# ARES: Adaptive, Reconfigurable, Erasure coded, Atomic Storage *

NICOLAS NICOLAOU, Algolysis Ltd, Limassol, Cyprus
VIVECK CADAMBE, Pennsylvania State University, US
N. PRAKASH, Intel Corp.
ANDRIA TRIGEORGI, University of Cyprus, Nicosia, Cyprus
KISHORI M. KONWAR, MURIEL MEDARD, and NANCY LYNCH, Massachusetts Institute of Technology, 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., [11]) 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, Andria Trigeorgi, Kishori M. Konwar, Muriel Medard, and Nancy Lynch. 2022. ARES: Adaptive, Reconfigurable, Erasure coded, Atomic Storage . 1, 1 (January 2022), 39 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Distributed Storage Systems (DSSes) 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

*A preliminary version of this work appeared in ICDCS'19 [44]

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.eduPennsylvania State University, US; N. Prakash, prakashn@mit.eduIntel Corp.; Andria Trigeorgi, aatrige01@cs.ucy.ac.cyUniversity of Cyprus, Nicosia, Cyprus; Kishori M. Konwar, kishori@csail.mit.edu; Muriel Medard, medard@mit.edu; Nancy Lynch, lynch@csail.mit.eduMassachusetts Institute of Technology, USA.


XXXX-XXXX/2022/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [10, 11, 21–23, 25, 26, 40]. A notable replication-based algorithm appears in the work by Attiya, Bar-Noy and Dolev [11] (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 DSSes are extremely beneficial to save storage and communication costs while maintaining similar fault-tolerance levels as in replication based DSSes [16]. 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 DSSes is amplified with the emergence of highly optimized erasure coding libraries, that reduce encoding/decoding overheads [3, 12, 46]. In fact, an exciting recent body of systems and optimization works [7, 33, 46, 49, 52–54, 58] 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 [13, 15, 16, 20, 34, 35, 56]. 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 [9], 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 [39] and DynaStore [8] are two of the handful of algorithms [17, 24, 27, 32, 47, 48] that allows reconfiguration on live systems; all these algorithms are replication-based.

A related body of work appeared for *erasure coded scaling*, although there exists important differences that distinguish the two problems. In particular works like [30, 50, 51, 55, 57] consider RAID-based systems with synchronous network communication and local computation. Synchrony allows processes to make assumptions on the time of message delivery, and in turn help them to infer whether a communicating party has failed or not. On an asynchronous system, similar to the one we consider in this work, messages may be delivered with arbitrary delays. Therefore, it is impossible to distinguish whether a message from a source is in transit or the source has crashed before sending a message. This uncertainty makes it impossible to detect failed from operating nodes, and thus challenging to design algorithms to guarantee atomicity (strong consistency) and completion of reads and writes.

Despite the attractive prospects of creating strongly consistent DSSes 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 [7, 33, 46, 49, 52–54, 58] and non-blocking erasure coded based consistent storage [13, 15, 16, 20, 34, 35, 56]. Specifically, the uses of our *reconfigurable* algorithm 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* [29, 38] 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 [8, 17, 24, 27, 32, 39, 47, 48], 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 [15, 20] 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.

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,

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

| Algorithm | #rounds /write | #rounds /read | Reconfig. | Repl. or EC | Storage cost | read bandwidth | write bandwidth |
|---|---|---|---|---|---|---|---|
| CASGC [14] | 3 | 2 | No | EC | $(\delta+1)\frac{n}{k}$ | $\frac{n}{k}$ | $\frac{n}{k}$ |
| SODA [34] | 2 | 2 | No | EC | $\frac{n}{k}$ | $(\delta+1)\frac{n}{k}$ | $\frac{n^2}{k}$ |
| ORCAS-A [20] | 3 | $\geq 2$ | No | EC | $n$ | $n$ | $n$ |
| ORCAS-B [20] | 3 | 3 | No | EC | $\infty$ | $\infty$ | $\infty$ |
| ABD [11] | 2 | 2 | No | Repl. | $n$ | $2n$ | $n$ |
| RAMBO [39] | 2 | 2 | Yes | Repl. | $\geq n$ | $\geq n$ | $\geq n$ |
| DYNASTORE [8] | $\geq 4$ | $\geq 4$ | Yes | Repl. | $\geq n$ | $\geq n$ | $\geq n$ |
| SMARTMERGE [32] | 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, $\delta < 4$ [16].

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 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.

We developed a *proof-of-concept* (PoC) implementation of ARES, and deployed it over a set of distributed devices in the experimental testbed Emulab [2]. The most important take home message from our experimental results is to show that it is possible to implement our algorithm according to the specifications and produces a correct execution and remains available during reconfiguration. Although, the correctness of the algorithm is shown analytically, the experimental validation corroborates the correctness. For this purpose, we have chosen simple parametarization (e.g., uniform selection of read/write invocation intervals), and picked ABD [11] as a benchmark algorithm which, despite being proposed more than 25 years ago, is the fundamental algorithm for emulating replicated quorum-based atomic shared memory. For instance, it is adopted in commercial/open-source implementations like Cassandra [36][†], and is being used as a standard benchmark algorithm (as can be seen in other recent works [19]). However, to demonstrate a real-world application we would need to compare with more algoritrhms and utilize a wide range of read/write distributions, and this is planned as a separate work.

[†]Cassandra [36] offers tuneable consistency, it uses protocol that is essentially ABD [11] for what they refer to as level 3 consistency (i.e., atomicity).

**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 7 provides operation latency and cost analysis, and Section 8 the DAP flexibility. Section 9 presents an experimental evaluation of the proposed algorithms. We conclude our work in Section 10. Due to lack of space omitted proofs can be found in [43].

## 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 sets, called *quorums*, s.t. each pair of quorums intersect, 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 $\xi$ is *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 [29]) property if the following holds [38]. 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 [31] 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 $\tau_2.w > \tau_1.w$.

For ease of reference, Table 2 presents the key notation used in this paper. Notice that some of the symbols shown are defined and used in following sections.

| | |
|---|---|
| $\mathcal{S}$ | the set of server identifiers |
| $\mathcal{I}$ | the set of client identifiers |
| $\mathcal{R}$ | the set of reader identifiers in $\mathcal{I}$ |
| $\mathcal{W}$ | the set of writer identifiers in $\mathcal{I}$ |
| $\mathcal{G}$ | the set of reconfigurer identifiers in $\mathcal{I}$ |
| $\mathcal{V}$ | the set of values allowed to be written on the shared object |
| $v$ | a value in $\mathcal{V}$ |
| $\mathcal{T}$ | the set of pairs in $\mathbb{N} \times \mathcal{W}$ |
| $\tau$ | a pair $(z, w) \in \mathcal{T}$ |
| $\mathcal{C}$ | the set of configuration identifiers |
| $c$ | a configuration with identifier in $\mathcal{C}$ |
| $c.Servers$ | the set of servers s.t. $c.Servers \subseteq \mathcal{S}$ in configuration $c$ |
| $c.Quorums$ | the set of subsets of servers s.t. $\forall Q \in c.Quorums$, $Q \subseteq c.Servers$ and $\forall Q_1, Q_2 \in c.Quorums, Q_1 \cap Q_2 \neq \emptyset$ |
| $\sigma$ | the state of an algorithm $A$ |
| $\sigma_p$ | the state of process $p \in \mathcal{I} \cup \mathcal{S}$ in state $\sigma$ determined over a set of state variables |
| $p.var\|_\sigma$ | the value of the state variable $var$ at process $p$ in state $\sigma$ |
| $\xi$ | an execution of algorithm $A$ which is a finite or infinite sequence of alternative states and actions beginning with the initial state of $A$ |
| $\Phi([v_1, \ldots, v_k])$ or $\Phi([v])$ | the $[n, k]$ encoder function given $k$ fragments of value $v$, $[v_1, \ldots, v_k]$ |
| $e_i$ | the $i^{th}$ encoded word, for $1 \leq i \leq n$, produced by $\Phi([v])$ |
| $\mathcal{G}_L$ | configuration sequence composed of pairs in $\{\mathcal{C} \cup \{\perp\}\} \times \{F, P\}$, where $F$ finalized and $P$ pending, and initially contains $\langle c_0, F\rangle$ |

Table 2. List of Symbols used to describe our model of computation.

## 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 $c \in \mathcal{C}$: $(i)$ $c$.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 $C$, the three primitives are defined as follows:

DEFINITION 1 (DATA ACCESS PRIMITIVES). *Given a configuration identifier $c \in C$, 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 6, 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 $c \in C$, $\langle \tau_\phi, v_\phi \rangle \in \mathcal{T} \times \mathcal{V}$, 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 a $c$.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 Algorithm 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 terminates, under the failure model assumed.

## 4 THE 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 form 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 $\langle cfg, status\rangle$, where $cfg$ is a configuration identifier ($cfg \in \mathcal{C}$), 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 $nextC \in \{\mathcal{C} \cup \{\perp\}\} \times \{P, F\}$, which is used to point to the configuration that follows $c$ in $\mathcal{G}_L$. Initially, at any server $nextC = \langle \perp, F\rangle$. Once $nextC$ 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., $\{s.nextC : s \in c.Servers \wedge s.nextC \neq \perp\}$ 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 $\langle c_0, F\rangle$.

Reconfiguration clients may introduce new configurations, each associated with a unique configuration identifier from $\mathcal{C}$. 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 [37], RAFT [45]) , 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})
      cs ← seq[μ]
4:    while cs ≠ ⊥ do
        cs ←get-next-config(cs.cfg)
6:      if cs ≠ ⊥ then
          μ ← μ + 1
8:        seq[μ] ← cs
          put-config(seq[μ − 1].cfg, seq[μ])
10:   end while
      return seq
12: end procedure

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

    procedure put-config(c, nextC)
24:   send (WRITE-CONFIG, nextC) to each s ∈
      c.Servers
        until ∃Q, Q ∈ c.Quorums s.t. rec_i receives ACK
      from ∀s ∈ Q
26: 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(), which returns the next configuration, assigned to $\widehat{c}$. While $\widehat{c} \neq \perp$, then: (a) $\widehat{c}$ is appended at the end of the sequence *seq*; (b) a put-config action is invoked to inform a quorum of servers in $c.Servers$ to update the value of their *nextC* variable to $\widehat{c}$. If $\widehat{c} = \perp$ 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}
 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].cfg.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 $c_k$
**State Variables:**
$\tau \in \mathbb{N} \times \mathcal{W}$, initially, $\langle 0, \perp \rangle$
$v \in V$, initially, $\perp$
$nextC \in C \times \{P, F\}$, initially $\langle \perp, P \rangle$

**Upon receive** (READ-CONFIG) $_{s_i,\, c_k}$ **from** $q$
send $nextC$ to $q$

**end receive**

**Upon receive** (WRITE-CONFIG, $cfgT_{in}$) $_{s_i,\, c_k}$ **from** $q$
**if** $nextC.cfg = \perp \vee nextC.status = P$ **then**
$nextC \leftarrow cfgT_{in}$
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 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 $d \in C$ 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 $cseq[i]$ 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 $cseq[\nu].cfg$.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 $d = cseq[\nu].cfg$, to $F$. The reconfigurer $rec_i$ informs a quorum of servers in the previous configuration $c = cseq[\nu - 1].cfg$, i.e. in some $Q \in c.Quorums$, about the change of status, by executing the put-config($c, \langle d, F \rangle$) action.

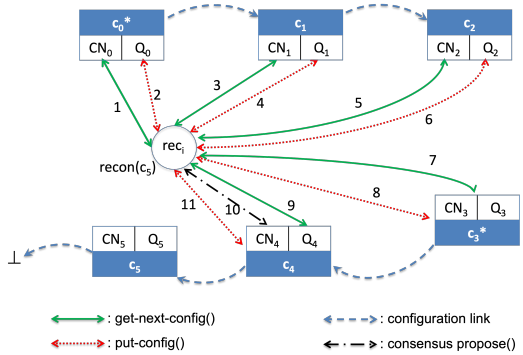



Fig. 1. Illustration of an execution of the reconfiguration steps.

**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 (get-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($c_5$).

## 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 $\mu \le j \le |cseq|$ (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 $\mu \le j \le |cseq|$ (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 $\langle \tau_{max}, v_{max}\rangle$.

**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] ∈ C × {F, P}
       Initialization:
 6:    cseq[0] = ⟨c_0, F⟩

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

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

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

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

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 $\mathcal{G}_L$. This is an essential step that ensures that any new value of the object is propagated in any recently introduced configuration. Omission to do so may lead an operation that reads from a newly established configuration to obtain an outdated value for the shared object, violating this way atomic consistency. Each operation repeats 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 [28] and the framework in [48]. The reconfiguration technique used in ARES ensures the prefix property on the configuration sequence (resembling a blockchain data structure [42]) 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 [48], allowing implementations of atomic read/write registers without relying on strong objects, like ranked registers [18]. Note that ARES is designed to capture a wide

class of algorithms with different redundancy strategies. So while not directly implementing an EC-based atomic memory, it provides the "vehicle" without which dynamic EC-based implementations would not have been possible. Lastly, even though ARES is designed to support crash failures, as noted by [41], reconfiguration is more general and allows an algorithm to handle benign recoveries. That is, a recovered node that loses its state can be introduced as a new member of a new configuration. Stateful recoveries on the other hand are indistinguishable from long delays, thus can be handled effectively by an algorithm designed for the asynchronous model like ARES.

## 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. Notice that the total number of servers in the system can be larger than $n$, however we can pick a subset of $n$ servers to use for this particular key and instance of the algorithm. 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.

**Algorithm 5** DAP implementation for ARES.

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

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

**procedure** c.get-data()
  **send** (QUERY-LIST) to each $s \in c.Servers$
  **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
  $Tags_{dec}^{\geq k}$ = set of tags that appears in $k$ lists with values
  $t^*_{max} \leftarrow \max Tags_*^{\geq k}$
  $t^{dec}_{max} \leftarrow \max Tags_{dec}^{\geq k}$
  **if** $t^{dec}_{max} = t^*_{max}$ **then**
    $v \leftarrow$ decode value for $t^{dec}_{max}$
    **return** $\langle t^{dec}_{max}, v \rangle$
**end procedure**

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

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 $\lceil \frac{n+k}{2} \rceil$ 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 $\lceil \frac{n+k}{2} \rceil$ 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

**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 $c_k$

**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** $q$
  $\tau_{max} = \max_{(t,c)\in List} t$
  Send $\tau_{max}$ to $q$
**end receive**

**Upon receive** (QUERY-LIST) $_{s_i,\, c_k}$ **from** $q$
  Send $List$ to $q$
**end receive**

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

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.

## 5.1 Correctness of the DAPs

To proof the correctness of the proposed DAPs, we need to show that they are both *safe*, i.e. ensure the necessary Property 1, and *live*, i.e. they allow each operation to terminate. We first proceed to prove that for any given execution $\xi$ containing operations of the proposed implementation, then examining any pair of operations in $\xi$ satisfy the DAP consistency properties (i.e. Property 1). That is, the tag returned by a get-tag() operation is larger than the value written by any preceding put-data() operation, and the value returned by a get-data() operation is either written by a put-data() operation or is the initial value of the object. Next, assuming that there cannot be more that $\delta$ put-data() operations concurrent with a single get-data() operation, we show that each operation in our implementation terminates. Otherwise a get-data() operation is at risk of not being able to discover a decodable value and thus fail to terminate and return a value.

**Safety (Property 1).** 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.*

PROOF. As mentioned above we are concerned with only configuration $c$, and therefore, in our proofs it suffices to examine only one configuration. Let $\xi$ be some execution of ARES, then we consider two cases for $\pi$ for proving property $C1$: $\pi$ is a get-tag, or $\pi$ is a get-data primitive.

Case $(a)$: $\phi$ is $c$.put-data$(\langle \tau_\phi, v_\phi \rangle)$ and $\pi$ is a $c$.get-tag() returns $\tau_\pi \in \mathcal{T}$. Let $c_\phi$ and $c_\pi$ denote the clients that invokes $\phi$ and $\pi$ in $\xi$. Let $S_\phi \subset \mathcal{S}$ denote the set of $\left\lceil \frac{n+k}{2} \right\rceil$ servers that responds to $c_\phi$, during $\phi$. Denote by $S_\pi$ the set of $\left\lceil \frac{n+k}{2} \right\rceil$ servers that responds to $c_\pi$, during $\pi$. Let $T_1$ be a point in execution $\xi$ after the completion of $\phi$ and before the invocation of $\pi$. Because $\pi$ is invoked after $T_1$, therefore, at $T_1$ each of the servers in $S_\phi$ contains $t_\phi$ in its *List* variable. Note that, once a tag is added to *List*, it is never removed. Therefore, during $\pi$, any server in $S_\phi \cap S_\pi$ responds with *List* containing $t_\phi$ to $c_\pi$. Note that since $|S_\phi| = |S_\pi| = \left\lceil \frac{n+k}{2} \right\rceil$ implies $|S_\phi \cap S_\pi| \geq k$, and hence $t^{dec}_{max}$ at $c_\pi$, during $\pi$ is at least as large as $t_\phi$, i.e., $t_\pi \geq t_\phi$. Therefore, it suffices to prove our claim with respect to the tags and the decodability of its corresponding value.

Case $(b)$: $\phi$ is $c$.put-data$(\langle \tau_\phi, v_\phi \rangle)$ and $\pi$ is a $c$.get-data() returns $\langle \tau_\pi, v_\pi \rangle \in \mathcal{T} \times \mathcal{V}$. As above, let $c_\phi$ and $c_\pi$ be the clients that invokes $\phi$ and $\pi$. Let $S_\phi$ and $S_\pi$ be the set of servers that responds to $c_\phi$ and $c_\pi$, respectively. Arguing as above, $|S_\phi \cap S_\pi| \geq k$ and every server in $S_\phi \cap S_\pi$ sends $t_\phi$ in response to $c_\phi$, during $\pi$, in their *List*'s and hence $t_\phi \in Tags^{\geq k}_*$. Now, because $\pi$ completes in $\xi$, hence we have $t^*_{max} = t^{dec}_{max}$. Note that $\max Tags^{\geq k}_* \geq \max Tags^{\geq k}_{dec}$ so $t_\pi \geq \max Tags^{\geq k}_{dec} = \max Tags^{\geq k}_* \geq t_\phi$. Note that each tag is always associated with its corresponding value $v_\pi$, or the corresponding coded elements $\Phi_s(v_\pi)$ for $s \in \mathcal{S}$.

Next, we prove the *C*2 property of DAP for the ARES algorithm. Note that the initial values of the *List* variable in each servers $s$ in $\mathcal{S}$ is $\{(t_0, \Phi_s(v_\pi))\}$. Moreover, from an inspection of the steps of the algorithm, new tags in the *List* variable of any servers of any servers is introduced via put-data operation. Since $t_\pi$ is returned by a get-tag or get-data operation then it must be that either $t_\pi = t_0$ or $t_\pi > t_0$. In the case where $t_\pi = t_0$ then we have nothing to prove. If $t_\pi > t_0$ then there must be a put-data$(t_\pi, v_\pi)$ operation $\phi$. To show that for every $\pi$ it cannot be that $\phi$ completes before $\pi$, we adopt by a contradiction. Suppose for every $\pi$, $\phi$ completes before $\pi$ begins, then clearly $t_\pi$ cannot be returned $\phi$, a contradiction. □

**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 $\pi$.*

PROOF. Note that in the read and write operation the get-tag and put-data operations initiated by any non-faulty client always complete. Therefore, the liveness property with respect to any write operation is clear because it uses only get-tag and put-data operations of the DAP. So, we focus on proving the liveness property of any read operation $\pi$, specifically, the get-data operation completes. Let $\xi$ be and execution of ARES and let $c_\omega$ and $c_\pi$ be the clients that invokes the write operation $\omega$ and read operation $\pi$, respectively.

Let $S_\omega$ be the set of $\left\lceil \frac{n+k}{2} \right\rceil$ servers that responds to $c_\omega$, in the put-data operations, in $\omega$. Let $S_\pi$ be the set of $\left\lceil \frac{n+k}{2} \right\rceil$ servers that responds to $c_\pi$ during the get-data step of $\pi$. Note that in $\xi$ at the point execution $T_1$, just before the execution of $\pi$, none of the write operations in $\Lambda$ is complete. Observe that, by algorithm design, the coded-elements corresponding to $t_\omega$ are garbage-collected from the *List* variable of a server only if more than $\delta$ higher tags are introduced by subsequent writes into the server. Since the number of concurrent writes $|\Lambda|$, s.t. $\delta > |\Lambda|$ the corresponding value of tag $t_\omega$ is not garbage collected in $\xi$, at least until execution point $T_2$ in any of the servers in $S_\omega$.

Therefore, during the execution fragment between the execution points $T_1$ and $T_2$ of the execution $\xi$, the tag and coded-element pair is present in the *List* variable of every in $S_\omega$ that is active. As a result, the tag and coded-element pairs, $(t_\omega, \Phi_s(v_\omega))$ exists in the *List* received from any $s \in S_\omega \cap S_\pi$ during operation $\pi$. Note that since $|S_\omega| = |S_\pi| = \left\lceil \frac{n+k}{2} \right\rceil$ hence $|S_\omega \cap S_\pi| \geq k$ and hence $t_\omega \in Tags^{\geq k}_{dec}$, the set of decode-able tag, i.e., the value $v_\omega$ can be decoded by $c_\pi$ in $\pi$, which demonstrates that $Tags^{\geq k}_{dec} \neq \emptyset$. Next we want to argue that $t^*_{max} = t^{dec}_{max}$ via a contradiction: we assume $\max Tags^{\geq k}_{*} > \max Tags^{\geq k}_{dec}$. Now, consider any tag $t$, which exists due to our assumption, such that, $t \in Tags^{\geq k}_{*}$, $t \notin Tags^{\geq k}_{dec}$ and $t > t^{dec}_{max}$. Let $S^k_\pi \subset S$ be any subset of $k$ servers that responds with $t^*_{max}$ in their *List* variables to $c_\pi$. Note that since $k > n/3$ hence $|S_\omega \cap S^k_\pi| \geq \left\lceil \frac{n+k}{2} \right\rceil + \left\lceil \frac{n+1}{3} \right\rceil \geq 1$, i.e., $S_\omega \cap S^k_\pi \neq \emptyset$. Then $t$ must be in some servers in $S_\omega$ at $T_2$ and since $t > t^{dec}_{max} \geq t_\omega$. Now since $|\Lambda| < \delta$ hence $(t, \perp)$ cannot be in any server at $T_2$ because there are not enough concurrent write operations (i.e., writes in $\Lambda$) to garbage-collect the coded-elements corresponding to tag $t$, which also holds for tag $t^*_{max}$. In that case, $t$ must be in $Tag^{\geq k}_{dec}$, a contradiction. □

## 6 CORRECTNESS OF ARES

In this section, we prove that ARES correctly implements an atomic, read/write, shared storage service. The correctness of ARES highly depends on the way the configuration sequence is constructed at each client process. Also, atomicity is ensured if the DAP implementation in each configuration $c_i$ satisfies Property 1.

As a roadmap, we begin by showing that some critical properties are preserved by the reconfiguration service proposed in subsection 6.1. In particular, we show that the configuration sequence maintained in two processes is either the same or one is the prefix of the other. This in turn helps us to proof the correctness of ARES in subsection 6.2 by showing that all the properties of atomicity

(see Section 2) are satisfied, given the properties on the configuration sequence hold, and that the DAPs used in each configuration satisfy Property 1.

We proceed by first introducing some definitions and notation, that we 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 $\xi$ (i.e. the value of the variable remains unchanged).

We define as the tag of a configuration $c$ the smallest tag among the maximum tags found in each quorum of $c$. This is essentially the smallest tag that an operation may witness when receiving replies from a single quorum in $c$. More formally:

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.*

Next we provide the notation to express the configuration sequence witnessed by a process $p$ in a state $\sigma$ (as $p.cseq|_\sigma$), the last finalized configuration in that sequence (as $\mu(\mathbf{c}_\sigma^p)$), and the length of that sequence (as $\nu(\mathbf{c}_\sigma^p)$). More formally:

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

Last, we define the prefix operation on two configuration sequences.

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 $\mathbf{x}[j] \neq \perp$.*

Table 3 summarizes the new notation for ease of reference.

| | |
|---|---|
| $\mathbf{c}_\sigma^p$ | the value of the configuration sequence variable *cseq* at process $p$ in state $\sigma$, i.e. a shorthand of $p.cseq|_\sigma$ |
| $\mathbf{c}_\sigma^p[i]$ | the $i^{th}$ element in the configuration sequence $\mathbf{c}_\sigma^p$ |
| $\mu(\mathbf{c}_\sigma^p)$ | last finalized configuration in $\mathbf{c}_\sigma^p$ |
| $\nu(\mathbf{c}_\sigma^p)$ | the length of $\mathbf{c}_\sigma^p$, i.e. $\|\mathbf{c}_\sigma^p\|$ |

Table 3. Additional notation used in this section.

## 6.1 Reconfiguration Protocol Properties

In this section we analyze the properties that we can achieve through our reconfiguration algorithm. In high-level, we do show that the following properties are preserved:

i **configuration uniqueness:** the configuration sequences in any two processes have identical configuration at any place $i$,
ii **sequence prefix:** the configuration sequence witnessed by an operation is a prefix of the sequence witnessed by any succeeding operation, and
iii **sequence progress:** if the configuration with index $i$ is finalized during an operation, then a configuration $j$, for $j \geq i$, will be finalized for a succeeding operation.

The first lemma shows that any two configuration sequences have the same configuration identifiers in the same indexes.

LEMMA 9. *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 10. *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 its $s.nextC$ variable for some specific configuration $c_k$ in a state $\sigma$ only when it receives a WRITE-CONF message. This is either the first WRITE-CONF message received at $s$ for $c_k$ (and thus $nextC = \perp$), or $s.nextC = \langle *, P\rangle$ and the message received contains a tuple $\langle c, F\rangle$. 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 11 (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 9, 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 thus by the agreement property the have to decide 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 $s.nextC(c_0) \neq \perp$, 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 A2:8. That completes the proof. □

Lemma 11 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 12. *In any execution $\xi$ of the algorithm , If for any process $p \in \mathcal{I}$, $\mathbf{c}^p_\sigma[i] \neq \perp$ in some state $\sigma$ in $\xi$, then $\mathbf{c}^p_{\sigma'}[i] \neq \perp$ 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 $\perp$ (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 $c \neq \perp$. Thus, in this case $\mathbf{c}^p_*[i]$ cannot be $\perp$. Also in the read-config procedure, $\mathbf{c}^p_*[i]$ is assigned to a value different than $\perp$ according to Line A2:8. Hence, if $\mathbf{c}^p_\sigma[i] \neq \perp$ at state $\sigma$ then it cannot become $\perp$ in any state $\sigma'$ after $\sigma$ in execution $\xi$. □

LEMMA 13. *Let $\sigma_1$ be some state in an execution $\xi$ of the algorithm. Then for any process $p$, if $k = max\{i : \mathbf{c}^p_{\sigma_1}[i] \neq \perp\}$, then $\mathbf{c}^p_{\sigma_1}[j] \neq \perp$, for $0 \leq j < k$.*

PROOF. Let us assume to derive contradiction that there exists $j < k$ such that $\mathbf{c}^p_{\sigma_1}[j] = \perp$ and $\mathbf{c}^p_{\sigma_1}[j+1] \neq \perp$. Consider first 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 $\mathbf{c}^p_{\sigma_1}[k] \neq \perp$, 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 9 $p$ decides for $k$ when it runs consensus over configuration $\mathbf{c}^p_{\sigma_1}[k-1].cfg$. Since this is equal to $\perp$, 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, $\mathbf{c}^p_{\sigma_1}[k-1] = \perp$ at $\sigma_1$. So if state $\sigma'$ is before $\sigma_1$ in $\xi$, then by Lemma 12, it follows that $\mathbf{c}^p_{\sigma'}[k-1] = \perp$. 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 and similar reasoning we can show that $\mathbf{c}^p_{\sigma_1}[j] \neq \perp$, for $0 \leq j \leq k-1$. □

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 14. *If at a state $\sigma$ of an execution $\xi$ of the algorithm $\mu(\mathbf{c}^p_\sigma) = k$ for some process $p$, then for any element $0 \leq j < k$, $\exists Q \in \mathbf{c}^p_\sigma[j].cfg.Quorums$ 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 $\mathbf{c}^p_\sigma[k] \neq \bot$, then $p$ propagated each $\mathbf{c}^p_{\sigma'}[j]$, for $0 < j \leq k$ to a quorum of servers $Q \in \mathbf{c}^p_{\sigma'}[j-1].cfg.Quorums$. And this completes the proof. □

LEMMA 15 (SEQUENCE 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 11 for any $i$ such that $\mathbf{c}^{p_1}_{\sigma_1}[i] \neq \bot$ and $\mathbf{c}^{p_2}_{\sigma_2}[i] \neq \bot$, then $\mathbf{c}^{p_1}_{\sigma_1}[i].cfg = \mathbf{c}^{p_2}_{\sigma_2}[i].cfg$. Also from Lemma 13 we know that for $0 \leq j \leq \nu_1$, $\mathbf{c}^{p_1}_{\sigma_1}[j] \neq \bot$, and $0 \leq j \leq \nu_2$, $\mathbf{c}^{p_2}_{\sigma_2}[j] \neq \bot$. So if we can show that $\nu_1 \leq \nu_2$ 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 $\mu \leq \nu_2$. If $\nu_1 \leq \mu$ then $\nu_1 \leq \nu_2$ 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 14, we know that by $\sigma_1$, $\exists Q \in \mathbf{c}^{p_1}_{\sigma_1}[j].cfg.Quorums$, for $0 \leq j < \nu_1$, 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 11, 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 $\mathbf{c}^{p_1}_{\sigma_1}[\mu+1] \neq \bot$ 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 $\mu \leq j < \nu_1$, 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 $0 \leq j \leq \nu_1$. Hence $\nu_1 \leq \nu_2$ 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 16. *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 $\mu(\mathbf{c}^p_\sigma) \leq \mu(\mathbf{c}^p_{\sigma'})$.*

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 $j \geq k$. But $\mathbf{c}^p_{\sigma'}[j].cfg$ is the last finalized configuration at $\sigma'$ and hence $\mu(\mathbf{c}^p_{\sigma'}) \geq \mu(\mathbf{c}^p_\sigma)$. □

LEMMA 17 (SEQUENCE 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 $\pi_1 \to \pi_2$ 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 $\mu(\mathbf{c}^{p_1}_{\sigma_1}) \leq \mu(\mathbf{c}^{p_2}_{\sigma_2})$.*

PROOF. By Lemma 15 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 16 $\mu_2 \leq \mu(\mathbf{c}^{p_2}_{\sigma_2})$ 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 15 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 11, $\mathbf{c}^{p_1}_{\sigma_1}[\mu_1 - 1].cfg = \mathbf{c}^{p_2}_{\sigma_2}[\mu_1 - 1].cfg$. 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 10), 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. □

Using the previous Lemmas we can conclude to the main result of this section.

THEOREM 18. *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*) ***Configuration Consistency**: $\mathbf{c}^{p_2}_{\sigma_2}[i].cfg = \mathbf{c}^{p_1}_{\sigma_1}[i].cfg$, for $1 \leq i \leq \nu(\mathbf{c}^{p_1}_{\sigma_1})$,*
(*b*) ***Sequence Prefix**: $\mathbf{c}^{p_1}_{\sigma_1} \leq_p \mathbf{c}^{p_2}_{\sigma_2}$, and*
(*c*) ***Sequence Progress**: $\mu(\mathbf{c}^{p_1}_{\sigma_1}) \leq \mu(\mathbf{c}^{p_2}_{\sigma_2})$*

PROOF. Statements (*a*), (*b*) and (*c*) follow from Lemmas 11, 15, and 16. □

### 6.2 Atomicity Property of ARES

Given the properties satisfied by the reconfiguration algorithm of ARES in any execution, we can now proceed to examine whether our algorithm satisfies the safety (atomicity) conditions. The propagation of the information of the distributed object is achieved using the get-tag, get-data, and put-data actions. We assume that the DAP used satisfy Property 1 as presented in Section 3, and we will show that, given such assumption, ARES satisfies atomicity.

We begin with a lemma stating that if a reconfiguration operation retrieves a configuration sequence of length $k$ during its read-config action, then it installs/finalizes the $k + 1$ configuration in the global configuration sequence.

LEMMA 19. *Let $\pi$ be a complete reconfiguration operation by a reconfigurer $rc$ in an execution $\xi$ of* ARES. *if $\sigma_1$ is the state in $\xi$ following the termination of the* read-config *action during $\pi$, then $\pi$ invokes a* finalize-config$(\mathbf{c}^{rc}_{\sigma_2})$ *at a state $\sigma_2$ in $\xi$, with $\nu(\mathbf{c}^{rc}_{\sigma_2}) = \nu(\mathbf{c}^{rc}_{\sigma_1}) + 1$.*

PROOF. This lemma follows directly from the implementation of the reconfig operation. Let $\pi$ be a reconfiguration operation reconfig$(c)$. At first, $\pi$ invokes a read-config to retrieve a latest value of the global configuration sequence, $\mathbf{c}^{rc}_{\sigma_1}$, in the state $\sigma_1$ in $\xi$. During the add-config action, $\pi$ proposes the addition of $c$, and appends at the end of $\mathbf{c}^{rc}_{\sigma_1}$ the decision $d$ of the consensus protocol. Therefore, if $\mathbf{c}^{rc}_{\sigma_1}$ is extended by $\langle d, P\rangle$ (Line A 2:17), and hence the add-config action returns a configuration sequence $\mathbf{c}^{rc}_{\sigma'_1}$ with length $\nu(\mathbf{c}^{rc}_{\sigma'_1}) = \nu(\mathbf{c}^{rc}_{\sigma_1}) + 1$. As $\nu(\mathbf{c}^{rc}_{\sigma'_1}$ does not change during the update-config action, then $\mathbf{c}^{rc}_{\sigma'_1}$ is passed to the finalize-config action at state $\sigma_2$, and hence $\mathbf{c}^{rc}_{\sigma_2} = \mathbf{c}^{rc}_{\sigma'_1}$. Thus, $\nu(\mathbf{c}^{rc}_{\sigma_2}) = \nu(\mathbf{c}^{rc}_{\sigma'_1}) = \nu(\mathbf{c}^{rc}_{\sigma_1}) + 1$ and the lemma follows. □

The next lemma states that only some reconfiguration operation $\pi$ may finalize a configuration $c$ at index $j$ in a configuration sequence $p.cseq$ at any process $p$. To finalize $c$, the lemma shows that $\pi$ must

witness a configuration sequence such that its last finalized configuration appears at an index $i < j$ in the configuration sequence $p.cseq$. In other words, reconfigurations always finalize configurations that are ahead from their latest observed final configuration, and it seems like "jumping" from one final configuration to the next.

LEMMA 20. *Suppose $\xi$ is an execution of* ARES. *For any state $\sigma$ in $\xi$, if $\mathbf{c}_\sigma^p[j].status = F$ for some process $p \in \mathcal{I}$, then there exists a* reconfig *operation $\pi$ by a reconfigurer $rc \in \mathcal{G}$, such that (i) $rc$ invokes* finalize-config($\mathbf{c}_{\sigma'}^{rc}$) *during $\pi$ at some state $\sigma'$ in $\xi$, (ii) $\nu(\mathbf{c}_{\sigma'}^{rc}) = j$, and (iii) $\mu(\mathbf{c}_{\sigma'}^{rc}) < j$.*

PROOF. A process sets the status of a configuration $c$ to $F$ in two cases: (i) either during a finalize-config($seq$) action such that $\nu(seq) = \langle c, P\rangle$ (Line A2:33), or (ii) when it receives $\langle c, F\rangle$ from a server $s$ during a read-next-config action. Server $s$ sets the status of a configuration $c$ to $F$ only if it receives a message that contains $\langle c, F\rangle$ (Line A3:10). So, (ii) is possible only if $c$ is finalized during a reconfig operation.

Let, w.l.o.g., $\pi$ be the first reconfiguration operation that finalizes $\mathbf{c}_\sigma^p[j].cfg$. To do so, process $rc$ invokes finalize-config($\mathbf{c}_{\sigma_1'}^{rc}$) during $\pi$, at some state $\sigma'$ that appears before $\sigma$ in $\xi$. By Lemma 11, $\mathbf{c}_\sigma^p[j].cfg = \mathbf{c}_{\sigma'}^{rc}[j].cfg$. Since, $rc$ finalizes $\mathbf{c}_{\sigma'}^{rc}[j]$, then this is the last entry of $\mathbf{c}_{\sigma'}^{rc}$ and hence $\nu(\mathbf{c}_{\sigma'}^{rc}) = j$. Also, by Lemma 20 it follows that the read-config action of $\pi$ returned a configuration $\mathbf{c}_{\sigma''}^{rc}$ in some state $\sigma''$ that appeared before $\sigma'$ in $\xi$, such that $\nu(\mathbf{c}_{\sigma''}^{rc}) < \nu(\mathbf{c}_{\sigma'}^{rc})$. Since by definition, $\mu(\mathbf{c}_{\sigma''}^{rc}) \leq \nu(\mathbf{c}_{\sigma''}^{rc})$, then $\mu(\mathbf{c}_{\sigma''}^{rc}) < j$. However, since only $\langle c, P\rangle$ is added to $\mathbf{c}_{\sigma''}^{rc}$ to result in $\mathbf{c}_{\sigma'}^{rc}$, then $\mu(\mathbf{c}_{\sigma''}^{rc}) = \mu(\mathbf{c}_{\sigma'}^{rc})$. Therefore, $\mu(\mathbf{c}_{\sigma'}^{rc}) < j$ as well and the lemma follows. □

We now reach an important lemma of this section. By ARES, before a read/write/reconfig operation completes it propagates the maximum tag it discovered by executing the put-data action in the last configuration of its local configuration sequence (Lines A2:18, A4:16, A4:38). When a subsequent operation is invoked, it reads the latest configuration sequence by beginning from the last finalized configuration in its local sequence and invoking read-data to all the configurations until the end of that sequence. The lemma shows that the latter operation will retrieve a tag which is higher than the tag used in the put-data action of any preceding operation.

LEMMA 21. *Let $\pi_1$ and $\pi_2$ be two completed read/write/reconfig operations invoked by processes $p_1$ and $p_2$ in $\mathcal{I}$, in an execution $\xi$ of* ARES, *such that, $\pi_1 \rightarrow \pi_2$. If $c_1$.*put-data($\langle \tau_{\pi_1}, v_{\pi_1}\rangle$) *is the last* put-data *action of $\pi_1$ and $\sigma_2$ is the state in $\xi$ after the completion of the first* read-config *action of $\pi_2$, then there exists a $c_2$.*put-data($\langle \tau, v\rangle$) *action in some configuration $c_2 = \mathbf{c}_{\sigma_2}^{p_2}[k].cfg$, for $\mu(\mathbf{c}_{\sigma_2}^{p_2}) \leq k \leq \nu(\mathbf{c}_{\sigma_2}^{p_2})$, such that (i) it completes in a state $\sigma'$ before $\sigma_2$ in $\xi$, and (ii) $\tau \geq \tau_{\pi_1}$.*

PROOF. Note that from the definitions of $\nu(\cdot)$ and $\mu(\cdot)$, we have $\mu(\mathbf{c}_{\sigma_2}^{p_2}) \leq \nu(\mathbf{c}_{\sigma_2}^{p_2})$. Let $\sigma_1$ be the state in $\xi$ after the completion of $c_1$.put-data($\langle \tau_{\pi_1}, v_{\pi_1}\rangle$) and $\sigma_1'$ be the state in $\xi$ following the response step of $\pi_1$. Since any operation executes put-data on the last discovered configuration then $c_1$ is the last configuration found in $\mathbf{c}_{\sigma_1}^{p_1}$, and hence $c_1 = \mathbf{c}_{\sigma_1}^{p_1}[\nu(\mathbf{c}_{\sigma_1}^{p_1})].cfg$. By Lemma 16 we have $\mu(\mathbf{c}_{\sigma_1}^{p_1}) \leq \mu(\mathbf{c}_{\sigma_1'}^{p_1})$ and by Lemma 17 we have $\mu(\mathbf{c}_{\sigma_1'}^{p_1}) \leq \mu(\mathbf{c}_{\sigma_2}^{p_2})$, since $\pi_2$ (and thus its first read-config action) is invoked after $\sigma_1'$ (and thus after the last read-config action during $\pi_1$). Hence, combining the two implies that $\mu(\mathbf{c}_{\sigma_1}^{p_1}) \leq \mu(\mathbf{c}_{\sigma_2}^{p_2})$. Now from the last implication and the first statement we have $\mu(\mathbf{c}_{\sigma_1}^{p_1}) \leq \nu(\mathbf{c}_{\sigma_2}^{p_2})$. Therefore, it remains to examine whether the last finalized configuration witnessed by $p_2$ appears before or after $c_1$, i.e.: (*a*) $\mu(\mathbf{c}_{\sigma_2}^{p_2}) \leq \nu(\mathbf{c}_{\sigma_1}^{p_1})$ and (*b*) $\mu(\mathbf{c}_{\sigma_2}^{p_2}) > \nu(\mathbf{c}_{\sigma_1}^{p_1})$.

**Case** (*a*)**:** Since $\pi_1 \rightarrow \pi_2$ then, by Theorem 18, $\mathbf{c}_{\sigma_2}^{p_2}$ value returned by read-config at $p_2$ during the execution of $\pi_2$ satisfies $\mathbf{c}_{\sigma_1}^{p_1} \preceq_p \mathbf{c}_{\sigma_2}^{p_2}$. Therefore, $\nu(\mathbf{c}_{\sigma_1}^{p_1}) \leq \nu(\mathbf{c}_{\sigma_2}^{p_2})$, and hence in this case $\mu(\mathbf{c}_{\sigma_2}^{p_2}) \leq \nu(\mathbf{c}_{\sigma_1}^{p_1}) \leq \nu(\mathbf{c}_{\sigma_2}^{p_2})$. Since $c_1$ is the last configuration in $\mathbf{c}_{\sigma_1}^{p_1}$, then it has index $\nu(\mathbf{c}_{\sigma_1}^{p_1})$. So if we take

$c_2 = c_1$ then the $c_1$.put-data($\langle \tau_{\pi_1}, v_{\pi_1} \rangle$) action trivially satisfies both conditions of the lemma as: (i) it completes in state $\sigma_1$ which appears before $\sigma_2$, and (ii) it puts a pair $\langle \tau, v \rangle$ such that $\tau = \tau_{\pi_1}$.

**Case** $(b)$**:** This case is possible if there exists a reconfiguration client $rc$ that invokes reconfig operation $\rho$, during which it executes the finalize-config($\mathbf{c}_*^{rc}$) that finalized configuration with index $\nu(\mathbf{c}_*^{rc}) = \mu(\mathbf{c}_{\sigma_2}^{p_2})$. Let $\sigma$ be the state immediately after the read-config of $\rho$. Now, we consider two sub-cases: $(i)$ $\sigma$ appears before $\sigma_1$ in $\xi$, or $(ii)$ $\sigma$ appears after $\sigma_1$ in $\xi$.

*Subcase* $(b)(i)$*:* Since read-config at $\sigma$ completes before the invocation of last read-config of operation $\pi_1$ then, either $\mathbf{c}_\sigma^{rc} \prec_p \mathbf{c}_{\sigma_1}^{p_1}$, or $\mathbf{c}_\sigma^{rc} = \mathbf{c}_{\sigma_1}^{p_1}$ due to Lemma 15. Suppose $\mathbf{c}_\sigma^{rc} \prec_p \mathbf{c}_{\sigma_1}^{p_1}$, then according to Lemma 19 $rc$ executes finalize-config on configuration sequence $\mathbf{c}_*^{rc}$ with $\nu(\mathbf{c}_*^{rc}) = \nu(\mathbf{c}_\sigma^{rc}) + 1$. Since $\nu(\mathbf{c}_*^{rc}) = \mu(\mathbf{c}_{\sigma_2}^{p_2})$, then $\mu(\mathbf{c}_{\sigma_2}^{p_2}) = \nu(\mathbf{c}_\sigma^{rc}) + 1$. If however, $\mathbf{c}_\sigma^{rc} \prec_p \mathbf{c}_{\sigma_1}^{p_1}$, then $\nu(\mathbf{c}_\sigma^{rc}) < \nu(\mathbf{c}_{\sigma_1}^{p_1})$ and thus $\nu(\mathbf{c}_\sigma^{rc}) + 1 \leq \nu(\mathbf{c}_{\sigma_1}^{p_1})$. This implies that $\mu(\mathbf{c}_{\sigma_2}^{p_2}) \leq \nu(\mathbf{c}_{\sigma_1}^{p_1})$ which contradicts our initial assumption for this case that $\mu(\mathbf{c}_{\sigma_2}^{p_2}) > \nu(\mathbf{c}_{\sigma_1}^{p_1})$. So this sub-case is impossible.

Now suppose, that $\mathbf{c}_\sigma^{rc} = \mathbf{c}_{\sigma_1}^{p_1}$. Then it follows that $\nu(\mathbf{c}_\sigma^{rc}) = \nu(\mathbf{c}_{\sigma_1}^{p_1})$, and that $\mu(\mathbf{c}_{\sigma_2}^{p_2}) = \nu(\mathbf{c}_{\sigma_1}^{p_1}) + 1$ in this case. Since $\sigma_1$ is the state after the last put-data during $\pi_1$, then if $\sigma_1'$ is the state after the completion of the last read-config of $\pi_1$ (which follows the put-data), it must be the case that $\mathbf{c}_{\sigma_1}^{p_1} = \mathbf{c}_{\sigma_1'}^{p_1}$. So, during its last read-config process $p_1$ does not read the configuration indexed at $\nu(\mathbf{c}_{\sigma_1}^{p_1}) + 1$. This means that the put-config completes in $\rho$ at state $\sigma_\rho$ after $\sigma_1'$ and the update-config operation is invoked at state $\sigma_\rho'$ after $\sigma_\rho$ with a configuration sequence $\mathbf{c}_{\sigma_\rho'}^{rc}$. During the update operation $\rho$ invokes get-data operation in every configuration $\mathbf{c}_{\sigma_\rho'}^{rc}[i].cfg$, for $\mu(\mathbf{c}_{\sigma_\rho'}^{rc}) \leq i \leq \nu(\mathbf{c}_{\sigma_\rho'}^{rc})$. Notice that $\nu(\mathbf{c}_{\sigma_\rho'}^{rc}) = \mu(\mathbf{c}_{\sigma_2}^{p_2}) = \nu(\mathbf{c}_{\sigma_1}^{p_1}) + 1$ and moreover the last configuration of $\mathbf{c}_{\sigma_\rho'}^{rc}$ was just added by $\rho$ and it is not finalized. From this it follows that $\mu(\mathbf{c}_{\sigma_\rho'}^{rc}) < \nu(\mathbf{c}_{\sigma_\rho'}^{rc})$, and hence $\mu(\mathbf{c}_{\sigma_\rho'}^{rc}) \leq \nu(\mathbf{c}_{\sigma_1}^{p_1})$. Therefore, $\rho$ executes get-data in configuration $\mathbf{c}_{\sigma_\rho'}^{rc}[j].cfg$ for $j = \nu(\mathbf{c}_{\sigma_1}^{p_1})$. Since $p_1$ invoked put-data($\langle \tau_{\pi_1}, v_{\pi_1} \rangle$) at the same configuration $c_1$, and completed in a state $\sigma_1$ before $\sigma_\rho'$, then by **C1** of Property 1, it follows that the get-data action will return a tag $\tau \geq \tau_{\pi_1}$. Therefore, the maximum tag that $\rho$ discovers is $\tau_{max} \geq \tau \geq \tau_{\pi_1}$. Before invoking the finalize-config action, $\rho$ invokes $c_1$.put-data($\langle \tau_{max}, v_{max} \rangle$). Since $\nu(\mathbf{c}_{\sigma_\rho'}^{rc}) = \mu(\mathbf{c}_{\sigma_2}^{p_2})$, and since by Lemma 11, then the action put-data is invoked in a configuration $c_2 = \mathbf{c}_{\sigma_2}^{p_2}[j].cfg$ such that $j = \mu(\mathbf{c}_{\sigma_2}^{p_2})$. Since the read-config action of $\pi_2$ observed configuration $\mu(\mathbf{c}_{\sigma_2}^{p_2})$, then it must be the case that $\sigma_2$ appears after the state where the finalize-config was invoked and therefore after the state of the completion of the put-data action during $\rho$. Thus, in this case both properties are satisfied and the lemma follows.

*Subcase* $(b)(ii)$*:* Suppose in this case that $\sigma$ occurs in $\xi$ after $\sigma_1$. In this case the last put-data in $\pi_1$ completes before the invocation of the read-config in $\rho$ in execution $\xi$. Now we can argue recursively, $\rho$ taking the place of operation $\pi_2$, that $\mu(\mathbf{c}_\sigma^{rc}) \leq \nu(\mathbf{c}_\sigma^{rc})$ and therefore, we consider two cases: $(a)$ $\mu(\mathbf{c}_\sigma^{rc}) \leq \nu(\mathbf{c}_{\sigma_1}^{p_1})$ and $(b)$ $\mu(\mathbf{c}_\sigma^{rc}) > \nu(\mathbf{c}_{\sigma_1}^{p_1})$. Note that there are finite number of operations invoked in $\xi$ before $\pi_2$ is invoked, and hence the statement of the lemma can be shown to hold by a sequence of inequalities. □

The following lemma shows the consistency of operations as long as the DAP used satisfy Property 1.

LEMMA 22. *Let $\pi_1$ and $\pi_2$ denote completed read/write operations in an execution $\xi$, from processes $p_1, p_2 \in \mathcal{I}$ respectively, such that $\pi_1 \rightarrow \pi_2$. If $\tau_{\pi_1}$ and $\tau_{\pi_2}$ are the local tags at $p_1$ and*

*$p_2$ after the completion of $\pi_1$ and $\pi_2$ respectively, then $\tau_{\pi_1} \leq \tau_{\pi_2}$; if $\pi_1$ is a write operation then $\tau_{\pi_1} < \tau_{\pi_2}$.*

PROOF. Let $\langle \tau_{\pi_1}, v_{\pi_1} \rangle$ be the pair passed to the last put-data action of $\pi_1$. Also, let $\sigma_2$ be the state in $\xi$ that follows the completion of the first read-config action during $\pi_2$. Notice that $\pi_2$ executes a loop after the first read-config operation and performs $c$.get-data (if $\pi_2$ is a read) or $c$.get-tag (if $\pi_2$ is a write) from all $c = \mathbf{c}^{p_2}_{\sigma_2}[i].cfg$, for $\mu(\mathbf{c}^{p_2}_{\sigma_2}) \leq i \leq \nu(\mathbf{c}^{p_2}_{\sigma_2})$. By Lemma 21, there exists a $c'$.put-data($\langle \tau, v \rangle$) action by some operation $\pi'$ on some configuration $c' = \mathbf{c}^{p_2}_{\sigma_2}[j].cfg$, for $\mu(\mathbf{c}^{p_2}_{\sigma_2}) \leq j \leq \nu(\mathbf{c}^{p_2}_{\sigma_2})$, that completes in some state $\sigma'$ that appears before $\sigma_2$ in $\xi$. Thus, the get-data or get-tag invoked by $p_2$ on $\mathbf{c}^{p_2}_{\sigma_2}[j].cfg$, occurs after state $\sigma_2$ and thus after $\sigma'$. Since the DAP primitives used satisfy **C1** and **C2** of Property 1, then the get-tag action will return a tag $\tau'_{\pi_2}$ or a get-data action will return a pair $\langle \tau'_{\pi_2}, v'_{\pi_2} \rangle$, with $\tau'_{\pi_2} \geq \tau$. As $p_2$ gets the maximum of all the tags returned, then by the end of the loop $p_2$ will retrieve a tag $\tau_{max} \geq \tau'_{\pi_2} \geq \tau \geq \tau_{\pi_1}$.

If now $\pi_2$ is a read, it returns $\langle \tau_{max}, v_{max} \rangle$ after propagating that value to the last discovered configuration. Thus, $\tau_{\pi_2} \geq \tau_{\pi_1}$. If however $\pi_2$ is a write, then before propagating the new value the writer increments the maximum timestamp discovered (Line A4:13) generating a tag $\tau_{\pi_2} > \tau_{max}$. Therefore the operation $\pi_2$ propagates a tag $\tau_{\pi_2} > \tau_{\pi_1}$ in this case. □

And the main result of this section follows:

THEOREM 23 (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.*

As algorithm ARES handles each configuration separately, then we can observe that the algorithm may utilize a different mechanism for the put and get primitives in each configuration. So the following remark:

REMARK 24. *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

A major challenge in reconfigurable atomic services is to examine the latency of terminating read and write operations, especially when those are invoked concurrently with reconfiguration operations. In this section we provide an in depth analysis of the latency of operations in ARES. Additionally, a storage and communication analysis is shown when ARES utilizes the erasure-coding algorithm presented in Section 5, in each configuration.

### 7.1 Latency Analysis

The idea behind our latency analysis is quite straight forward: we construct the worst case execution that would allow all concurrent reconfigurations to add their proposed configuration. This leads to the longest configuration sequence that a read/write operation needs to traverse before completing. Thus, given a bounded delay, we compute the delay for each operation and we finally compute how long it is going to take for a read/write operation to catch up in the worst case and complete.

Liveness (termination) 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.

Let us first examine what is the action delays based on the boundaries we assume. It is easy to see that actions put-config, get-next-config perform two message exchanges thus take time $2d \leq T(\phi) \leq 2D$. From this we can derive the delay of a read-config action.

LEMMA 25. *Let $\phi$ be a* read-config *operation invoked by a non-faulty reconfiguration client rc, with the input argument and returned values of $\phi$ as $\mathbf{c}^{rc}_{\sigma}$ and $\mathbf{c}^{rc}_{\sigma'}$ respectively. Then the delay of $\phi$ is:* $4d(\nu(\mathbf{c}^{rc}_{\sigma'}) - \mu(\mathbf{c}^{rc}_{\sigma}) + 1) \leq T(\phi) \leq 4D(\nu(\mathbf{c}^{rc}_{\sigma'}) - \mu(\mathbf{c}^{rc}_{\sigma}) + 1)$.

From Lemma 25 it is clear that the latency of a read-config action depends on the number of configurations installed since the last finalized configuration known to the recon client.

Given the latency of a read-config, we can compute the minimum amount of time it takes for $k$ configurations to be installed.

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 26. *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 27. *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.*

PROOF. Let $\sigma_s$ and $\sigma_e$ be the states before the invocation and after the completion step of a read/write operation $\pi$ by $p$ respectively, in some execution $\xi$ of ARES. By algorithm examination we can see that any read/write operation performs the following actions in this order: $(i)$ read-config, $(ii)$ get-data (or get-tag), $(iii)$ put-data, and $(iv)$ read-config. Let $\sigma_1$ be the state when the first read-config, denoted by read-config$_1$, action terminates. By Lemma 25 the action will take time:

$$T(\text{read-config}_1) \leq 4D(\nu(\mathbf{c}^p_{\sigma_1}) - \mu(\mathbf{c}^p_{\sigma_s}) + 1)$$

The get-data action that follows the read-config (Lines Alg. 4:34-35) also took at most $(\nu(\mathbf{c}^p_{\sigma_1}) - \mu(\mathbf{c}^p_{\sigma_s}) + 1)$ time units, given that no new finalized configuration was discovered by the read-config action. Finally, the put-data and the second read-config actions of $\pi$ may be invoked at most $(\nu(\mathbf{c}^p_{\sigma_e}) - \nu(\mathbf{c}^p_{\sigma_1}) + 1)$ times, given that the read-config action discovers one new configuration every time it runs. Merging all the outcomes, the total time of $\pi$ can be at most:

$$\begin{aligned} T(\pi) &\leq 4D(\nu(\mathbf{c}^p_{\sigma_1}) - \mu(\mathbf{c}^p_{\sigma_s}) + 1) + 2D(\nu(\mathbf{c}^p_{\sigma_1}) - \mu(\mathbf{c}^p_{\sigma_s}) + 1) + (4D + 2D)(\nu(\mathbf{c}^p_{\sigma_e}) - \nu(\mathbf{c}^p_{\sigma_1}) + 1) \\ &\leq 6D\left[\nu(\mathbf{c}^p_{\sigma_e}) - \mu(\mathbf{c}^p_{\sigma_s}) + 2\right] \leq 6D(k+1) \end{aligned}$$

where $\nu(\mathbf{c}^p_{\sigma_e}) - \mu(\mathbf{c}^p_{\sigma_s}) \leq k+1$ since there can be at most $k$ new configurations installed. and the result of the lemma follows. □

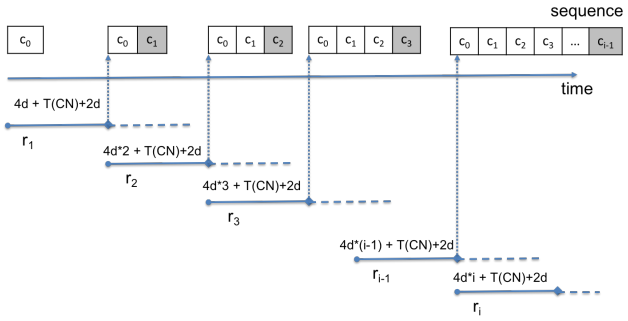



Fig. 2. Successful reconfig operations.

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. We first show how long it takes for $k$ configurations to be installed.

LEMMA 28. *Let $\sigma$ be the last state of a fair execution of* ARES, $\xi$. *Then $k$ configurations can be installed to $\mathbf{c}_\sigma$, in time $T(k) \geq 4d \sum_{i=1}^{k} i + k\,(T_{min}(CN) + 2d)$ time units.*

PROOF. In ARES a reconfig operation has four phases: $(i)$ read-config($cseq$), reads the latest configuration sequence, $(ii)$ add-config($cseq, c$), attempts to add the new configuration at the end of the global sequence $\mathcal{G}_L$, $(iii)$ update-config($cseq$), transfers the knowledge to the added configuration, and $(iv)$ finalize-config($cseq$) finalizes the added configuration. So, a new configuration is appended to the end of the configuration sequence (and it becomes visible to any operation) during the add-config action. In turn, the add-config action, runs a consensus algorithm to decide on the added configuration and then invokes a put-config action to add the decided configuration. Any operation that is invoked after the put-config action observes the newly added configuration.

Notice that when multiple reconfigurations are invoked concurrently, then it might be the case that all participate to the same consensus instance and the configuration sequence is appended by a single configuration. The worst case scenario happens when all concurrent reconfigurations manage to append the configuration sequence by their configuration. In brief, this is possible when the read-config action of each reconfig operation appears after the put-config action of another reconfig operation.

More formally we can build an execution where all reconfig operations append their configuration in the configuration sequence. Consider the partial execution $\xi$ that ends in a state $\sigma$. Suppose that every process $p \in \mathcal{I}$ knows the same configuration sequence, $\mathbf{c}^p_\sigma = \mathbf{c}_\sigma$. Also let the last finalized operation in $\mathbf{c}_\sigma$ be the last configuration of the sequence, e.g. $\mu(\mathbf{c}_\sigma) = \nu(\mathbf{c}_\sigma)$. Notice that $\mathbf{c}_\sigma$ can also be the initial configuration sequence $\mathbf{c}^p_{\sigma_0}$. We extend $\xi_0$ by a series of reconfig operations, such that each reconfiguration $rc_i$ is invoked by a reconfigurer $r_i$ and attempts to add a configuration $c_i$. Let $rc_1$ be the first reconfiguration that performs the following actions without being concurrent with any other reconfig operation:

- read-config starting from $\mu(\mathbf{c}_\sigma)$
- add-config completing both the consensus proposing $c_1$ and the put-config action writing the decided configuration

Since $rc_1$ its not concurrent with any other reconfig operation, then is the only process to propose a configuration in $\mu(\mathbf{c}_\sigma)$, and hence by the consensus algorithm properties, $c_1$ is decided. Thus, $\mathbf{c}_\sigma$ is appended by a tuple $\langle c_1, P\rangle$.

Let now reconfiguration $rc_2$ be invoked immediately after the completion of the add-config action from $rc_1$. Since the local sequence at the beginning of $rc_2$ is equal to $\mathbf{c}_\sigma$, then the read-config action of $rc_2$ will also start from $\mu(\mathbf{c}_\sigma)$. Since, $rc_1$ already propagated $c_1$ to $\mu(\mathbf{c}_\sigma)$ during is put-config action, then $rc_2$ will discover $c_1$ during the first iteration of its read-config action, and thus it will repeat the iteration on $c_1$. Configuration $c_1$ is the last in the sequence and thus the read-config action of $rc_2$ will terminate after the second iteration. Following the read-config action, $rc_2$ attempts to add $c_2$ in the sequence. Since $rc_1$ is the only reconfiguration that might be concurrent with $rc_2$, and since $rc_1$ already completed consensus in $\mu(\mathbf{c}_\sigma)$, then $rc_2$ is the only operation to run consensus in $c_1$. Therefore, $c_2$ is accepted and $rc_2$ propagates $c_2$ in $c_1$ using a put-config action.

So in general we let configuration $rc_i$ to be invoked after the completion of the add-config action from $rc_{i-1}$. As a result, the read-config action of $rc_i$ performs $i$ iterations, and the configuration $c_i$ is added immediately after configuration $c_{i-1}$ in the sequence. Figure 2 illustrates our execution construction for the reconfiguration operations.

It is easy to notice that such execution results in the worst case latency for all the reconfiguration operations $rc_1, rc_2, \ldots, rc_i$. As by Lemma 25 a read-config action takes at least $4d$ time to complete, then as also seen in Figure 2, $k$ reconfigs may take time $T(k) \geq \sum_{i=1}^{k} [4d * i + (T_{min}(CN) + 2d)]$. Therefore, it will take time $T(k) \geq 4d \sum_{i=1}^{k} i + k\,(T_{min}(CN) + 2d)$ and the lemma follows. □

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 $\xi$.*

PROOF. By Lemma 28, $k$ configurations may be installed in: $T(k) \geq 4d \sum_{i=1}^{k} i + k\,(T_{min}(CN) + 2d)$. Also by Lemma 27, we know that operation $\pi$ takes at most $T(\pi) \leq 6D\left(\nu(\mathbf{c}^p_{\sigma_e}) - \mu(\mathbf{c}^p_{\sigma_s}) + 2\right)$. Assuming that $k = \nu(\mathbf{c}^p_{\sigma_e}) - \mu(\mathbf{c}^p_{\sigma_s})$, the total number of configurations observed during $\pi$, then $\pi$ may terminate before a $k+1$ configuration is added in the configuration sequence if $6D(k+2) \leq 4d \sum_{i=1}^{k} i + k\,(T_{min}(CN) + 2d)$ then we have $d \geq \frac{3D}{k} - \frac{T_{min}(CN)}{2(k+2)}$. And that completes the lemma. □

### 7.2 Storage and Communication Costs for ARES.

Storage and Communication costs for ARES highly depends on the DAP that we use in each configuration. For our analysis we assume that each configuration utilizes the algorithms and the DAPs presented in Section 5.

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.

LEMMA 30. *The worst-case total storage cost of Algorithm 5 is $(\delta + 1)\frac{n}{k}$.*

PROOF. The maximum number of (tag, coded-element) pair in the *List* is $\delta + 1$, and the size of each coded element is $\frac{1}{k}$ while the tag variable is a metadata and therefore, not counted. So, the total storage cost is $(\delta + 1)\frac{n}{k}$. □

We next state the communication cost for the write and read operations in Aglorithm 5. Once again, note that we ignore the communication cost arising from exchange of meta-data.

LEMMA 31. *The communication cost associated with a successful write operation in Algorithm 5 is at most* $\frac{n}{k}$.

PROOF. During read operation, in the get-tag phase the servers respond with their highest tags variables, which are metadata. However, in the put-data phase, the reader sends each server the coded elements of size $\frac{1}{k}$ each, and hence the total cost of communication for this is $\frac{n}{k}$. Therefore, we have the worst case communication cost of a write operation is $\frac{n}{k}$. □

LEMMA 32. *The communication cost associated with a successful read operation in Algorithm 5 is at most* $(\delta + 2)\frac{n}{k}$.

PROOF. During read operation, in the get-data phase the servers respond with their *List* variables and hence each such list is of size at most $(\delta + 1)\frac{1}{k}$, and then counting all such responses give us $(\delta + 1)\frac{n}{k}$. In the put-data phase, the reader sends each server the coded elements of size $\frac{1}{k}$ each, and hence the total cost of communication for this is $\frac{n}{k}$. Therefore, we have the worst case communication cost of a read operation is $(\delta + 2)\frac{n}{k}$. □

From the above Lemmas we get.

THEOREM 33. *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* $(\delta + 2)\frac{n}{k}$.

## 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 34 (ATOMICITY OF TEMPLATE *A*). *Suppose the DAP implementation satisfies the consistency properties C*1 *and C*2 *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* $c.\mathcal{F}$.

PROOF. We prove the atomicity by proving properties *A*1, *A*2 and *A*3 presented in Section 2 for any execution of the algorithm.

*Property A*1: Consider two operations $\phi$ and $\pi$ such that $\phi$ completes before $\pi$ is invoked. We need to show that it cannot be the case that $\pi \prec \phi$. We break our analysis into the following four cases:

Case (*a*): *Both $\phi$ and $\pi$ are writes*. The $c$.put-data($*$) of $\phi$ completes before $\pi$ is invoked. By property **C1** the tag $\tau_\pi$ returned by the $c$.get-data() at $\pi$ is at least as large as $\tau_\phi$. Now, since $\tau_\pi$ is incremented by the write operation then $\pi$ puts a tag $\tau'_\pi$ such that $\tau_\phi < \tau'_\pi$ and hence we cannot have $\pi \prec \phi$.

Case (*b*): *$\phi$ is a write and $\pi$ is a read*. In execution $\xi$ since $c$.put-data($\langle t_\phi, * \rangle$) of $\phi$ completes before the $c$.get-data() of $\pi$ is invoked, by property **C1** the tag $\tau_\pi$ obtained from the above $c$.get-data() is at least as large as $\tau_\phi$. Now $\tau_\phi \leq \tau_\pi$ implies that we cannot have $\pi \prec \phi$.

Case (*c*): *$\phi$ is a read and $\pi$ is a write*. Let the id of the writer that invokes $\pi$ we $w_\pi$. The $c$.put-data($\langle \tau_\phi, * \rangle$) call of $\phi$ completes before $c$.get-tag() of $\pi$ is initiated. Therefore, by property **C1** get-tag($c$) returns $\tau$ such that, $\tau_\phi \leq \tau$. Since $\tau_\pi$ is equal to $inc(\tau)$ by design of the algorithm, hence $\tau_\pi > \tau_\phi$ and we cannot have $\pi \prec \phi$.

Case (*d*): *Both $\phi$ and $\pi$ are reads*. In execution $\xi$ the $c$.put-data($\langle t_\phi, * \rangle$) is executed as a part of $\phi$ and completes before $c$.get-data() is called in $\pi$. By property **C1** of the data-primitives, we have $\tau_\phi \leq \tau_\pi$ and hence we cannot have $\pi \prec \phi$.

*Property A*2: Note that because the tag set $\mathcal{T}$ is well-ordered we can show that A2 holds by first showing that every write has a unique tag. This means that any two pair of writes can be ordered. Note that a read can be ordered w.r.t. any write operation trivially if the respective tags are different, and by definition, if the tags are equal the write is ordered before the read.

Observe that two tags generated from different writers are necessarily distinct because of the id component of the tag. Now if the operations, say $\phi$ and $\pi$ are writes from the same writer then, by well-formedness property, the second operation will witness a higher integer part in the tag by property **C1**, and since the $c$.get-tag() is followed by $c$.put-data($*$). Hence $\pi$ is ordered after $\phi$.

*Property A*3: By **C2** the $c$.get-data() may return a tag $\tau$, only when there exists an operation $\pi$ that invoked a $c$.put-data($\langle \tau, * \rangle$). Otherwise it returns the initial value. Since a write is the only operation to put a new tag $\tau$ in the system then Property *A*3 follows from **C2**. □

### 8.1 Representing Known Algorithms in terms of data-access primitives

A number of known tag-based algorithms that implement atomic read/write objects (e.g., ABD [11], FAST[21] ), can be expressed in terms of DAP. In this subsection we demonstrate how we can transform the very celebrated ABD algorithm [11].

MWABD *Algorithm.* The multi-writer version of the ABD can be transformed to the generic algorithm Template *A*. Algorithm 8 illustrates the three DAP for the ABD algorithm. The get-data primitive encapsulates the query phase of MWABD, while the put-data primitive encapsulates the propagation phase of the algorithm.

Let us now examine if the primitives satisfy properties **C1** and **C2** of Property 1. We begin with a lemma that shows the monotonicity of the tags at each server.

LEMMA 35. *Let $\sigma$ and $\sigma'$ two states in an execution $\xi$ such that $\sigma$ appears before $\sigma'$ in $\xi$. Then for any server $s \in \mathcal{S}$ it must hold that $s.tag|_\sigma \leq s.tag|_{\sigma'}$.*

PROOF. According to the algorithm, a server $s$ updates its local tag-value pairs when it receives a message with a higher tag. So if $s.tag|_\sigma = \tau$ then in a state $\sigma'$ that appears after $\sigma$ in $\xi$, $s.tag|_{\sigma'} \geq \tau$. □

**Algorithm 8** Implementation of DAP for ABD at each process $p$ using configuration $c$

**Data-Access Primitives at process $p$:**

**procedure** c.put-data($\langle \tau, v \rangle$)
  **send** (WRITE, $\langle \tau, v \rangle$) to each $s \in c.Servers$
  **until** $\exists Q, Q \in c.Quorums$ s.t. $p$ receives ACK from $\forall s \in Q$
**end procedure**

**procedure** c.get-tag()
  **send** (QUERY-TAG) to each $s \in c.Servers$
  **until** $\exists Q, Q \in c.Quorums$ s.t.
    $p$ receives $\langle \tau_s, v_s \rangle$ from $\forall s \in Q$
  $\tau_{max} \leftarrow \max(\{\tau_s : p$ received $\langle \tau_s, v_s \rangle$ from $s\})$
  **return** $\tau_{max}$
**end procedure**

**procedure** c.get-data()
  **send** (QUERY) to each $s \in c.Servers$
  **until** $\exists Q, Q \in c.Quorums$ s.t.
    $p$ receives $\langle \tau_s, v_s \rangle$ from $\forall s \in Q$
  $\tau_{max} \leftarrow \max(\{\tau_s : r_i$ received $\langle \tau_s, v_s \rangle$ from $s\})$
  **return** $\{\langle \tau_s, v_s \rangle : \tau_s = \tau_{max} \wedge p$ received $\langle \tau_s, v_s \rangle$ from $s\}$
**end procedure**

---

**Primitive Handlers at server $s_i$ in configuration $c$:**

**Upon receive** (QUERY-TAG) **from** $q$
  send $\tau$ to $q$
**end receive**

**Upon receive** (QUERY) **from** $q$
  send $\langle \tau, v \rangle$ to $q$
**end receive**

**Upon receive** (WRITE, $\langle \tau_{in}, v_{in} \rangle$) **from** $q$
  **if** $\tau_{in} > \tau$ **then**
    $\langle \tau, v \rangle \leftarrow \langle \tau_{in}, v_{in} \rangle$
  send ACK to $q$
**end receive**

In the following two lemmas we show that property **C1** is satisfied, that is if a put-data action completes, then any subsequent get-data and get-tag actions will discover a higher tag than the one propagated by that put-data action.

LEMMA 36. *Let $\phi$ be a* c.put-data($\langle \tau, v \rangle$) *action invoked by $p_1$ and $\gamma$ be a* c.get-tag() *action invoked by $p_2$ in a configuration c, such that $\phi \rightarrow \gamma$ in an execution $\xi$ of the algorithm. Then $\gamma$ returns a tag $\tau_\gamma \geq \tau$.*

PROOF. The lemma follows from the intersection property of quorums. In particular, during the $c$.put-data($\langle \tau, v \rangle$) action, $p_1$ sends the pair $\langle \tau, v \rangle$ to all the servers in $c.Servers$ and waits until all the servers in a quorum $Q_i \in c.Quorums$ reply. When those replies are received then the action completes.

During a $c$.get-data() action on the other hand, $p_2$ sends query messages to all the servers in $c.Servers$ and waits until all servers in a quorum $Q_j \in c.Quorums$ (not necessarily different than $Q_i$) reply. By definition $Q_i \cap Q_j \neq \emptyset$, thus any server $s \in Q_i \cap Q_j$ reply to both $\phi$ and $\gamma$ actions. By Lemma 35 and since $s$ received a tag $\tau$, then $s$ replies to $p_2$ with a tag $\tau_s \geq \tau$. Since $\gamma$ returns the maximum tag it discovers then $\tau_\gamma \geq \tau_s$. Therefore $\tau_\gamma \geq \tau$ and this completes the proof. □

With similar arguments and given that each value is associated with a unique tag then we can show the following lemma.

LEMMA 37. *Let $\pi$ be a* c.put-data($\langle \tau, v \rangle$) *action invoked by $p_1$ and $\phi$ be a* c.get-data() *action invoked by $p_2$ in a configuration c, such that $\pi \rightarrow \phi$ in an execution $\xi$ of the algorithm. Then $\phi$ returns a tag-value $\langle \tau_\phi, v_\phi \rangle$ such that $\tau_\phi \geq \tau$.*

Finally we can now show that property **C2** also holds.

LEMMA 38. *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 a* c.put-data($\langle \tau_\pi, v_\pi \rangle$) *and $\phi \nrightarrow \pi$.*

PROOF. This follows from the facts that (i) servers set their tag-value pair to a pair received by a put-data action, and (ii) a get-data action returns a tag-value pair that it received from a server. So if a $c$.get-data() operation $\phi$ returns a tag-value pair $\langle \tau_\pi, v_\pi \rangle$, there should be a server $s$ that replied to that operation with $\langle \tau_\pi, v_\pi \rangle$, and $s$ received $\langle \tau_\pi, v_\pi \rangle$ from some $c$.put-data($\langle \tau_\pi, v_\pi \rangle$) action, $\pi$. Thus, $\pi$ can proceed or be concurrent with $\phi$, and hence $\phi \not\rightarrow \pi$. □

## 9 EXPERIMENTAL EVALUATION

The theoretical findings suggest that ARES is an algorithm to provide robustness and flexibility on shared memory implementations, without sacrificing strong consistency. In this section we present a *proof-of-concept* implementation of ARES and we run preliminary experiments to get better insight on the efficiency and adaptiveness of ARES. In particular, our experiments measure the latency of each read, write, and reconfig operations, and examine the persistence of consistency even when the service is reconfigured between configurations that add/remove servers and utilize different shared memory algorithms.

### 9.1 Experimental Testbed

We ran experiments on two different setups: (i) simulated locally on a single machine, and (ii) on a LAN. Both type of experiments run on *Emulab* [2], an emulated WAN environment testbed used for developing, debugging, and evaluating the systems. We used nodes with two 2.4 GHz 64-bit 8-Core E5-2630 "Haswell" processors, 64 GB RAM, with 1 GB and 10 GB NICs. In both setups we used an external implementation of Raft[45] consensus algorithms, which was used for the service reconfiguration (line 16 of Alg. 2) and was deployed on top of small RPi devices. Small devices introduced further delays in the system, reducing the speed of reconfigurations and creating harsh conditions for longer periods in the service. The Python implementation of Raft used for consensus is PySyncObj [5]. Some modifications were done to allow the execution of Raft in the ARES environment. We built an HTTP API for the management of the Raft subsystem. A reconfigurer can propose a configuration at a particular index in the configuration sequence by sending a POST request to the url of each Raft node, and receives a response from the RAFT on which configuration is decided for that index.

**Local Experimental Setup:** The local setup was used to have access to a global synchronized clock (the clock of the local machine) in order to examine whether our algorithm preserves global ordering and hence atomicity even when using different algorithms between configurations. Therefore, all the instances are hosted on the same physical machine avoiding the skew between computer clocks in a distributed system. Furthermore, the use of one clock guarantees that when an event occurs after another, it will assign a later time.

**Distributed Experimental Setup:** The distributed experiments in Emulab enabled the examination of the performance of the algorithm in a close to real environment. For the deployment and remote execution of the experimental tasks on the Emulab, we used *Ansible Playbooks* [1]. All physical nodes were placed on a single LAN using a DropTail queue without delay or packet loss. Each physical machine runs one server or client process. This guarantees a fair communication delay between a client and a server node.

**Node Types:** In all experiments, we use four distinct types of nodes, writers, readers , reconfigurers and servers. Their main role is listed below:

- **writer** $w \in W \subseteq C$ **:** a client that sends write requests to all servers and waits for a quorum of the servers to reply
- **reader** $r \in R \subseteq C$**:** a client that sends read requests to servers and waits for a quorum of the servers to reply

- **reconfigurer** $g \in G \subseteq C$**:** a client that sends reconfiguration requests to servers and waits for a quorum of the servers to reply
- **server** $s \in S$**:** a server listens for read and write requests, it updates its object replica according to the atomic shared memory and replies to the process that originated the request.

**Performance Metric:** The metric for evaluating the algorithms is the operational latency. This includes both communication and computational delays. The operation latency is computed as the average of all clients' average operation latencies. For better estimations, each experiment in every scenario was repeated 6 times. In the graphs, we use error bars to illustrate the standard error of the mean (SEM) from the 6 repeated experiments.

## 9.2 Experimental Scenarios

In this section, we describe the scenarios we constructed and the settings for each of them. In our scenarios we constructed the $DAP_s$ and used two different atomic storage algorithms in ARES: (i) the erasure coding based algorithm presented in Section 5, and (ii) the ABD algorithm (see Section 8.1).
**Implementation of** $DAP_s$**:** Clients initialize the appropriate configuration objects to handle any request. Notice that the client creates a configuration object when it is the first time that the client requests an operation or when doing a reconfiguration operation. Once the configuration object is initialized, it is stored on the client *cseq* and it is retrieved directly on any subsequent request from the client. Therefore, the $DAP_s$ procedures are called from a configuration object. The asynchronous communication between components is achieved by using DEALER and ROUTER sockets, from the ZeroMQ Python library [6].
**Erasure Coding:** The type of erasure coding we use is ($n$,$k$)-Reed-Solomon code, which guarantees that any $k$ of $n$ coded fragments is enough to reassemble the original object. The parameter $k$ is the number of encoded data fragments, $n$ is the total number of servers and $m$ is the number of parity fragments, i.e. $n - k$. A high number of $k$ and consequently a small number of $m$ means less redundancy with the system tolerating fewer failures. When $k = 1$ we essentially converge to replication. In practice, the get-data and put-data functions from algorithm 5 integrate the standard Reed-Solomon implementation provided by liberasurecode from the PyEClib Python library [4].
**Fixed Parameters:** In all scenarios, the number of servers is fixed to 10. The number of writers and the value of delta are set to 5; delta being the maximum number of concurrent put-data operations. The parity value of the EC is set to 2 in order to minimize the redundancy, leading to 8 data servers and 2 parity servers. It is worth mentioning that the quorum size of the EC algorithm is $\left\lceil \frac{10+8}{2} \right\rceil = 9$, while the quorum size of ABD algorihtm is $\left\lfloor \frac{10}{2} \right\rfloor + 1 = 6$. In relation to the EC algorithm, we can conclude that the parameter $k$ is directly proportional to the quorum size. But as the value of $k$ and quorum size increase, the size of coded elements decreases.
**Distributed Experiments:** For the distributed experiments we use a *stochastic* invocation scheme in which readers and writers pick a random time uniformly distributed (discrete) between intervals to invoke their next operations. Respectively the intervals are $[1...rInt]$ and $[1..wInt]$, where $rInt, wInt = 2sec$. In total, each writer performs 60 writes and each reader 60 reads. The reconfigurer invokes its next operation every $15sec$ and performs a total of 6 reconfigurations. The intervals are set within these values in order to generate a continuous flow of operations and stress the concurrency in the system. Note that these values are not based on any real world scenario.

*In particular, we present six types of scenarios:*

- **File Size Scalability (Emulab):** The first scenario is made to evaluate how the read and write latencies are affected by the size of the shared object. There are two separated runs, one for

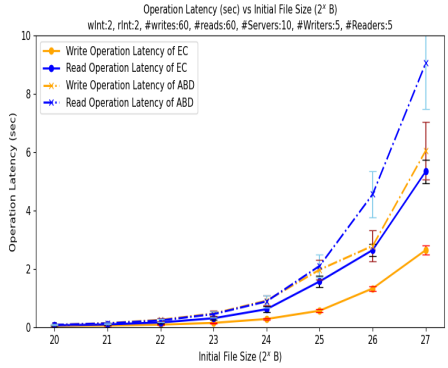
each examined storage algorithm. The file size is doubled from 1 MB to 128 MB. The number of readers is fixed to 5, without any reconfigurers.

- **Reader Scalability (Emulab):** This scenario is constructed to compare the read and write latency of the system with two different storage algorithms, while the readers increase. In particular, we execute two separate runs, one for each storage algorithm. We use only one reconfigurer which requests recon operations that lead to the same shared memory emulation and server nodes. The size of the file used is 4 MB.
- **Changing Reconfigurations (Emulab):** In this scenario, we evaluate how the read and write latencies are affected when increasing the number of readers, while also changing the storage algorithm. We run two different runs which differ in the way the reconfigurer chooses the next storage algorithm: (i) the reconfigurer chooses randomly between the two storage algorithms, and (ii) the reconfigurer switches between the two storage algorithms. The size of the file used, in both scenarios, is 4 MB.
- $k$ **Scalability (Emulab, EC only):** In this scenario, we examine the read and write latencies with different numbers of $k$ (a parameter of Reed-Solomon). We increase the $k$ of the EC algorithm from 1 to 9. The number of readers is fixed to 5, without any reconfigurers. The size of the file used is 4 MB.
- **Changing the number of Reads/Writes (Emulab):** In these scenarios, we examine the read and write latencies with different numbers of read and write operations respectively. We change the number of reads/writes that each reader/writer performs, from 10 to 60, increasing by 10. We calculate all possible pairs of writes and reads. The number of readers is fixed to 5. The reconfigurer switches between the two storage algorithms. The size of the file used is 4 MB
- **Consistency Persistence (Local):** In this scenario, we run multiple client operations in order to check if the data is consistent across servers. The number of readers is set to 5. The readers and writers invoke their next operations without any time delay, while the reconfigurer waits $15sec$ for the next invocation. We run two different scenarios which differ in the reconfigurations. In both scenarios, the reconfigurer switches between the two storage algorithms. In the second scenario, the reconfigurer changes concurrently the quorum of servers. In total, each writer performs 500 writes, each reader 500 reads and the reconfigurer 50 reconfigurations. The size of the file used is 4 MB.

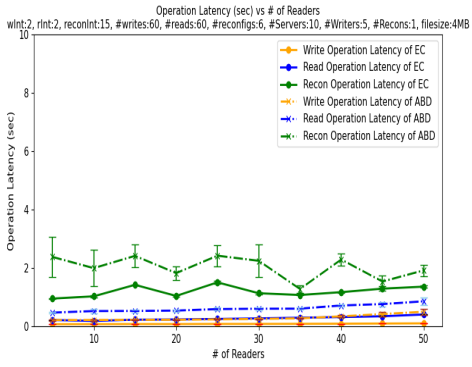
### 9.3 Experimental Results

In this section, we present and explain the evaluation results of each scenario. As a general observation, the ARES algorithm with the EC storage provides data redundancy with a lower communicational and storage cost compared to the ABD storage that uses a strict replication technique.

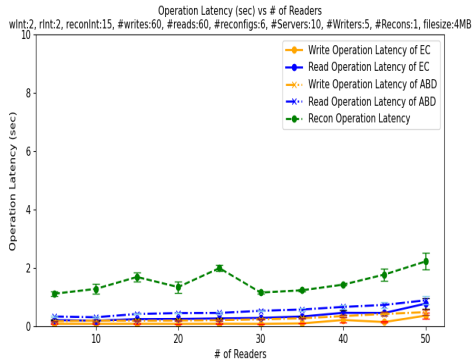
**File Size Scalability Results:** Fig. 3(a) shows the results of the file size scalability experiments. The read and write latencies of both storage algorithms remain in low levels until 16 MB. In bigger sizes we observe the latancies of all operations to grow significantly. It is worth noting that the fragmentation applied by the EC algorithm, benefits its write operations which follow a slower increasing curve than the rest of the operations. From the rest the reads seem to suffer the worst delay hit, as they are engaged in more communication phases. Nevertheless, the larger messages sent by ABD result in slower read operations. We had noticed that EC has lower SEM values than ABD, which indicates that the calculated mean latencies of EC align very closely throughout the experiments. As EC breaks each file into smaller fragments, in combination with the fact that the variation is smaller when using smaller files in ABD, may lead to the conclusion that the file size has a significant impact on the error variation. To this end it appears that larger file sizes introduce higher variation on the delivery times of the file and hence higher statistical errors.



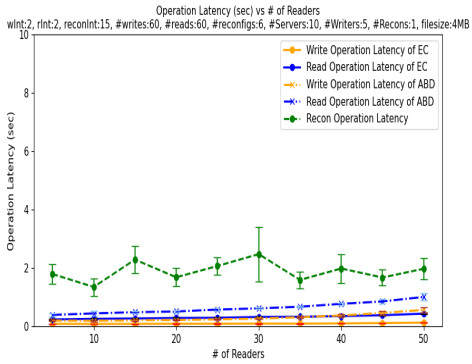

(a)



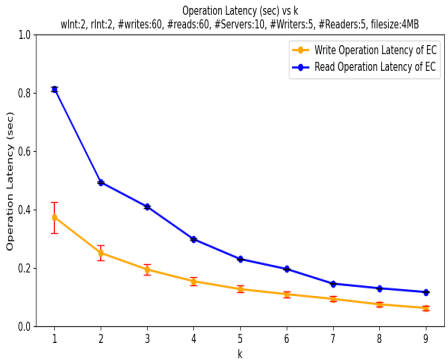

(b)



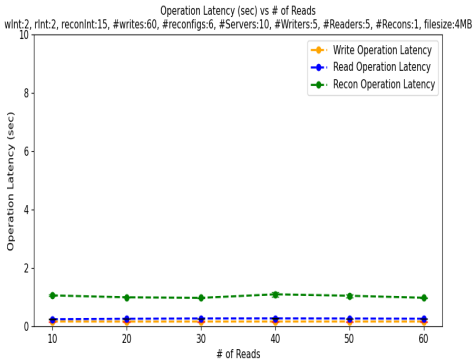

(c)




(d)




(e)




(f)

Fig. 3. Simulation results.

**Reader Scalability Results:** The results of reader scalability experiments can be found in Fig. 3(b). The read and write latencies of both algorithms remain almost unchanged, while the number of readers increases. This indicates that the system does not reach a state where it can not handle the concurrent read operations. Still, the reduced message size of read and write operations in EC keep their latencies lower than the corresponding latencies of ABD. On the other hand, the reconfiguration latency in both algorithms witnesses wild fluctuations between about 1 sec and 4 sec. This is probably

due to the unstable connection in the external service which handles the reconfigurations. Notice that the number of readers does not have a great impact on the results we obtain in each experiment as the SEM error bars are small. The same goes for the next scenario where the number of readers changes while switching algorithms between reconfigurations.

**Changing Reconfigurations Results:** Fig. 3(c) illustrates the results of experiments with the random storage change. While, in Fig. 3(d), we can find the results of the experiments when the reconfigurer switches between storage algorithms. During both experiments, there are cases where a single read/write operation may access configurations that implement both ABD and EC algorithms, when concurrent with a recon operation. Thus, the latencies of such operations are accounted in both ABD and EC latencies. As we mentioned earlier, our choice of $k$ minimizes the coded fragment size but introduces bigger quorums and thus larger communication overhead. As a result, in smaller file sizes, ARES may not benefit from the coding, bringing the delays of the two algorithms closer to each other. It is again obvious that the reconfiguration delays are higher than the delays of all other operations.

**$k$ Scalability Results:** From Figs. 3(e) we can infer that when smaller $k$ are used, the write and read latencies reach their highest values. In both cases, small $k$ results in the generation of smaller number of data fragments and thus bigger sizes of the fragments and higher redundancy. For example we can see that for RS(10,8) and RS(10,7) we have the same size of quorum, equal to 9, whereas the latter has more redundant information. As a result, with a higher number of $m$ (i.e. smaller $k$) we achieve higher levels of fault-tolerance, but that it would waste storage efficiency. The write latency seems to be less affected by the number of $k$ since the encoding is considerably faster as it requires less computation. In conclusion, there appears to be a trade-off between operation latency and fault-tolerance in the system: the further increase of the $k$ (and thus lower fault-tolerance) the smaller the latency of read/write operations. This experiment proves that the object size plays a significant role on the error variation. Notice that while $k$ is small, and thus the object we send out is bigger, the error is higher. As $k$ goes bigger and the fragments get smaller the SEM minimizes. This is an indication that communication of larger data over the wire may fluctuate the delivery times (as also seen in the file size scenario).

**Changing the number of Reads/Writes Results:** Fig. 3(f) shows a subset of the results of the experiments where the number of read operations changes and the number of write operations is fixed to 60. The experiments show that the total read/write latency (both EC and ABD) has very similar values for all the combinations of writes and reads, which indicates that the system performance is not affected by the number of reads and writes. This is expected since the number of participants is the same in all cases and by well-formedness (i.e., each participant invokes a single operation at a time) at most 10 operations will be concurrent in the execution at any given state. Higher concurrency can be captured by the scalability scenario. Note again that the read latency is higher than the write one, since the read operation actually transfers data twice: once to fetch the data from the servers, and once during the propagation phase.

**Consistency Persistence Results:** Though ARES protocol is probably strongly consistent, it is important to ensure that our implementation is correct. Validating strong consistency of an execution requires precise clock synchronization across all processes, so that one can track operations with respect to a global time. This is impossible to achieve in a distributed system where clock drift is inevitable. To circumvent this, we deploy all the processes in a single beefy machine so that every process observes the same clock running in the same physical machine.

Our checker gathers data regarding an execution, and this data includes start and end times of all the operations, as well as other parameters like logical timestamps used by the protocol. The checker logic is based on the conditions appearing in Lemma 13.16 [38], which provide a set of sufficient conditions for guaranteeing strong consistency. The checker validates strong consistency property for every atomic object individually for the execution under consideration. Note that consistency holds

despite the existence of concurrent read/write and reconfiguration operations that may add/remove servers and switch the storage algorithm in the system.

## 10 CONCLUSIONS

We presented an algorithmic framework suitable for reconfigurable, erasure code-based atomic memory service in asynchronous, message-passing environments. In particular, we provide a new modular framework, called ARES, which abstracts the implementation of the underlying shared memory within a set of DAPs with specific correctness properties. Using these structures, ARES may implement a large class of atomic shared algorithms (those that can be expressed using the proposed DAPs) allowing any such algorithm to work on a reconfigurable environment. A set of erasure-coded-based atomic memory algorithms are included in this class. To demonstrate the use of our framework, we provided a new two-round erasure code-based algorithm that has near optimal storage cost, implemented in terms of the proposed DAPs. Such implementation gave rise to the first (to our knowledge) reconfigurable erasure-coded atomic shared memory object. We provided a proof-of-concept implementation of our framework and obtained initial experimental results proving the feasibility of the presented approach, demonstrating its correctness and comparing its performance with traditional approaches.

ARES is designed to address the real-world problem of system migration from a replicated system to a system that uses erasure codes and vice-versa. ARES can also enable replacing of failed nodes with new non-failed nodes. It's key difference with existing state of the art systems is that it can perform such reconfigurations with relatively minimal interruption of service unlike current implementations that would block ongoing operations for reconfiguration. We anticipate that ARES will be very useful for workloads that are prone to fast changes in properties, and have stringent constraints on the latencies. For such workloads, ARES enables the system to adapt itself to the changes in the workload in an agile manner - utilizing the full flexibility that EC brings to the system - without causing latency constraint or consistency violations due to interruptions.

Our main goal was to establish that non-blocking reconfiguration is feasible and compatible with EC based atomic data storage.Our experimental study is designed as a *proof-of-concept* prototype to verify the correctness properties we have developed, and show some benefits. It must be emphasized that a full-fledged system study of our algorithms - albeit an interesting area of future work that is motivated by our paper - is outside our current scope. In particular, although our study provides some initial hints, we anticipate such a future study would examine the following questions in more detail:

- Real-world applications that would indeed benefit from our design
- Workloads generated from these real-world applications to test our algorithms, and competing ones
- The synergies between our reconfiguration algorithm and existing failure detection and recovery mechanisms.
- Adding efficient repair and reconfiguration using regenerating codes

## REFERENCES

[1] Ansible. https://www.ansible.com/overview/how-ansible-works.
[2] Emulab network testbed. https://www.emulab.net/.
[3] Intel storage acceleration library (open source version). https://goo.gl/zkVl4N.
[4] PyEClib.
[5] PySyncObj.
[6] ZeroMQ.
[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] CACHIN, C., AND TESSARO, S. Optimal resilience for erasure-coded byzantine distributed storage. In *Dependable Systems and Networks, International Conference on* (Los Alamitos, CA, USA, 2006), IEEE Computer Society, pp. 115–124.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] DOBRE, D., KARAME, G. O., LI, W., MAJUNTKE, M., SURI, N., AND VUKOLIÄĞ, M. Proofs of writing for robust storage. *IEEE Transactions on Parallel and Distributed Systems 30*, 11 (2019), 2547–2566.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] HUANG, J., LIANG, X., QIN, X., XIE, P., AND XIE, C. Scale-rs: An efficient scaling scheme for rs-coded storage clusters. *IEEE Transactions on Parallel and Distributed Systems 26*, 6 (2015), 1704–1717.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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).

[36] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

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

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

[39] 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.

[40] 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.

[41] MICHAEL, E., PORTS, D. R. K., SHARMA, N. K., AND SZEKERES, A. Recovering Shared Objects Without Stable Storage. In *31st International Symposium on Distributed Computing (DISC 2017)* (Dagstuhl, Germany, 2017), A. W. Richa, Ed., vol. 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 36:1–36:16.

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

[43] 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).

[44] NICOLAOU, N., CADAMBE, V., PRAKASH, N., KONWAR, K., MEDARD, M., AND LYNCH, N. Ares: Adaptive, reconfigurable, erasure coded, atomic storage. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (2019), pp. 2195–2205.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] WU, C., AND HE, X. Gsr: A global stripe-based redistribution approach to accelerate raid-5 scaling. In *2012 41st International Conference on Parallel Processing* (2012), pp. 460–469.

[51] WU, C., AND HE, X. A flexible framework to enhance raid-6 scalability via exploiting the similarities among mds codes. In *2013 42nd International Conference on Parallel Processing* (2013), pp. 542–551.

[52] 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.

[53] 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.

[54] 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, pp. 1–13.

[55] ZHANG, G., LI, K., WANG, J., AND ZHENG, W. Accelerate rdp raid-6 scaling by reducing disk i/os and xor operations. *IEEE Transactions on Computers 64*, 1 (2015), 32–44.

[56] 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.

[57] ZHANG, X., HU, Y., C. LEE, P. P., AND ZHOU, P. Toward optimal storage scaling via network coding: From theory to practice. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications* (2018), pp. 1808–1816.

[58] 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.