ARES: Adaptive, Reconfigurable, Erasure coded, atomic Storage

Nicolas Nicolaou\textsuperscript{1}, Viveck Cadambe\textsuperscript{2}, N. Prakash\textsuperscript{3}, Kishori M. Konwar\textsuperscript{3}, Muriel Medard\textsuperscript{3} and Nancy Lynch\textsuperscript{3}

\textsuperscript{1}Algorysis Ltd. \& KIOS Research and Innovation Center of Excellence, Univ. of Cyprus, Nicosia, Cyprus
\textsuperscript{2}EE Department Penn. State University University Park, PA, US
\textsuperscript{3}Department of EECS, Massachusetts Institute of Technology, Cambridge MA, USA
nicolasn@cs.ucy.ac.cy, vxc12@engr.psu.edu, prakashn@mit.edu, kishori@csail.mit.edu, medard@mit.edu, lynch@csail.mit.edu

Abstract—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.

Index Terms—atomic storage; erasure codes; fault tolerance;

I. 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 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], [16], [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 \(\frac{k}{2}\) 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 [7], [1], [36]. In fact, an exciting recent body of systems and optimization works [44], [2], [36], [39], [41], [26], [40], [42] 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.
As the set of storage servers becomes older and unreliable, wide range of purposes, especially during system maintenance, are replaced with new ones to ensure data-durability. 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 [44], [2], [36], [39], [41], [26], [40], [42] and non-blocking erasure coded based consistent storage [8], [10], [14], [27], [28], [11], [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 atomic 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 [31], [3], [12], [18], [21], [25], [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.

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

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.

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, 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 I 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 communication 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 II, presents the model assumptions and Section III, the DAP primitives. In Section IV, we present the implementation of the reconfiguration and read/write protocols in ARES using the DAPs. In Section V, we present an erasure-coded implementation of a set of DAPs, which can be used in every configuration of the ARES algorithm. Section VI provides operation latency and cost analysis, and Section VII the DAP flexibility. We conclude our work in Section VIII.
TABLE I: 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].

Due to lack of space omitted proofs can be found in [34].

II. 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 \( V \). We assume a system consisting of four distinct sets of processes: a set \( W \) of writers, a set \( R \) of readers, a set \( G \) of reconfiguration clients, and a set \( S \) of servers.

Let \( I = W \cup R \cup G \) 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 defines 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 III). More formally, a configuration, \( c \in C \), consists of: (i) \( c.Servers \subseteq S \): 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 \( I \) 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 \( p \in I \cup S \). 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 \( c.F \) to describe the failure model of a configuration \( c \).

**Reconfigurable Atomic Read/Write Objects.** A reconfigurable atomic object supports three operations: read(), write() and reconfig(). A read() operation returns the value of the atomic object, write() attempts to modify the value of the object to \( v \in V \), and the reconfig() 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 \( \pi \rightarrow \pi' \), \( \pi \prec \pi' \), or \( \pi' \prec \pi \).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>#rounds /write</th>
<th>#rounds /read</th>
<th>Reconfig.</th>
<th>Repl. or EC</th>
<th>Storage cost</th>
<th>read bandwidth</th>
<th>write bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASGC [9]</td>
<td>3</td>
<td>2</td>
<td>No</td>
<td>EC</td>
<td>( (\delta + 1) \frac{n}{\delta} )</td>
<td>( \frac{n}{\delta} )</td>
<td>( \frac{n}{\delta} )</td>
</tr>
<tr>
<td>SODA [27]</td>
<td>2</td>
<td>2</td>
<td>No</td>
<td>EC</td>
<td>( \frac{n}{\delta} )</td>
<td>( (\delta + 1) \frac{n}{\delta} )</td>
<td>( \frac{n}{\delta} )</td>
</tr>
<tr>
<td>ORCAS-A [14]</td>
<td>( \geq 2 )</td>
<td></td>
<td>No</td>
<td>EC</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>ORCAS-B [14]</td>
<td>3</td>
<td>3</td>
<td>No</td>
<td>EC</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
</tr>
<tr>
<td>ABD [6]</td>
<td>2</td>
<td>2</td>
<td>No</td>
<td>Repl.</td>
<td>n</td>
<td>2n</td>
<td>n</td>
</tr>
<tr>
<td>RAMBO [31]</td>
<td>2</td>
<td>2</td>
<td>Yes</td>
<td>Repl.</td>
<td>( \geq n )</td>
<td>( \geq n )</td>
<td>( \geq n )</td>
</tr>
<tr>
<td>DYNASTORE [3]</td>
<td>( \geq 4 )</td>
<td>( \geq 4 )</td>
<td>Yes</td>
<td>Repl.</td>
<td>( \geq n )</td>
<td>( \geq n )</td>
<td>( \geq n )</td>
</tr>
<tr>
<td>SMARTMERGE [25]</td>
<td>2</td>
<td>2</td>
<td>Yes</td>
<td>Repl.</td>
<td>( \geq n )</td>
<td>( \geq n )</td>
<td>( \geq n )</td>
</tr>
<tr>
<td>ARES (this paper)</td>
<td>2</td>
<td>2</td>
<td>Yes</td>
<td>EC</td>
<td>( (\delta + 1) \frac{n}{\delta} )</td>
<td>( (\delta + 1) \frac{n}{\delta} )</td>
<td>( \frac{n}{\delta} )</td>
</tr>
</tbody>
</table>
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 recovered to 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 $e_i$ 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 $i$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 $\tau_2.w > \tau_1.w$.

III. 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($⟨\tau, v⟩$), via which a client can ingest the tag value $v$ in the configuration $c$; (ii) $c$.get-data($⟨\tau, v⟩$), used to retrieve the most up to date tag and value pair stored in the configuration $c$; and (iii) $c$.get-tag($⟨\tau⟩$), 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($⟨\tau⟩$) that returns a tag $\tau \in \mathcal{T}$;
$D2$: $c$.get-data($⟨\tau, v⟩$) that returns a tag-value pair $(\tau, v) \in \mathcal{T} \times \mathcal{V}$;
$D3$: $c$.put-data($⟨\tau, v⟩$) 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 VI-B, 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($⟨\tau_\phi, v_\phi⟩$), for $c \in \mathcal{C}$, $(\tau_\phi, v_\phi) \in \mathcal{T} \times \mathcal{V}$, and $\pi$ is $c$.get-tag($⟨\tau⟩$) or $c$.get-data($⟨\tau⟩$) that returns $(\tau, v)$ in $\mathcal{T}$ ($or (\tau, v) \in \mathcal{T} \times \mathcal{V}$) and $\phi$ completes before $\pi$ is invoked in $\xi$, then $\tau \geq \tau_\phi$.

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

In Section V 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 VII, 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.
IV. 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 III. 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.

A. 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 form of a “distributed list”, which we refer to as the global configuration sequence (or list) \( G_L \). Conceptually, \( G_L \) represents an ordered list of pairs \((c, \text{status})\), where \( c \) is a configuration identifier \((c \in C)\), and a binary state variable \( \text{status} \in \{F, P\} \) that denotes whether \( c \) is finalized \((F)\) or is still pending \((P)\). Initially, \( G_L \) contains a single element, say \((c_0, F)\), which is known to every participant in the service.

To facilitate the creation of \( G_L \), each server in \( c.Servers \) maintains a local variable \( nextC \in \{C \cup \{\perp\}\} \times \{P, F\} \), which is used to point to the configuration that follows \( c \) in \( G_L \). Initially, at any server \( nextC = (\perp, F) \). 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 \( \perp \) agree, i.e., \( \{s.nextC : s \in c.Servers \land s.nextC \neq \perp\} \) is either empty of has only one element.

Clients discover the configuration that follows a \((c, \ast)\) in the sequence by contacting a subset of servers in \( c.Servers \) and collecting their \( nextC \) variables. Every client in \( T \) maintains a local variable \( cseq \) that is expected to be some subsequence of \( G_L \). Initially, at every client the value of \( cseq \) is \((c_0, F) \). We use the notation \( \hat{x} \) (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 \( G_L \). 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 \( G_L \), and (ii) reconfiguration operation that installs new configurations in \( G_L \). 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 \((c, F)\) 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 \( G_L \). 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', 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 \( G_L \). At invocation, the client starts with the last finalized configuration \((c, F)\) in the given \( seq \) (Line A1:2), and enters a loop to traverse \( G_L \) by invoking get-next-config(c), which returns the next configuration, say \( c' \). While \( c' \neq \perp \), then: (a) \( c' \) is appended at the end of the sequence \( seq \); (b) a put-config(c', c') is invoked to inform a quorum of servers in \( c.Servers \) to update the value of their \( nextC \) variable to \( c' \); and (c) variable \( c \) is set to \( c'.cfg \). If \( c' = \perp \), the loop terminates and the action read-config returns \( seq \).

Reconfiguration Operation. A reconfiguration operation \( \text{reconf}(c), c \in C \), invoked by any reconfiguration client \( \text{rec}_i \), attempts to append \( c \) to \( G_L \). The set of server processes in \( c \) are not a part of any other configuration different from \( c \). In a high level, \( \text{rec}_i \) first executes a sequence traversal to discover the latest state of \( G_L \). 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 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, \( \text{rec}_i \) 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 \( \text{rec}_i \) (see Alg. 2):

- read-config(seq): The phase read-config(seq) at \( \text{rec}_i \), 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 \( G_L \)). Suppose the last configuration in \( seq \) is \( c' \)

Reconfiguration protocol of algorithm ARES.

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

```
procedure read-config(seq)
2: $\mu = \max\{j : seq[j].status = F\}$
3: $c \leftarrow seq[\mu]$
4: while $c \neq \bot$ do
5: $c' \leftarrow$-get-next-config($c$.cfg)
6: if $c' \neq \bot$ then
7: $\mu \leftarrow \mu + 1$
8: $seq[\mu] \leftarrow c'$
9: put-config($seq[\mu - 1]$.cfg, seq[\mu])
10: $c \leftarrow seq[\mu]$
end while
12: return seq
end procedure
```

Algorithm 2 Reconfiguration protocol of algorithm ARES.

```
procedure add-config(seq, c)
14: $\nu \leftarrow |seq|$
15: $c' \leftarrow seq[\nu]$.cfg
16: $d \leftarrow c'.Con.propose(c)$
17: $seq[\nu + 1] \leftarrow (d, P)$
18: put-config($c', (d, P)$)
end procedure
```

Procedure finalize-config(seq)
```
20: return seq
end procedure
```

Procedure update-config(seq)
```
22: $\mu \leftarrow \max\{j : seq[j].status = F\}$
23: $\nu \leftarrow |seq|$
24: $M \leftarrow \emptyset$
25: for $i = \mu : \nu$ do
26: $(t, v) \leftarrow seq[i]$.cfg.get-data()
27: $M \leftarrow M \cup \{(t, v)\}$
28: $(\tau, v) \leftarrow \max\{(t, v) : (t, v) \in M\}$
29: seq[\nu].put-data((\tau, v))
end for
end procedure
```

Procedure get-next-config(c)
```
14: procedure get-next-config(c)
16: if $\exists s \in \mathcal{Q}$ s.t. $rec_i$ receives $nextC_s$ from $\forall s \in \mathcal{Q}$
18: return $nextC_s$
else if $\exists s \in \mathcal{Q}$ s.t. $nextC_s.status = P$
20: return $nextC_s$
else
22: return $\bot$
end procedure
```

Algorithm 3 Server protocol of algorithm ARES.

```
8: end receive
```

Upon receive (WRITE-CONFIG, $cfgT_{in}$)$_{s_i,c_k}$ from $q$
```
10: if $nextC_c.cfg = \bot \lor nextC_c.status = P$
then
11: $nextC_c \leftarrow cfgT_{in}$
end if
```

At each server $s_i$ in configuration $c_k$
```
12: send ACK to $q$
end receive
```

(update-conf $(seq)$): 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 $cseq[i]$ 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 $(t_{max}, v_{max})$ is the tag value pair corresponding to the highest

$\mu$. On receiving $\nu$, it then invokes $read-config$ at $\nu$, and adds the configuration decided by $\mu$ to the local sequence $cseq$. If $\mu + 1 = \nu$, the reconfigurer $rec_i$ returns the extended configuration $seq$. If $\mu + 1 = \nu$, the reconfigurer $rec_i$ continues to execute Algorithm 1, but ignores the configuration $c$. If $\mu + 1 = \nu$, the reconfigurer $rec_i$ continues to execute Algorithm 1, but ignores the configuration $c$. If $\mu + 1 = \nu$, the reconfigurer $rec_i$ continues to execute Algorithm 1, but ignores the configuration $c$.
Then, \( \text{cfgT} \) is written to the configuration \( d \) via the invocation of \( \text{cseq} \), put-data(\( \langle \text{cfgT}, \text{v}\rangle \)).

**Implementation of Read and Write operations.**

For each configuration, and discovered configuration sequence, for the latest starting from the last finalized configuration to the end of the configuration sequence in \( G_L \), in both operations.

A write (or read) operation is invoked at a client \( p \) when a read-config action to obtain the latest introduced configuration in \( G_L \), in both operations.

If \( \pi \) is a write \( p \) detects the last finalized entry in \( \text{cseq} \), say \( \mu \), and performs a \( \text{cseq}[\mu].\text{conf}.\text{get-tag}() \) action, for \( \mu \leq j \leq |\text{cseq}| \) (line Alg. 4.9). Then \( p \) discovers the maximum tag among all the returned tags \( \langle \text{cseq}, \text{v}\rangle \), and it increments the maximum tag discovered (by incrementing the integer part of \( \text{cseq} \)), generating a new tag, say \( \tau \). It assigns \( \langle \tau, \text{v}\rangle \) to \( \langle \text{cseq}, \text{v}\rangle \) (Line Alg. 4.13).

If \( \pi \) is a read, \( p \) detects the last finalized entry in \( \text{cseq} \), say \( \nu \), and performs a \( \text{cseq}[\nu].\text{conf}.\text{get-data}() \) action, for \( \mu \leq j \leq |\text{cseq}| \) (line Alg. 4.32). Then \( p \) discovers the maximum tag-value pair \( \langle \text{cseq}[\nu].\text{v}, \text{v}\rangle \) among the replies, and assigns \( \langle \tau, \text{v}\rangle \) to \( \langle \text{cseq}[\nu].\text{v}, \text{v}\rangle \).

Once specifying the \( \langle \tau, \text{v}\rangle \) to be propagated, both reads and writes execute the \( \text{cseq}[\nu].\text{conf}.\text{put-data}() \) action, where \( \nu = \langle \text{cseq}, \text{v}\rangle \), followed by executing read-config action, to examine whether new configurations were introduced in \( G_L \). The repeat these steps until no new configuration is discovered (lines Alg. 4.15–21, or lines Alg. 4.37–43). Let \( \text{cseq}' \) be the sequence returned by the read-config action. If \( \text{cseq}' = |\text{cseq}| \) then no new configuration is introduced, and the read/write operation terminates; otherwise, \( p \) sets \( \text{cseq} \) to \( \text{cseq}' \) and repeats the two actions. Note, in an execution of \( \text{ARES} \), two consecutive read-config operations that return \( \text{cseq}' \) and \( \text{cseq}'' \) respectively must hold that \( \text{cseq}'' \) is a prefix of \( \text{cseq}' \), and hence \( |\text{cseq}''| = |\text{cseq}'| \) only if \( \text{cseq}'' = \text{cseq}' \). Finally, if \( \pi \) is a read operation the value with the highest tag discovered is returned to the client.

**Discussion** \( \text{ARES} \) shares similarities with previous algorithms like RAMO [22] and the framework in [38]. The reconfiguration technique used in \( \text{ARES} \) ensures the prefix property on the configuration sequence (resembling a blockchain data structure [33]) while the array structure in RAMO 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].

**V. 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 \( i = 1, \ldots, n \). The implementations of DAP primitives used in \( \text{ARES} \) are shown in Alg. 5, and the servers’ responses in Alg. 6.

Each server \( s_i \) stores one state variable, \( \text{List} \), which is a set of up to \((\delta + 1) \) (tag, coded-element) pairs. Initially the set at \( s_i \) contains a single element, \( \text{List} = \{(t_0, \Phi_1(t_0))\} \). Below we describe the implementation of the DAPs.
Algorithm 4 Write and Read protocols at the clients for ARES.

Write Operation:
2: at each writer $w_i$  
State Variables:  
4: $cseq[[s..t..cseq[j]]] \in C \times \{F, P\}$ with members:  
Initialization:  
6: $cseq[0] = (c_0, F)$

operation write$(val), \ val \in V$
8: $cseq \leftarrow read-config(cseq)$  
10: $\nu \leftarrow \max\{i : cseq[i].status = F\}$  
for $i = \mu : \nu$ do
12: $\tau_{max} \leftarrow \max(cseq[i].cfg.get-tag(), \tau_{max})$  
14: $\langle \tau, \nu \rangle \leftarrow (\langle \tau_{max}.ts + 1, \omega_i, \nu \rangle, val)$
while not done do
16: $cseq[v].cfg.put-data(\langle \tau, \nu \rangle)$  
18: if $|cseq| = \nu$ then
20: end while
else
22: $\nu \leftarrow |cseq|$  
end while
end operation

Read Operation:
24: at each reader $r_i$
26: State Variables:  
28: Initialization:  
30: operation read()  
32: $cseq \leftarrow read-config(cseq)$
34: for $i = \mu : \nu$ do
36: $\nu \leftarrow \max(cseq[i].cfg.get-data(), \langle \tau, \nu \rangle)$
while not done do
38: $cseq[v].cfg.put-data(\langle \tau, \nu \rangle)$  
40: if $|cseq| = \nu$ then
42: end while
else
44: $\nu \leftarrow |cseq|$  
end while
46: end operation

Algorithm 5 DAP implementation for ARES.

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

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

8: procedure c.get-data()
10: until $p_i$ receives $List_s$ from each server $s \in S_g$ s.t. $|S_g| \geq \lceil \frac{n + k}{2} \rceil$ and $S_g \subseteq c.Servers$
end procedure

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

at each server $s_i \in S$ in configuration $c_k$

2: State Variables:  
List $\subseteq T \times C_\Sigma$, initially $\{(t_0, \Phi_\Sigma(t_0))\}$  
Upon receive (QUERY-TAG) $s_i, c_k$ from $q$
4: $\tau_{max} = \max_{(t, e) \in List} t$
5: Send $\tau_{max}$ to $q$
6: end receive

Upon receive (QUERY-LIST) $s_i, c_k$ from $q$
8: Send $List$ to $q$
10: end receive

Upon receive (PUT-DATA, $\langle \tau, e_i \rangle, s_i, c_k$) from $q$
12: $List \leftarrow List \cup \{(\tau, e_i)\}$
13: if $|List| > \delta + 1$ then
14: $\tau_{min} \leftarrow \min\{t : (t, e) \in List\}$
15: $List \leftarrow List \setminus \{(\tau, e) : \tau = \tau_{min} \wedge (\tau, e) \in List\}$
16: $List \leftarrow List \cup \{(\tau_{min}, \bot)\}$
17: Send ACK to $q$
18: end receive

Algorithm 7 DAP implementation for ARES.

\begin{align*}
c\text{-get-tag}(): & \text{ A client, during the execution of a } c\text{-get-tag()} \text{ primitive, queries all the servers in } c.Servers \text{ for the highest tags in their Lists, and awaits responses from } \lceil \frac{n + k}{2} \rceil \text{ servers. A server upon receiving the GET-TAG request, responds to the client with the highest tag, as } \tau_{max} \equiv \max_{(t, e) \in List} t. \text{ Once the client receives the tags from } \lceil \frac{n + k}{2} \rceil \text{ servers, it selects the highest tag } t \text{ and returns it.} \\
c\text{-put-data}((t_w, v))(): & \text{ During the execution of the primitive}
\end{align*}
c.put-data((t_w, v)), a client sends the pair (t_w, \Phi(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 \(*\), 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.

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 \(\left\lceil \frac{n + k}{2} \right\rceil\) servers. Once the client receives Lists from \(\left\lceil \frac{n + k}{2} \right\rceil\) 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.

VI. CORRECTNESS, PERFORMANCE AND LATENCY OF ARES

In this section, we prove the atomicity property of ARES. We also provide an analysis of its storage and communication costs, and the latency of read and write operations. The atomicity property of ARES hinges on the Property 1 of the DAP implementation in each individual configuration used in ARES. Therefore, we start by proving, in subsection VI-A, that Property 1 holds for the DAP implementation in Section V. Based on this, in subsection VI-B, we prove the atomicity of ARES. Next, in sub-section VI-C, we derive the storage and communication costs of read and write operations, and in subsection VI-D, 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. Due to lack of space proofs are omitted and can be found in the extended version of the paper [34].

A. 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 2 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 \(\Lambda = \{\lambda : \lambda > tag(\phi^*)\}\). 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 than \(\delta\) 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 \(\pi\).

B. Atomicity Property of ARES

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

Theorem 6. Let \(\pi_1\) and \(\pi_2\) two completed read-config actions invoked by processes \(p_1, p_2 \in \Pi\) 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: \(c_{\pi_2}^p[i], c_{\pi_2}^p[i] = c_{\pi_1}^p[i], c_{\pi_1}^p[i], for 1 \leq i \leq \mu(c_{\sigma_1}^p)\), (b) Seq. Prefix: \(c_{\pi_1}^p \subseteq c_{\pi_2}^p\), and (c) Seq. Progress: \(\mu(c_{\pi_1}^p) \leq \mu(c_{\pi_2}^p)\).

Given the properties satisfied by the reconfiguration algorithm of ARES and assuming that the DAP used satisfy Property 1, as presented in Section III, then we have the following result.
Theorem 7 (Atomicity). In any execution \( \xi \) of ARES, if in every configuration \( c \in G_L \), \( c.\text{get-data()} \), \( c.\text{put-data()} \), and \( c.\text{get-tag()} \) satisfy Property 1, then ARES satisfy atomicity.

Remark 8. Algorithm ARES satisfies atomicity even when the implementation of the DAPs in two different configurations \( c_1 \) and \( c_2 \) are not the same, given that the \( c_1.\text{get-tag} \), \( c_1.\text{get-data} \), and \( c_1.\text{put-data} \) primitives in each \( c_i \) satisfy Property 1.

C. 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 9. The ARES algorithm has: (i) storage cost \((\delta+1)\frac{n}{\pi}\), (ii) communication cost for each write at most to \(n\frac{2}{\pi}\), and (iii) communication cost for each read at most \((\delta+2)\frac{n}{\pi}\).

D. 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, consider any execution \( \xi \) of ARES, which contains \( k \) reconfig operations. For any configuration \( c \) in an execution of ARES, we assume that any \( c.\text{Con} \) 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 10. 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 11. Consider any execution of ARES where at most \( k \) reconfiguration operations are invoked. Let \( \sigma_1 \) and \( \sigma_2 \) 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 catch up 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 12. 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 \( \xi \).

VII. 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 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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( (t, v) \leftarrow c.\text{get-data}() )</td>
<td>6: ( t_w \leftarrow \text{inc}(t) )</td>
</tr>
<tr>
<td>2: ( c.\text{put-data}((t, v)) )</td>
<td>8: ( c.\text{get-tag}() )</td>
</tr>
<tr>
<td>4: ( \text{return } (t, v) )</td>
<td>10: ( \text{end operation} )</td>
</tr>
</tbody>
</table>

A read operation in \( A \) performs \( c.\text{get-data}() \) to retrieve a tag-value pair, \( (\tau, v) \) from a configuration \( c \), and then it performs \( c.\text{put-data}((\tau, v)) \) 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 13 (Atomicity of template A). Suppose the DAP implementation satisfies the consistency properties C1 and C2 of Property 1 for a configuration \( c \in C \). 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.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.

VIII. 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.