y., LAN, T., AGGARWAL, V., AND CHEN, Y.-F. R. Joint latency and cost optimization for erasure-coded data center storage. *IEEE/ACM Transactions on Networking (TON) 24*, 4 (2016), 2443–2457.

[42] YU, Y., HUANG, R., WANG, W., ZHANG, J., AND LETAIEF, K. B. Sp-cache: load-balanced, redundancy-free cluster caching with selective partition. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (2018), IEEE Press, p. 1.

[43] ZHANG, H., DONG, M., AND CHEN, H. Efficient and available in-memory kv-store with hybrid erasure coding and replication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 167–180.

[44] ZHOU, P., HUANG, J., QIN, X., AND XIE, C. Pars: A popularity-aware redundancy scheme for in-memory stores. *IEEE Transactions on Computers* (2018), 1–1.