ARES: Adaptive, Reconfigurable, Erasure coded, atomic Storage

Viveck Cadambe * Nicolas Nicolaou † Kishori M. Konwar ‡ N. Prakash ‡
Nancy Lynch ‡ Muriel Médard ‡

May 11, 2018

Abstract

Atomicity or strong consistency is one of the fundamental, most intuitive, and hardest to provide primitives in distributed shared memory emulations. To ensure survivability, scalability, and availability of a storage service in the presence of failures, traditional approaches for atomic memory emulation, in message passing environments, replicate the objects across multiple servers. Compared to replication based algorithms, erasure code-based atomic memory algorithms has much lower storage and communication costs, but usually, they are harder to design. The difficulty of designing atomic memory algorithms further grows, when the set of servers may be changed to ensure survivability of the service over software and hardware upgrades (scale-up or scale-down), while avoiding service interruptions. Atomic memory algorithms for performing server reconfiguration, in the replicated systems, are very few, complex, and are still part of an active area of research; reconfigurations of erasure-code based algorithms are non-existent.

In this work, we present ARES, an algorithmic framework that allows reconfiguration of the underlying servers, and is particularly suitable for erasure-code based algorithms emulating atomic objects. ARES introduces new configurations while keeping the service available. To use with ARES we also propose a new, and to our knowledge, the first two-round erasure-code based algorithm TREAS, for emulating multi-writer, multi-reader (MWMR) atomic objects in asynchronous, message-passing environments, with near-optimal communication and storage costs. Our algorithms can tolerate crash failures of any client and some fraction of servers, and yet, guarantee safety and liveness property. Moreover, by bringing together the advantages of ARES and TREAS, we propose an optimized algorithm where new configurations can be installed without the objects values passing through the reconfiguration clients.

*EE Department, Pennsylvania State University, University Park, PA, USA, vxc12@engr.psu.edu

†Algolysis Ltd & KIOS Research and Innovation Center of Excellence, U. of Cyprus, Nicosia, Cyprus, nicolasn@cs.ucy.ac.cy

‡Department of EECS, Massachusetts Institute of Technology, Cambridge MA, USA, kishori@csail.mit.edu, prakashn@mit.edu, lynch@csail.mit.edu, medard@mit.edu

1 Introduction

With the rapid increase of computing power on portable devices, such as, smartphone, laptops, tablets, etc., and the near ubiquitous availability of Internet connectivity, our day to day lives are becoming increasingly dependent on Internet-based applications. Most of these applications, rely on large volumes of data from a wide range of sources, and their performance improves with the easy accessibility of the data. Today, data is gathered at an even faster pace from numerous sources of interconnected devices around the world. In order to keep abreast with this veritable tsunami of data, researchers, in both industry and academia, are hurtling to invent new ways to increase the capacity of durable, large-scale distributed storage systems, and the efficient ingestion and retrieval of data. Currently, most of the data is stored in cloud-based storages, offered by major providers like Amazon, Dropbox, Google, etc. To 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 generating replicating data across multiple servers, but this creates the headache of keeping those copies consistent, especially when they can be updated concurrently by different operations.

Atomic Distributed Storage. To solve this problem, researches developed a series of consistency semantics, that formally specify the behavior of concurrent operations. Atomicity or strong consistency is the strongest and most intuitive consistency semantic which provides the illusion that a data object is accessed sequentially, even when operations may access different copies of that data object concurrently. In addition, atomic objects are composable [17, 24], enabling the creation of large shared memory systems from individual atomic data objects. Such large-scale shared memory services make application development much simpler. Finally, the strongest and most desirable form of availability or liveness of atomic data access is wait-freedom [24] where operations complete irrespective of the speed of the clients.

Replication-based Atomic Storage. A long stream of works used replication of data across multiple servers to implement atomic read/write storage [3, 4, 9, 10, 11, 13, 14, 26]. Popular replication-based algorithms appear in the work by Attiya, Bar-Noy and Dolev [4] (we refer to this as the ABD algorithm) and also in the work by Fan and Lynch [10] (which is referred to as the LDR algorithm). 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 1 TB 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 3 TB. Additionally, every write or read operation may need to transmit at most 3 MB of data, incurring high communication cost.

Erasure Code-based Atomic Storage. To avoid the high storage and communication costs stemmed from the use of replication, erasure codes provide an alternative way to emulate fault-tolerant shared atomic storage. In comparison to replication, algorithms based on erasure codes significantly reduce both the storage and communication costs of the implementation. An $[n, k]$ erasure code splits the value v of size 1 unit into k elements, each of size $\frac{1}{k}$ units, creates n coded elements, and stores one coded element per server. The size of each coded element is also $\frac{1}{k}$

units, and thus the total storage cost across the n servers is $\frac{n}{k}$ units. For example, if we use an $[n = 3, k = 2]$ MDS code, the storage cost is simply 1.5 TB, which is 2 times lower than the storage in the case of ABD. A similar reduction in bandwidth per operation is possible in many erasure code-based algorithms for implementing atomicity. A class of erasure codes known as Maximum Distance Separable (MDS) codes have the property that value v can be reconstructed from any k out of these n coded elements. Motivated by these approaches, recently, several ensure code based algorithms for implementing strong consistency on message-passing models have been proposed in theory [5, 6, 8, 20, 22], and in practice [7, 29]. However, the leap from replication-based to erasure code-based algorithms for strong consistency comes with the additional burden of synchronizing the access of multiple pieces of coded-elements from the same version of the the data object. This naturally leads to relatively complex algorithms.

Reconfigurable Atomic Storage. Although replication and erasure-codes may help the system survive the failure of a subset of servers, they do not suffice to ensure the liveness of the service in a longer period where a larger number of servers may fail. Reconfiguration is the process that allows addition or removal of servers from a live system, without affecting its normal operation. However, performing reconfiguration of a system, without service interruption, is very challenging and not well-understood even in replicated systems, and is still and active area of research. RAMBO [25] and DynaStore [1] are two of the handful of algorithms [12, 15, 19, 27] that allows reconfiguration on live systems. Recently, the authors in [28] presented a general algorithmic framework for consensus-free reconfiguration algorithms.

So far, all reconfiguration approaches, implicitly or explicitly, assume a replication-based system in each configuration. Therefore, none of the above methods can reconfigure from or to a configuration where erasure codes are used. In particular, erasure code based atomic memory algorithms requires a fixed set of servers in a configuration, therefore, moving from an existing coding scheme would entail deploying a new set of servers, or change the set of servers in an existing configuration. In RAMBO [25], messages originating from a read or write operation are sent as part of an ongoing gossiping protocol, and this makes counting the communication cost of each operation obscure. In both RAMBO [25] and DynaStore [1], the clients and servers are combined, and therefore, not immediately suitable for settings where clients and servers are separate processes, a commonly studied architecture, both in theory [24] and in practice [23]. In DynaStore [1] and [28], an active new configuration, generated by the underlying SpSn algorithm, may often consist of a set of servers proposed during configuration proposed by multiple clients. They assume that as long as a majority of the servers in the active configurations are non- faulty the service can guarantee liveness. In erasure code based algorithms additional assumptions are required. For example, if some client proposes a configuration with certain MDS $[n, k]$ code then If we have more than n servers in the configuration increases the storage cost, while having fewer than n servers will not permit using the proposed code parameters. Moreover, in the algorithms in [1] and [28], the configurations proposed by the clients are incremental changes, e.g., $\{-s_1, -s_2, +s_3\}$, where $-$ is a suggestion to remove a server and $+$ a suggestion to add it. Therefore, some additional mechanism is necessary for the reconfiguration client to know the total number of existing active servers before proposing its change as an attempt to generate a configuration of a desired size.

Reconfigurations are usually desirable to system administrators [2], usually during system maintenance. Therefore, during the migration of the objects, from one configuration to the next, it is highly likely that all stored objects are moved to the newer configuration almost at the same time. In the above algorithms, data transfer is done, by using the client as a conduit, creating a possible bottleneck. Although, setting proxy servers for reconfiguration is an ad hoc solution it suffers from a the same bottleneck. In such situations, it is more reasonable for the data objects to migrate directly from one set of servers to another.

Our Contributions. In this work, we present ARES, an algorithm that allows reconfiguration of the servers that emulates an atomic memory, specifically suitable for atomic memory service that uses erasure codes, while keeping the service available at all times. In order to keep ARES general, so as to allow adaptation of already known atomic memory algorithms to the configurations, we introduced a new set of data primitives, that (i) provides a modular framework to describe atomic read/write storage implementations, and (ii) enhances the ease for reasoning about algorithm's correct implementation of atomic memory.Using these primitives, we are able to prove the safety property (atomic memory) of an execution of ARES that involves ongoing reconfiguration operations. We also present TREAS, the first two-round erasure code-based MWMR algorithm, with cost-effective storage and communication costs, for emulating shared atomic memory under message-passing environment in the presence of crash-failure of processes. We prove safety and liveness conditions for TREAS. Then we describe a new algorithm ARES-TREAS, where we use a modified version of TREAS as the underlying atomic memory algorithm in every configuration and modify ARES, so that data from one configuration is transferred to another directly, thereby, avoiding the reconfiguration client as the possible bottleneck.

Document Structure. The remainder of the manuscript consists of the following sections. In Section 2, we present the model assumptions for our setting. In Section 3, we present our two-round erasure code-based algorithm for emulating shared atomic storage under the message-passing model for MWMR setting. In Section 4, we present our ARES framework for emulating shared atomic memory with erasure-codes where the system can undergo reconfiguration, while it is live. Sub-section 4.4 provides latency analysis read, write and reconfiguration operations. In Section 5, we describe ARES-TREASalgorithm. Finally, in Section 6, we conclude our work.

2 Model and Definitions

A shared atomic storage can be emulated by composing individual atomic objects. Therefore, we aim to implement only one atomic read/write memory object on a set of servers. Each data 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 modify the set of servers in the system in order to mask transient errors and to ensure the longevity of the service. Processes communicate via messages through asynchronous, reliable channels. ARES allows the set of server host to be modified during the course of an execution for masking transient or permanent failures of servers and preserve the longevity of the service.

Configuration A configuration, identified by a unique identifier c, is a data type that describes explicitly: (i) the set identifiers of the set of servers that are a part of it, denote as $c.Servers$; (ii)

the set of quorums that are defined on $c.Servers$; (iii) an underlying algorithm that implements atomic memory in $c.Servers$, including related parameters; and (iv) a consensus instance, $c.Con$, with the values from $\mathcal{C}$, the set of all unique configuration identifiers, implemented on top of the servers in $c.Servers$. We refer to a server $s \in c.Servers$ as a member of configuration c.

Liveness and Safety Conditions. The algorithms we present in this paper satisfy wait-free termination (Liveness) and atomicity (Safety). Wait-free termination [16] requires that any process terminates in a finite number of steps, independent of the progress of any other process in the system.

An implementation A of a read/write object satisfies the atomicity property if the following holds [24]. Let the set Π contain all complete operations in any well-formed execution of A. Then for operations in Π there exists an irreflexive partial ordering $\prec$ satisfying the following:

A1. For any operations π_1 and π_2 in Π, 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 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 π invoked by a process p, is measured with respect to three metrics, during the interval between the invocation and the response of π: (i) communication round-trips, accounting the number of messages sent during π, (ii) storage efficiency (storage cost), accounting the maximum storage requirements for any single object at the servers during π, and (iii) message bit complexity (communication cost) which measures the length of the messages used during π.

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 the 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.

Background on Erasure coding. In TREAS, we use an $[n, k]$ linear MDS code [18] 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 $c_1, c_2, \ldots, c_n$ as output, i.e., $[c_1, \ldots, c_n] = \Phi([v_1, \ldots, v_k])$, where Φ denotes the encoder. For ease of notation, we simply write $\Phi(v)$ to mean $[c_1, \ldots, c_n]$. The vector $[c_1, \ldots, c_n]$ is referred to as the codeword corresponding

*In practice v is a file, which is divided into many stripes based on the choice of the code, various stripes are individually encoded and stacked against each other. We omit details of represent-ability of v by a sequence of symbols of $\mathbb{F}_q$, and the mechanism of data striping, since these are fairly standard in the coding theory literature.

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 Φ_i to denote the projection of Φ on to the i^{th} output component, i.e., $c_i = \Phi_i(v)$. Without loss of generality, we associate the coded element c_i with server i, $1 \leq i \leq n$.

Tags. A tag τ 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 $\tau_2.w > \tau_1.w$.

2.1 The Data-Access Primitives

In the next section, we present the TREAS algorithm using three *data access primitives* (DAP) which can be associated with a configuration c. These data access primitives in the context of c are: (i) $c.\text{put-data}(\langle\tau, v\rangle)$, (ii) $c.\text{get-data}()$, and (iii) $c.\text{get-tag}()$. Assuming a set of totally ordered timestamps $\mathcal{T}$, a value domain of the distributed atomic object $\mathcal{V}$, and a set of configuration identifiers $\mathcal{C}$, the three primitives defined over a configuration $c \in \mathcal{C}$, tag $\tau \in \mathcal{T}$, and a value $v \in \mathcal{V}$ as follows:

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

D1. $c.\text{get-tag}()$*: returns a tag* $\tau \in \mathcal{T}$

D2. $c.\text{get-data}()$*: returns a tag-value pair* $(\tau, v) \in \mathcal{T} \times \mathcal{V}$

D3. $c.\text{put-data}(\langle\tau, v\rangle)$*: the tag-value pair* $(\tau, v) \in \mathcal{T} \times \mathcal{V}$ *as argument.*

Algorithm 1 Read and write operations of generic algorithm A_1

operation read()
  $\langle t, v\rangle \leftarrow c.\text{get-data}()$
  $c.\text{put-data}(\langle t, v\rangle)$
  return $\langle t, v\rangle$
end operation

operation write(v)
  $t \leftarrow c.\text{get-tag}()$
  $t_w \leftarrow inc(t)$
  $c.\text{put-data}(\langle t_w, v\rangle)$
end operation

A number of known tag-based algorithms that implement atomic read/write objects (e.g., ABD, FAST – see Appendix A.1), can be expressed in terms of DAP. In particular, many algorithms can be transformed in an algorithmic template, say A_1, presented in Alg. 10. In brief, a read operation in A_1 performs $c.c.\text{get-data}()$ to retrieve a tag-value pair, $\langle\tau, v\rangle$ form configuration c, and then it performs a $c.\text{put-data}(\langle\tau, v\rangle)$ to propagate that pair to 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. It can be shown (see Appendix A) that an algorithm described in the form A_1 satisfies atomic guarantees and liveness, if the DAP satisfy the following consistency properties:

Definition 2 (DAP Consistency Properties). *In an execution ξ we say that a DAP operation in an execution ξ is complete if both the invocation and the matching response step appear in ξ. If Π is the set of complete DAP operations in execution ξ then for any $\phi, \pi \in \Pi$:*

C1 *If ϕ is c.*put-data($\langle\tau_\phi, v_\phi\rangle$)*, for $c \in \mathcal{C}$, $\tau_1 \in \mathcal{T}$ and $v_1 \in \mathcal{V}$, and π 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 ϕ completes before π in ξ, then $\tau_\pi \geq \tau_\phi$.*

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

Expressing an atomic algorithm in terms of the DAP primitives allows one to achieve a modular design for atomic object tag-based algorithms, serving multiple purposes. First, describing an algorithm according to templates A_1 (like TREAS in Section 3) allows one to proof that the algorithm is *safe* (atomic) by just showing that the appropriate DAP properties hold, and the algorithm is *live* if the implementation of each primitive is live. Second, the safety and liveness proofs for more complex algorithms (like ARES in Section 4) become easier as one may reason on the DAP properties that are satisfied by the primitives used, without involving the underlying implementation of those primitives. Last but not least, describing a reconfigurable algorithm using DAPs, provides the flexibility to use different implementation mechanics for the DAPs in each reconfiguration, as long as the DAPs used in a single configuration satisfy the appropriate DAP properties. Hence, makes our algorithm adaptive.

3 TREAS: A new two-round erasure-code based algorithm

In this section, we present the first, two-round, erasure-code based algorithm for implementing atomic memory service, we call TREAS. The algorithm uses $[n, k]$ MDS codes for storage. We implement and instance of the algorithm in a configuration of n server processes.

The read and write operations of algorithm TREAS are implemented using A_1 (Alg. 10), the DAP primitives are implemented in Alg. 2, and the servers' responses in Automaton 3. In high level, both the read and write operations take two phases to complete (similar to the ABD algorithm). As in algorithm A_1, a write operation π, discovers the maximum tag t^* from a quorum in $c.Quorums$ by executing c.get-tag(); creates a new tag $t_w = tag(\pi) = (t^*.z + 1, w)$ by incorporating the writer's own ID; and it performs a c.put-data($\langle t_w, v\rangle$) to propagate that pair to configuration c. A read operation performs c.get-data() to retrieve a tag-value pair, $\langle\tau, v\rangle$ form configuration c, and then it performs a c.put-data($\langle\tau, v\rangle$) to propagate that pair to the servers $c.Servers$.

To facilitate the use of erasure-codes, 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, $List = \{(t_0, \Phi_i(v_0)\}$. Given this set we can now describe the implementation of the DAP.

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, with $k \geq \frac{2n}{3}$. 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 to c.

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. Every time a (PUT-DATA, t_w, c_i) message arrives at a server s_i from a writer, the pair gets added to the $List$. As the size of the $List$ at each s_i is bounded by $(\delta + 1)$, then following an insertion in the $List$, s_i trims the coded-elements associated with the smallest tags. In particular,

Algorithm 2 The protocols for the DAP primitives for template A_1 to implement TREAS.

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

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

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

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

Algorithm 3 The response protocols at any server $s_i \in \mathcal{S}$ in TREAS for client requests.

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

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

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

Upon receive (QUERY-LIST) $_{s_i, c_k}$ **from** q
8: Send $List$ to q
end receive
10:
Upon receive (PUT-DATA, $\langle \tau, e_i \rangle$) $_{s_i, c_k}$ **from** q
12: $List \leftarrow List \cup \{\langle \tau, e_i \rangle\}$
 if $|List| > \delta + 1$ **then**
14: $\tau_{min} \leftarrow \min\{t : \langle t, * \rangle \in List\}$

 // remove the coded value and retain the tag
 $List \leftarrow List \backslash \{\langle \tau, e \rangle : \tau = \tau_{min} \wedge \langle \tau, e \rangle \in List\} \cup \{(\tau_{min}, \bot)\}$
16: Send ACK to q
end receive

s_i replaces the coded-elements of the older tags with $\bot$, and maintains only the coded-elements associated with the $(\delta + 1)$ highest tags in the $List$ (see Line Alg. 3:15). The client completes the primitive operation after getting acks from $\left\lceil \frac{n+k}{2} \right\rceil$ servers.

A client during the execution of a c.get-data() primitive, it 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. 2:11-14) 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.

Storage and Communication Costs for TREAS**.** We now briefly present the storage and communication costs associated with TREAS. Due to space limitations the proofs appear in Appendix B. Recall that by our assumption, the storage cost counts the size (in bits) of the coded elements stored in the $List$ variable at each server. We ignore the storage cost due to meta-data and temporary variables. Also, for the communication cost we measure the bits sent on the wire between the nodes.

Theorem 3. *The* TREAS *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 to* $(\delta + 2)\frac{n}{k}$*.*

3.1 Safety, Liveness and Performance cost of TREAS

In this section we are concerned with only one configuration c, consisting of a set of servers $\mathcal{S}$, and a set of reader and writer clients $\mathcal{R}$ and $\mathcal{W}$, respectively. In other words, in such static system the sets $\mathcal{S}$, $\mathcal{R}$ and $\mathcal{W}$ are fixed, and at most $f \leq \frac{n-k}{2}$ servers may crash fail. Below we prove Lemma 5, which proves the consistency properties of the DAP implementation of TREAS, which implies the atomicity city properties.

3.1.1 Correctness and Liveness

Now we can show that if DAP properties are satisfied from the DAP, then algorithm A_1 implements an atomic read/write algorithm. We can show that A_1 satisfy atomic guarantees and liveness if the DAP in the above algorithms satisfy the DAP consistency properties.

Theorem 4 (Atomicity of A_1)**.** *Suppose the DAP implementation satisfies the consistency properties $C1$ and $C2$ of Definition 31. Then any execution ξ the atomicity protocols A_1 on a configuration $c \in \mathcal{C}$, is atomic and live if DAPs are live in ξ.*

Lemma 5. *The data-access primitives, i.e.,* get-tag, get-data *and* put-data *primitives, implemented in the* TREAS *algorithm satisfy the consistency properties.*

Theorem 6 (Atomicity)**.** *Any execution of* TREAS*, is atomic.*

The parameter δ captures all the write operations that overlap with the read, until the time the reader has all data needed to attempt decoding a value. However, we ignore those write operations that might have started in the past, and never completed yet, if their tags are less than that of any write that completed before the read started. This allows us to ignore write operations due to failed writers, while counting concurrency, as long as the failed writes are followed by a successful write that completed before the read started.

Definition 7 (Valid read operations)**.** *A read operation π will be called as a valid read if the associated reader remains alive until the reception of the $\lceil \frac{n+k}{2} \rceil$ responses during the get-data phase.*

Definition 8 (Writes concurrent with a valid read)**.** *Consider a valid read operation π. Let T_1 denote the point of initiation of π. For π, let T_2 denote the earliest point of time during the execution when the associated reader receives all the $\lceil \frac{n+k}{2} \rceil$ responses. Consider the set $\Sigma = \{\sigma : \sigma$ is any write operation that completes before π is initiated$\}$, and let $\sigma^* = \arg\max_{\sigma \in \Sigma} tag(\sigma)$. Next, consider the set $\Lambda = \{\lambda : \lambda$ is any write operation that starts before T_2 such that $tag(\lambda) > tag(\sigma^*)\}$. We define the number of writes concurrent with the valid read operation π to be the cardinality of the set Λ.*

The above definition captures all the write operations that overlap with the read, until the time the reader has all data needed to attempt decoding a value. However, we ignore those write operations that might have started in the past, and never completed yet, if their tags are less than that of any write that completed before the read started. This allows us to ignore write operations due to failed writers, while counting concurrency, as long as the failed writes are followed by a successful write that completed before the read started.

Theorem 9 (Liveness). *Let β denote a well-formed execution of* TREAS*, with $[n, k]$, where n is the number of servers and $k > n/3$, and δ be the maximum number of write operations concurrent with any valid read operation then the read and write operations β are live.*

4 Algorithm Framework for ARES

In this section, we provide the description of an atomic reconfigurable read/write storage, we call ARES. In the presentation of ARES algorithm we decouple the reconfiguration service from the shared memory emulation, by utilizing the data access primitives presented in Section 2.1. 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.

In the rest of the section we first provide the specification of the reconfiguration mechanism used in ARES, along with the properties that this service offers. Then, we discuss the implementation of read and write operations and how they utilize the reconfiguration service to ensure atomicity even in cases where read/write operations are concurrent with reconfiguration operations. The read and write operations are described in terms of the data access primitives presented in Section 2.1 and we show that if the DAP properties are satisfied then ARES preserves atomicity. This allows ARES to deploy the transformation of any atomic read/write algorithm in terms of the presented DAPs without compromising consistency.

4.1 Implementation of the Reconfiguration Service

In this section, we describe the reconfiguration service that is used in ARES, where reconfig clients introduce new configurations. In our setting, we assume throughout an execution of ARES, every configuration is attempted to be reconfigured at most once. Multiple clients may attempt concurrently to introduce a different configuration for the same index i in the configuration sequence. ARES uses consensus to resolve such conflicts. In particular, each configuration c is associated with an external consensus service, denoted by $c.Con$, that runs on a subset of servers in the configuration. We use the data-type $status \in \{F, P\}$, corresponding to a configuration, say c, to denote whether c is finalized (F) or is still pending (P). Each reconfigurer may change the system configuration by introducing a new configuration identifier. The data type configuration sequence is an array of pairs $\langle cfg, status\rangle$, where $c \in \mathcal{C}$ and $status \in \{F, P\}$. We denote each such pair by the caret over a variable name, e.g., $\widehat{x}$ or $\widehat{config}$, or $\hat{c}$, etc.

The service relies on an underlying sequence of configurations in the from of a "distributed list", global configuration sequence $\mathcal{G}_L$. In any configuration c, every server in $c.Servers$ has a configuration sequence variable $cseq$, initially $\langle c_0, F\rangle$, where new configurations can be added to the end of the list. We use the notation $|\hat{c}|$ to denote the length of the array. Every server in $c.Servers$ has a variable $nextC$, $nextC \in \mathcal{C} \cup \{\bot\}$. Initially, at any server $nextC = \bot$, and once it it set to a value in $\mathcal{C}$ it is never altered. For any $c \in \mathcal{C}$, at any point in time, all the values of $nextC$, such that $nextC \neq \bot$, in the processes in $c.Servers$ are the same. At any point in an execution of ARES, for any $c_i, c_j \in \mathcal{C}$, we say c_i points c_j is a link in $\mathcal{G}_L$ if at that point in the execution where a server in $c_j.Servers$ has $nextC = c_j$.

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

Algorithm 4 Sequence traversing at each process $p \in \mathcal{I}$ of algorithm ARES.

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

procedure read-next-config(c)
14: **send** (READ-CONFIG) to each $s \in c.Servers$
until $\exists Q, Q \in c.Quorums$ s.t. rec_i receives $nextC_s$ from $\forall s \in Q$
16: **if** $\exists s \in Q$ s.t. $nextC_s.status = F$ **then**
return $nextC_s$
18: **else if** $\exists s \in Q$ s.t. $nextC_s.status = P$ **then**
return $nextC_s$
20: **else**
return $\perp$
22: **end procedure**

procedure put-config($c, nextC$))
24: **send** (WRITE-CONFIG, $cfgPtr$) to each $s \in c.Servers$
until $\exists Q, Q \in c.Quorums$ s.t. rec_i receives ACK from $\forall s \in Q$
26: **end procedure**

Sequence Traversal. Any *read/write/reconfig* operation π 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 π'. The sequence parsing consists of three actions: (i) get-next-config(), (ii) put-config(), and (iii) read-config(). We do present their specification and implementations as follows (Fig. 4):

get-next-config(c): The action get-next-config returns the configuration that follows c in $\mathcal{G}_L$. During get-next-config(c) action sends READ-CONFIG messages to all the servers in $c.Servers$. Once a server receives such a message responds with the value of its $nextC$ variable variable. Once it receives replies from a quorum in $c.Quorums$, then if there exists a reply that contains a $nextC \neq \perp$ the action returns $nextC$; otherwise it returns $\perp$.

put-config(c, c'): The put-config(c, c') action propagates c' to the 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 configurations in $\mathcal{G}_L$ in order to discover the latest state of the sequence $\mathcal{G}_L$. At invocation, the client starts with last finalized configuration c_μ in seq (Line A4:2), say c and enters a loop to traverse $\mathcal{G}_L$ by invoking get-next-config(c), which returns the next configuration, say c'. If $c' \neq \perp$, then: (a) c' is appended at the end of the sequence seq; (b) a put-config(c, c_r) is invoked to inform a quorum of servers in $c.Servers$ to update the value of their $nextC$ variable to c_r; and (c) variable c is set to c_r. If $c' = \perp$ the loop terminates the action read-config returns seq.

Server Protocol. Each server responds to requests from clients (Alg. 6). 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 $nextC.c_k$ variable of the server with a particular tuple $nextT_{in}$. A server changes the value of its local $nextC.c_k$ in two cases: (i) $nextC.c_k = \perp$, or (ii) $nextC.status = P$.

Algorithm 5 Reconfiguration protocol of algorithm ARES.

at each reconfigurer rec_i
State Variables:
$cseq[]s.t.cseq[j] \in \mathcal{C} \times \{F, P\}$ with members:
Initialization:
$cseq[0] = \langle c_0, F \rangle$

operation reconfig(c)
  if $c \neq \bot$ **then**
    $cseq \leftarrow$read-config$(cseq)$
    $cseq \leftarrow$ add-config$(cseq, c)$
    update-config$(cseq)$
    $cseq \leftarrow$ finalize-config$(cseq)$
end operation

procedure add-config(seq, c)
  $\nu \leftarrow |seq|$
  $c' \leftarrow seq[\nu].cfg$
  $d \leftarrow c'.Con.propose(c)$
  $seq[\nu+1] \leftarrow \langle d, P \rangle$
  put-config$(c', \langle d, P \rangle)$
  return seq'
end procedure

procedure update-config(seq)
  $\mu \leftarrow \max(\{j : seq[j].status = F\})$
  $\nu \leftarrow |seq|$
  $M \leftarrow \emptyset$
  for $i = \mu : \nu$ **do**
    $\langle t, v \rangle \leftarrow seq[i].cfg$.get-data()
    $M \leftarrow M \cup \{\langle \tau, v \rangle\}$
  $\langle \tau, v \rangle \leftarrow \max_t\{\langle t, v \rangle : \langle t, v \rangle \in M\}$
  $seq[\nu]$.put-data$(\langle \tau, v \rangle)$
end procedure

procedure finalize-config(seq)
  $\nu = |seq|$
  $seq[\nu].status \leftarrow F$
  put-config$(seq[\nu-1].cfg, seq[\nu])$
  return seq
end procedure

Algorithm 6 Server protocol of algorithm ARES.

at each server s_i in configuration c_k
State Variables:
$\tau \in \mathbb{N} \times \mathcal{W}$, initially, $\langle 0, \bot \rangle$
$v \in V$, intially, $\bot$
$nextC \in \mathcal{C} \times \{P, F\}$, initially $\langle \bot, P \rangle$

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 $nextC.cfg = \bot \vee nextC.status = P$ **then**
    $nextC \leftarrow cfgT_{in}$
  send ACK to q
end receive

Reconfiguration operation. A reconfiguration operation reconfig(c), $c \in \mathcal{C}$, invoked by a non-faulty reconfiguration client rec_i, attempts to append c to $\mathcal{G}_L$. The operation consists of the following phases, executed consecutively by rec_i: (i) read-config; (ii) add-config; (iii) update-config and (iv) finalize-config.

read-config(seq): The phase read-config(seq) at rec_i, reads the recent global configuration sequence starting with some initial guess of seq. As described above, the read-config action completes the traversal by returning a possibly extended configuration sequence to $cseq$.

add-config(seq, c): The add-config(seq, c) attempts to append a new configuration c to the end of seq (the approximation of $\mathcal{G}_L$). Suppose the last configuration in seq is c', then in order to decide the most recent configuration, rec_i invokes $c'.Con.propose(c)$, on the consensus object associated with configuration c', where d is the decided configuration identifier returned the configuration 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 μ the index of the last configuration in $cseq$, at rec_i, such that its corresponding status is F; and ν denote the last index of $cseq$. Next rec_i invokes update-config(seq), 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

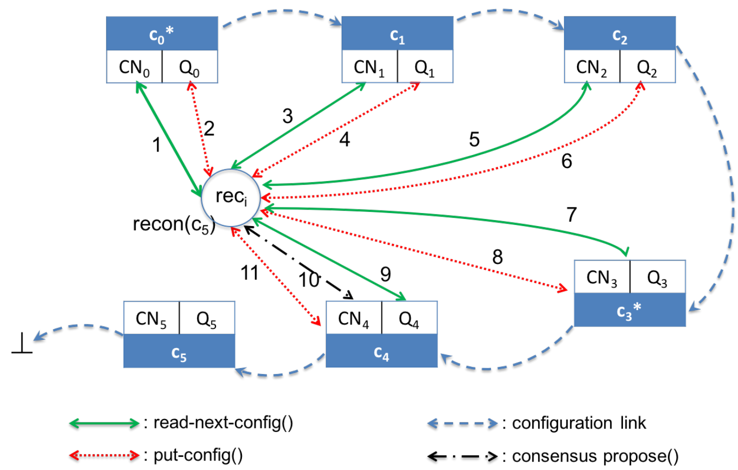

Figure 1: An illustration of an execution of the reconfiguration process.

was added by the add-config action. The get-data and put-data actions are implemented with respect to the atomic algorithm that is used in each of the configurations that are 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 $seq[\nu].cfg.$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 $cseq$, i.e. d=$cseq[\nu]$, to F. rec_i informs a quorum of servers in the previous configuration, 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 illustrates an example execution of a reconfiguration operation recon(c_5). In this example, the reconfigurer rec_i goes through a number of configuration queries (read-next-config) before it reaches configuration c_4 in which a quorum of servers replies with $nextC.cfg = \perp$. There it proposes c_5 to the consensus object of c_4 ($c_4.Con.propose(c_5)$ on arrow 10), and once c_5 is decided, recon(c_5) completes after executing finalize-config(c_5).

Read and Write operations. The read and write operations in Ares are expressed in terms of the DAP primitives (see Section 3). A read consists of an execution of get-data primitive followed by a put-data primitive, while a write consists of calls to get-tag and put-data primitives. This provides the flexibility to Ares to use different implementation of DAP primitives, without changing the Ares framework. 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, 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. Now we describe the execution of a read or write operation π in more detail.

In line Alg. 7:8 for the writer (or line Alg. 7:31 for reader), a write (or read) operation is invoked at a client p, then p issues a read-config action to obtain the latest introduced configuration in $\mathcal{G}_L$.

In lines Alg. 7:9 if π is a write (resp. Alg. 7:32 if π is a read), p detects the last finalized entry in

Algorithm 7 Write and Read protocols at the clients for ARES.

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

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

24: **Read Operation:**
at each reader r_i
26: **State Variables:**
$cseq[]s.t.cseq[j] \in \mathcal{C} \times \{F, P\}$ with members:
28: **Initialization:**
$cseq[0] = \langle c_0, F \rangle$

30: **operation** read( )
$cseq \leftarrow$read-config($cseq$)
32: $\mu \leftarrow \max(\{j : cseq[j].status = F\})$
$\nu \leftarrow |cseq|$
34: **for** $i = \mu : \nu$ **do**
$\langle \tau, v \rangle \leftarrow \max(cseq[i].cfg.\text{get-data}(), \langle \tau, v \rangle)$
36: $done \leftarrow$ **false**
while not $done$ **do**
38: $cseq[\nu].cfg.\text{put-data}(\langle \tau, v \rangle)$
$cseq \leftarrow$read-config($cseq$)
40: **if** $|cseq| = \nu$ **then**
$done \leftarrow true$
42: **else**
$\nu \leftarrow |cseq|$
44: **end while**
end operation

$cseq$, say μ, and performs a $cseq[j].conf.$get-tag() action if π is a *write*, or $cseq[j].conf.$get-data() action if π is a *read* action, for $\mu \leq j \leq |cseq|$. Then p discovers the maximum tag among all the returned tags (τ_{max}) or tag-value pairs ($\langle \tau_{max}, v_{max} \rangle$) respectively. If π is a *write*, p increments the maximum tag discovered (by incrementing the integer part of τ_{max}), generates a new tag, say τ_{new}, and assigns $\langle \tau, v \rangle$ to $\langle \tau_{new}, val \rangle$, where val is the value he wants to write (Line Alg. 7:13). If π is a *read*, then p assigns $\langle \tau, v \rangle$ to $\langle \tau_{max}, v_{max} \rangle$, i.e., the maximum discovered tag-value pair.

In lines Alg. 7:15–21 if π is a write, or lines Alg. 7:37–43 if π is a read, p repeatedly executes 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 $\mathcal{G}_L$. 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 $cseq' = cseq''$.

4.2 Properties of Reconfiguration

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 σ of an execution ξ. If server s crashes at a state σ_f in an execution ξ then $s.var|_{\sigma} \triangleq s.var|_{\sigma_f}$ for any state variable var and for any state σ that appears after σ_f in ξ.

Definition 10 (Tag of a configuration)**.** *Let $c \in \mathcal{C}$ be a configuration, σ be a state in some execution ξ then we define the tag of c at state σ as $tag(c)|_{\sigma} \triangleq \min_{Q \in c.Quorums} \max_{s \in Q} \ (s.tag|_{\sigma})$. We drop the suffix $|_{\sigma}$, and simply denote as $tag(c)$, when the state is clear from the context.*

Definition 11. *Let* $\mathbf{c}_\sigma^p = p.cseq|_\sigma$. *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 number of elements in configuration vector* $\mathbf{c}_\sigma^p$ *that are not equal to* $\perp$.

Definition 12 (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* $\mathbf{x}[j] \neq \perp$.

Next we analyze some properties of ARES. The first lemma shows that any two configuration sequences have the same configuration identifiers in the same indexes.

Lemma 13 (Configuration Uniqueness)**.** *For any processes* $p, q \in \mathcal{I}$ *and any states* σ_1, σ_2 *in an execution* ξ, *it must hold that* $\mathbf{c}_{\sigma_1}^p[i].cfg = \mathbf{c}_{\sigma_2}^q[i].cfg$, $\forall i$ *s.t.* $\mathbf{c}_{\sigma_1}^p[i].cfg, \mathbf{c}_{\sigma_2}^q[i].cfg \neq \perp$.

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 (Configuration Prefix)**.** *Let* π_1 *and* π_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* ξ. *Let* σ_1 *be the state after the response step of* π_1 *and* σ_2 *the state after the response step of* π_2. *Then* $\mathbf{c}_{\sigma_1}^{p_1} \preceq_p \mathbf{c}_{\sigma_2}^{p_2}$.

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 15 (Configuration Progress)**.** *Let* π_1 *and* π_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* ξ. *Let* σ_1 *be the state after the response step of* π_1 *and* σ_2 *the state after the response step of* π_2. *Then* $\mu(\mathbf{c}_{\sigma_1}^{p_1}) \leq \mu(\mathbf{c}_{\sigma_2}^{p_2})$.

Theorem 16. *Let* π_1 *and* π_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* ξ. *Let* σ_1 *be the state after the response step of* π_1 *and* σ_2 *the state after the response step of* π_2. *Then the following properties hold:* (a) $\mathbf{c}_{\sigma_2}^{p_2}[i].cfg = \mathbf{c}_{\sigma_1}^{p_1}[i].cfg$, *for* $1 \leq i \leq \nu(\mathbf{c}_{\sigma_1}^{p_1})$, (b) $\mathbf{c}_{\sigma_1}^{p_1} \preceq_p \mathbf{c}_{\sigma_2}^{p_2}$, *and* (c) $\mu(\mathbf{c}_{\sigma_1}^{p_1}) \leq \mu(\mathbf{c}_{\sigma_2}^{p_2})$

4.3 ARES Safety

Once we showed some properties that are satisfied by the reconfiguration service 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 primitives used satisfy properties $C1$ and $C2$ as presented in Section 2.1, and we will show that, given such assumption, ARES satisfies atomicity.

We begin with a lemma that states 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 $\mathcal{G}_L$.

Lemma 17. *Let π be a complete reconfiguration operation by a reconfigurer rc in an execution ξ of* ARES*. if σ_1 is the state in ξ following the termination of the* read-config *action during π, then π invokes a* finalize-config$(\mathbf{c}^{rc}_{\sigma_2})$ *at a state σ_2 in ξ, with $\nu(\mathbf{c}^{rc}_{\sigma_2}) = \nu(\mathbf{c}^{rc}_{\sigma_1}) + 1$.*

The next lemma states that each finalized configuration c at index j in a configuration sequence $p.cseq$ at any process p, is finalized by some reconfiguration operation ρ. To finalize c, the lemma shows that ρ must obtain a configuration sequence such that its last finalized configuration that appears before c 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 18. *Suppose ξ is an execution of* ARES*. For any state σ in ξ, if $\mathbf{c}^{p}_{\sigma}[j].status = F$ for some process $p \in \mathcal{I}$, then there exists a* reconfig *action ρ by a reconfigurer $rc \in \mathcal{G}$, such that (i) rc invokes* finalize-config$(\mathbf{c}^{rc}_{\sigma'})$ *during ρ at some state σ' in ξ, (ii) $\nu(\mathbf{c}^{rc}_{\sigma'}) = j$, and (iii) $\mu(\mathbf{c}^{rc}_{\sigma'}) < j$.*

In ARES, before a read/write/reconfig completes it propagates the maximum tag it discovered by executing the put-data action in the last configuration of its local configuration sequence. When a subsequent operation is invoked, reads the latest configuration sequence and, beginning from the last finalized configuration, it invokes read-data to all the configurations until the end of the sequence. The lemma shows that the latter operation will retrieve a tag which is higher than the tag used in the put-data action. For the following proof we use the notation $\nu^c(\mathbf{c}^{p}_{\sigma}) = \mathbf{c}^{p}_{\sigma}[\nu(\mathbf{c}^{p}_{\sigma})].cfg$. In other words, $\nu^c(\mathbf{c}^{p}_{\sigma})$ is the last configuration in the sequence $\mathbf{c}^{p}_{\sigma}$.

Lemma 19. *Let π_1 and π_2 be two completed read/write/reconfig operations invoked by processes p_1 and p_2 in $\mathcal{I}$, in an execution ξ 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 π_1 and σ_2 is the state in ξ, after the completion of the first* read-config *action of π_2, then there exists a c_2.*put-data$(\langle\tau, v\rangle)$ *action in some configuration $c_2 = \mathbf{c}^{p_2}_{\sigma_2}[i].cfg$, for $\mu(\mathbf{c}^{p_2}_{\sigma_2}) \leq i \leq \nu(\mathbf{c}^{p_2}_{\sigma_2})$, such that (i) it completes in a state σ' before σ_2 in ξ, and (ii) $\tau \geq \tau_{\pi_1}$.*

The following lemma shows the consistency of operations as long as the DAP used satisfy properties **C1** and **C2**.

Lemma 20. *Let π_1 and π_2 denote completed read/write operations in an execution ξ, from processes $p_1, p_2 \in \mathcal{I}$ respectively, such that $\pi_1 \rightarrow \pi_2$. If τ_{π_1} and τ_{π_2} are the local tags at p_1 and p_2 after the completion of π_1 and π_2 respectively, then $\tau_{\pi_1} \leq \tau_{\pi_2}$; if π_1 is a write then $\tau_{\pi_1} < \tau_{\pi_2}$.*

And the safety result of this section follows as:

Theorem 21 (Atomicity)**.** ARES *implements a reconfigurable atomic storage service, given that the* get-data, get-tag, *and* put-data *primitives used satisfy properties **C1** and **C2** of Definition 31.*

In ARES, each configuration may utilize a separate way of implementing the DAP primitives as stated below:

Remark 22. *Algorithm* ARES *satisfies atomicity even when the DAP primitives used 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 used in each c_i satisfy properties **C1** and **C2** of Definition 31.*

4.4 Latency Analysis

No liveness properties are provided for ARES, this is because our reconfiguration mechanism uses consensus, therefore, we provide a conditional performance analysis here. In this section, we examine closely the latencies of each operation in ARES, and study the completion of each operation under various environmental conditions. For our analysis, we assume that local computation take negligible time and delays are only introduced due to the message exchange among the processes. We measure delays in time units of some global clock T, which is visible only to an external viewer. No process has access to T and the clock. Let d and D be the minimum and maximum durations taken by any message to go from one process to another. Also, let $T(\pi)$ denote the communication delay of an operation (or action) π. For simplicity of our analysis, we assume that any propose operation to a consensus instance terminates in $T(CN)$ time units. Given d and D, and through inspection of the algorithm we can provide the delay bounds of the operations and actions used by ARES as follows:

Lemma 23. *If any message send from a process p_1 to a process p_2, s.t. $p_1, p_2 \in \mathcal{I} \cup \mathcal{S}$, takes at least d and at most D time units to be delivered, then the following operations may terminate within the following time intervals: (i) $2d \leq T(\textsf{put-config}) \leq 2D$ and (ii) $2d \leq T(\textsf{read-next-config}) \leq 2D$.*

From Lemma 23 we can derive the delay of a read-config action.

Lemma 24. *For any* read-config *action ϕ such that it accepts an input seq and returns seq', if $\mu = \mu(seq)$ and $\nu = \nu(seq')$ then for ϕ $4d(\nu - \mu + 1) \leq T(\phi) \leq 4D(\nu - \mu + 1)$.*

Lemma 25. *Starting from the last state of ξ, σ, and given that d is the minimum communication delay, then k configurations can be appended to $\mathbf{c}_\sigma$, in time: $T(k) \geq 4d \sum_{i=1}^{k} i + k\,(T(CN) + 2d)$ in our execution construction.*

Now, we can bound the latency of that a read or write operation. The implementation, and the maximum delays, of the DAPs used by ARES impact the delay of read and write operations. For our analysis, taking ABD algorithm as an example, we assume that the implementation of each of get-data, get-tag and put-data has two phases of communication.

Lemma 26. *If any message send from a process p_1 to a process p_2, s.t. $p_1, p_2 \in \mathcal{I} \cup \mathcal{S}$, takes at least d and at most D time units to be delivered, then the DAPs may terminate in: (i) $2d \leq T(\textsf{put-data}) \leq 2D$; (ii) $2d \leq T(\textsf{get-tag}) \leq 2D$; and (iii) $2d \leq T(\textsf{get-data}) \leq 2D$.*

Having the delays for the DAPs we can now compute the delay of a read/write operation π.

Lemma 27. *Let σ_s and σ_e be the states before the invocation and after the completion step of a read/write operation π by p respectively, in some execution ξ of* ARES,*. Then π takes time at most: $T(\pi) \leq 6D\left[\nu(\mathbf{c}^p_{\sigma_e}) - \mu(\mathbf{c}^p_{\sigma_s}) + 2\right]$.*

It remains now to examine if a read/write operation may "catch up" with any ongoing reconfigurations. For worst case analysis, we assume that reconfiguration operations may communicate respecting the minimum delay d, whereas read and write operations suffer the maximum delay D in each message exchange. We consider three cases with respect to the number of configurations installed k, and the minimum delay d of messages: (i) k is finite, and d may be very small; (ii) k is infinite, and d may be very small; (iii) k is infinite, and d can be bounded.

k **is finite, and** d **may be unbounded small.** As the d is unbounded, it follows that reconfigurations may be installed almost instantaneously. Let us first examine what is the maximum delay bound of a any read/write operation. [NN:There is something wrong with this statement.]

k **is infinite, and** d **is bounded.** We show bounds on d with respect to the D and k if we want to allow a read/write operation to reach ongoing reconfigurations.

Lemma 28. *A read/write operation π may terminate in any execution ξ of* ARES *given that k configurations are installed during π, if $d \geq \frac{3D}{k} - \frac{T(CN)}{2(k+2)}$*

5 Efficient state transfer during reconfiguration

In this section, we show that TREAS can be adapted to allow reconfiguration where the object values are transferred directly from the servers in configuration, that is already finalized, to those of a new configuration, without the recon client handling object values. In Algs. 8 and 9, we show the changes necessary to adapt ARES and TREAS for achieve this. Here every configuration uses TREAS as the underlying atomic memory emulation algorithm, and we refer to this algorithm as ARES-TREAS.

In ARES, the procedure update-config (see Alg. 5) is modified as shown in Alg. 8. Consider a reconfiguration client rc, which invokes update-config, during a reconfiguration operation, where it iteratively gathers the tag-config ID pairs in the set variable M by calling get-tag (lines Alg. 8: 5-9). Suppose $\langle \tau, C \rangle$ is the tag and configuration ID pair corresponding to the highest tag in M (lines Alg. 8:11). Next, rc executes procedure forward-code-element(REQ-FW-CODE-ELEM, τ, C, C'), to send a requests to the servers in C to forward their respective coded elements corresponding to τ, to each server in $C'.Servers$. Suppose the MDS code parameters in C and C' are $[n, k]$ and $[n', k']$, respectively, such that, $|C.Servers| = n$, $|C'.Servers| = n'$, and for some $k \geq \frac{2n}{3}$ and $k' \geq \frac{2n'}{3}$. In forward-code-element, the call to md-primitive(REQ-FW-CODE-ELEM, τ, C'), presented in [21], delivers the message (REQ-FW-CODE-ELEM, τ, C') to either every non-faulty servers in $C.Servers$ or none. We rely on the semantics of md-primitive to avoid lingering of coded elements for ever in D due to crash failure of the rc, or servers in $C.Servers$. For example, suppose rc communicates only to one server, say s_i, in $C.Servers$ and crashes, then the rest of the servers in C would not send their coded elements to the servers in C'. As a result, the coded element from s_i will linger around in the D variables in the servers in $C'.Servers$ without ever being removed, thereby, progressively increasing the storage cost. Upon delivering these messages to any server s_i, in $C.Servers$, if (τ, e_i) in $List$ in s_i, then s_i sends (FWD-CODE-ELEM, $\langle \tau, e_i \rangle, rc$) to servers in C'.

Next upon receiving any of the FWD-CODE-ELEM messages, at any server s'_j in $C'.Servers$ if $rc \in Recons$ in s'_j (Alg. 9:9) then it ignores it because rc has already been updated by s'_j regarding the object value of τ. Otherwise, s'_j checks if $\langle \tau, e_j \rangle \in List$ (Alg. 9:10), if is not, then s'_j adds the incoming pair $\langle \tau, e_i \rangle$ to D. Next, s'_j checks if the value for τ is decodable (Alg. 9:12), from the coded elements in D, if it is, then s'_j decodes the value v, using decoder for C', with parameters $[n, k]$, and re-encodes, according to parameters $[n', k']$ to get $e'_j \equiv \Phi_{C'}(v)_j$. Then s'_j proceeds to store $\langle \tau, e'_j \rangle$ in a similar steps as in the put-data response in the TREAS (Alg. 9). Then in lines Alg. 9:20-22 if $\langle \tau, * \rangle \in List$ then s'_j adds rc to the list $Recons$ and s'_j sends rc an ACK. Finally, once rc receives ACKs from $\lceil \frac{n'+k'}{2} \rceil$ servers in $C'.Servers$ it completes the call to update-config.

Finally, it can be shown that ARES-TREAS implements an atomic memory service as stated in the following theorem.

Algorithm 8 Alternate update-config for the reconfiguration protocol of ARES.

procedure update-config(seq)
  $\mu \leftarrow \max(\{j : seq[j].status = F\})$
  $\nu \leftarrow |seq|$
  $M \leftarrow \emptyset$
  for $i = \mu : \nu$ **do**
    ~~$t \leftarrow seq[i].cfg$.get-data()~~
    $t \leftarrow seq[i].cfg$.get-tag()

    ~~$M \leftarrow M \cup \{\langle \tau, v \rangle\}$~~
    $M \leftarrow M \cup \{\langle t, seq[i].cfg \rangle\}$

  ~~$\langle \tau, v \rangle \leftarrow \max_t\{\langle t, v \rangle : \langle t, v \rangle \in M\}$~~
  $\langle \tau, C \rangle \leftarrow \max_t\{\langle t, cfg \rangle : \langle t, cfg \rangle \in N\}$

  /* $C' \equiv seq[\nu]$) */
  ~~C'.put-data($\langle \tau, v \rangle$)~~
  forward-code-element(τ, C, C')
end procedure

procedure forward-code-element(τ, C, C')
  Call md-primitive (REQ-FW-CODE-ELEM, τ, C') on servers in C
  until ACK from $\left\lceil \frac{n'+k'}{2} \right\rceil$ servers in $C'.Servers$
end procedure

Algorithm 9 Additional server protocol and state-variable at a server in TREAS.

at each server s_i in any configuration
Additional State Variables:
$D \subseteq \mathcal{T} \times \mathcal{C}_s$, initially $\{(t_0, \Phi_i(v_0))\}$
$Recons$, set of reconfig client ids, initially empty

/*s_i in configuration C */
Upon recv (REQ-FW-CODE-ELEM, t, C') $_{s_i}$ **from** rc
  if $(t, e_i) \in List$ **then**
    Send (FWD-CODE-ELEM, $\langle \tau, e_i \rangle, rc$) to servers in C'
end receive

/*s'_j in configuration C' */
Upon recv (FWD-CODE-ELEM, $\langle \tau, e_i \rangle, rc$) $_{s'_j}$ **from** s_i
  if $rc \notin Recons$ **then**
    if $(t, *) \notin List$ **then**
      $D \leftarrow D \cup \{\langle t, e_i \rangle\}$
      if isDecodable(D, t) **then**
        $v \leftarrow$ decode(D, t) with Φ_C^{-1}
        $D \leftarrow D - \{\langle t, e_i \rangle\} \cup \{\langle t, \perp \rangle\}$
        $e'_j \leftarrow \Phi_{C'}(v)_j$
        $List \leftarrow List \cup \{\langle \tau, e_j \rangle\}$
        if $|List| > \delta + 1$ **then**
          $\tau_{min} \leftarrow \min\{t : \langle t, * \rangle \in List\}$
          /* remove the coded value, keep the tag */
          $List \leftarrow List \backslash \{\langle \tau, e \rangle : \tau = \tau_{min} \wedge \langle \tau, e \rangle \in List\} \cup \{(\tau_{min}, \perp)\}$
    if $(t, *) \in List$ **then**
      $Recon \leftarrow Recons \cup \{rc\}$
      Send ACK to rc
end receive

Theorem 29 (Atomicity)**.** *Algorithm* ARES-TREAS *implements a reconfigurable atomic storage service, if* get-data, get-tag, *and* put-data *primitives used satisfy **C1** and **C2** of Definition 31.*

6 Conclusions

In this paper, we presented an new algorithmic framework suitable for reconfiguration of the set of servers that implements erasure code-based atomic memory service in message-passing environments. We also provided a new two-round erasure code-based algorithm that has near optimal storage cost, and bandwidth costs per read or write operation. Moreover, this algorithm is suitable specifically where during new configuration installation the object values passes directly from servers in older configuration to those in the newer configurations. Future work will involve adding efficient repair and reconfiguration using regenerating codes.

References

[1] 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.

[2] 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.

[3] 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.

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

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

[6] 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.

[7] 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 17) (2017), pp. 539–551.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] Herlihy, M., and Shavit, N. On the nature of progress. In Proceedings of the 15th International Conference on Principles of Distributed Systems (Berlin, Heidelberg, 2011), OPODIS'11, Springer-Verlag, pp. 313–328.

[17] Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS) 12, 3 (1990), 463–492.

[18] Huffman, W. C., and Pless, V. Fundamentals of error-correcting codes. Cambridge university press, 2003.

[19] 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.

[20] Konwar, K. M., Prakash, N., Kantor, E., Lynch, N., Medard, M., and Schwarzmann, A. A. Storage-optimized data-atomic algorithms for handling erasures and errors in distributed storage systems. In 30th IEEE International Parallel & Distributed Processing Symposium (IPDPS) (2016).

[21] 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.

[22] 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).

[23] Lu, H., Hodsdon, C., Ngo, K., Mu, S., and Lloyd, W. The SNOW theorem and latency-optimal read-only transactions. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) (Savannah, GA, 2016), pp. 135–150.

[24] Lynch, N. Distributed Algorithms. Morgan Kaufmann Publishers, 1996.

[25] 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.

[26] 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.

[27] 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 10) (2010), p. 2226.

[28] 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.

[29] 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.

Appendix

A The data access primitives

In this section we focus on algorithms that utilize logical tags to implement atomic read/write objects, and analayze their correctness and liveness in terms of three *data access primitives* (DAP). These data access primitives are specific to any configuration c in context : (i) put-data($\langle\tau, v\rangle$), (ii) c.get-data(), and (iii) c.get-tag(). Assuming a set of totally ordered timestamps $\mathcal{T}$, a value domain of the distributed atomic object $\mathcal{V}$, and a set of configuration identifiers $\mathcal{C}$, the three primitives can be defined over a configuration $c \in \mathcal{C}$, tag $\tau \in \mathcal{T}$, and a value $v \in \mathcal{V}$ as follows:

Definition 30 (Data Access Primitives)**.** *Given a configuration identifier $c \in \mathcal{C}$, any non-faulty client process p may invoke the following data access primitives during an execution ξ:*

D1. c.get-tag()*: returns a tag $\tau \in \mathcal{T}$*

D2. c.get-data()*: returns a tag-value pair $(\tau, v) \in \mathcal{T} \times \mathcal{V}$*

D3. c.put-data($\langle\tau, v\rangle$)*: the tag-value pair $(\tau, v) \in \mathcal{T} \times \mathcal{V}$ as argument*

where c is added to specify the configuration specific implementation of these privimites.

Most logical timestamp-based shared atomic memory implementations, assume that a tag $\tau \in \mathcal{T}$ is defined as a pair (z, w), where $z \in \mathbb{N}$ and $w \in \mathcal{W}$, an ID of a writer. 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 $\tau_2.w > \tau_1.w$. Now consider an algorithmic template (see

Algorithm 10 Read and write operations of generic algorithm A_1

operation read()
  $\langle t, v\rangle \leftarrow c.\text{get-data}()$
  $c.\text{put-data}(\langle t, v\rangle)$
  return $\langle t, v\rangle$
end operation

operation write(v)
  $t \leftarrow c.\text{get-tag}()$
  $t_w \leftarrow \langle t.z + 1, w\rangle$
  $c.\text{put-data}(\langle t_w, v\rangle)$
end operation

Automaton 10), we call A_1. In brief, a read operation in A_1 performs $c.c.\text{get-data}()$ to retrieve a tag-value pair, $\langle \tau, v\rangle$ form configuration c, and then it performs a $c.\text{put-data}(\langle \tau, v\rangle)$ to propagate that pair to 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. We can show that A_1 satisfy atomic guarantees and liveness if the DAP in the above algorithms satisfy the following consistency properties:

Definition 31 (DAP Consistency Properties)**.** *In an execution ξ we say that a DAP operation in an execution ξ is complete if both the invocation and the matching response step appear in ξ. If Π is the set of complete DAP operations in execution ξ then for any $\phi, \pi \in \Pi$:*

C1 If ϕ is a $c.\text{put-data}(\langle \tau_\phi, v_\phi\rangle)$, for $c \in \mathcal{C}$, $\tau_1 \in \mathcal{T}$ and $v_1 \in \mathcal{V}$, and π is a $c.\text{get-tag}()$ (or a $c.\text{get-data}()$) that returns $\tau_\pi \in \mathcal{T}$ (or $\langle \tau_\pi, v_\pi\rangle \in \mathcal{T} \times \mathcal{V}$) and ϕ completes before π in ξ, then $\tau_\pi \geq \tau_\phi$.

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

Algorithm 11 Read and write operations of generic algorithm A_2

operation read()
  $\langle t, v\rangle \leftarrow c.\text{get-data}()$
  return $\langle t, v\rangle$
end operation

operation write(v)
  $t \leftarrow c.\text{get-tag}()$
  $t_w \leftarrow \langle t.z + 1, w\rangle$
  $c.\text{put-data}(\langle t_w, v\rangle)$
end operation

A slightly different algorithmic template A_2 (see Automaton 11), captures algorithms where read operations avoid the propagation phase. In A_2 the write protocol is the same as in A_1 however the read operation terminates as soon as the get-data action returns. In addition to **C1** and **C2**, algorithms that are transformed in the generic algorithm $A2$ need to satisfy the following property:

$C3$ If ϕ is a $c.\text{get-data}()$ that returns $\langle \tau_\phi, v_\phi\rangle$ and π is a $c.\text{get-data}()$ that returns $\langle \tau_\pi, v_\pi\rangle$, and $\phi \rightarrow \pi$, then $\tau_\phi \leq \tau_\pi$.

Now we can show that if those properties are satisfied from the DAP, then algorithms A_1 and A_2 implement an atomic read/write algorithm.

Theorem 32 (Atomicity of A_1)**.** *Suppose the DAP implementation satisfies the consistency properties $C1$ and $C2$ of Definition 31. Then any execution ξ the atomicity protocols A_1 on a configuration $c \in \mathcal{C}$, is atomic and live if DAPs are live in ξ.*

Proof. We prove atomicity by proving properties $A1$, $A2$ and $A3$ appearing in the definition of atomicity in Section 2 for any execution of the algorithm.

Property $A1$: Consider two operations ϕ and π such that ϕ completes before π 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 ϕ and π are writes. The c.put-data$(*)$ of ϕ completes before π is invoked. By property $C1$ the tag τ_π returned by the c.get-data$()$ at π is at least as large as τ_ϕ. Now, since τ_π is incremented by the write operation then π puts a tag τ'_π such that $\tau_\phi < \tau'_\pi$ and hence we cannot have $\pi \prec \phi$.

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

Case (c): ϕ is a read and π is a write. Let the id of the writer that invokes π we w_π. The c.put-data$(\langle \tau_\phi, *\rangle)$ call of ϕ completes before c.get-tag$()$ of π is initiated. Therefore, by property $C1$ get-tag(c) returns τ such that, $\tau_\phi \leq \tau$. Since τ_π is equal to $(\tau.z + 1, w_\pi)$ by design of the algorithm, hence $\tau_\pi > \tau_\phi$ and we cannot have $\pi \prec \phi$.

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

Property $A2$: Note that because $\mathcal{T}$ is well-ordered we can show that this property by first showing that every write has a unique tag. This means any two pair of writes can be ordered. Now, a read can be ordered . Note that a read can be ordered w.r.t. to 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.

Now observe that two tags generated from two write operations from different writers are necessarily distinct because of the id component of the tag. Now if the operations, say ϕ and π are writes from the same writer then by well-formedness property the second operation is invoked after the first completes, say without loss of generality ϕ completes before π is invoked. In that case the integer part of the tag of π is higher by property $C1$, and since the c.get-tag$()$ is followed by c.put-data$(*)$. Hence π is ordered after ϕ.

Property $A3$: This is clear because the tag of a reader is defined by the tag of the value it returns by property (b). Therefore, the reader's immediate previous value it returns. On the other hand if does note return any write operation's value it must return v_0. □

Theorem 33 (Atomicity of A_2)**.** *Suppose the DAP implementation satisfies the consistency properties $C1$, $C2$ and $C3$ of Definition 31. Then any execution ξ the atomicity protocols A_2 on a configuration $c \in \mathcal{C}$, is atomic and live if DAPs are live in ξ.*

Proof. The proof of the claim is similar to the case of algorith A_1 except for the argument in Property $P1$ Case (d). In case of A_2, the c.get-data$()$ is executed as a part of ϕ and completes before c.get-data$()$ is called in π. By property $C3$ of the data-access primitives, we have $\tau_\phi \leq \tau_\pi$ and hence we cannot have $\pi \prec \phi$. □

Expressing an atomic algorithm in terms of the DAP primitives serves multiple purposes. First, describing an algorithm according to templates A_1 or A_2 allows one to proof that the algorithm is *safe* (atomic) by just showing that the appropriate DAP properties hold, and the algorithm is *live* if the implementation of each primitive is live. Secondly, the safety and liveness proofs for more complex algorithms (like ARES in Section 4) become easier as one may reason on the DAP properties that are satisfied by the primitives used, without involving the underlying implementation of those primitives. And last but not least, describing a reconfigurable algorithm using DAPs, provides the flexibility to use different implementation mechanics for the DAPs in each reconfiguration, as long as the DAPs used in a single configuration satisfy the appropriate DAP properties. In Section 4, we discuss how ARES may change the primitives mechanisms in each established configuration without affecting the safety guarantees of the service. Such approaches can adapt to the configuration design, and vary the performance of the service based on the environmental conditions. In other words, ABD [4] can be used for maximum fault tolerance and when majority quorums are used, whereas fast algorithms similar to the ones presented in [9, 3], could be used in configurations that satisfy the appropriate participation bounds.

A.1 Representing Known Algorithms in terms of data-access primitives

Any tag-based algorithm can be transformed into a generic algorithm A_1 or A_2. A straight-forward transformation of any multi-reader multi-writer atomic memory algorithm A is to convert it to A_1 by appropriately defining $c.\text{get-data}()$ action in terms of the protocol for the read operation in A; and the $c.\text{put-data}(\langle \tau, v \rangle)$ in terms of the write protocol. Since A is an atomic algorithm such implementation of the primitives would satisfy the DAP consistency properties and therefore A_1 would also satisfy atomicity. However, such transformation of A is not nessarily efficient in terms of the number of rounds or communication complexity associated with an operation, or even storage cost, as each read and write operation performs both the read and write protocols of the original algorithm A.

In this subsection we demonstrate how two well known algorithms for emulating atomic read/write memory can be transformed to their generic equivalent algorithms. In particular, we will present the very celebrated ABD algorithm [4] and the LDR algorithm presented in [10]. For both algorithms we will specify a transformation to a generic algorithm and present the implementations of their data-primitives as well as the primitive handlers.

MWABD **Algorithm.** The multi-writer version of the ABD can be transformed to the generic algorithm A_1. Automaton 12 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**. We begin with a lemma that shows the monotonicity of the tags at each server.

Lemma 34. *Let σ and σ' two states in an execution ξ such that σ appears before σ' in ξ. 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 σ' that appears after σ in ξ, $s.tag|_{\sigma'} \geq \tau$. □

Algorithm 12 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 $\forall s \in Q$
  $\tau_{max} \leftarrow \max(\{\tau_s : p \text{ received } \langle \tau_s, v_s \rangle \text{ from } s\})$
  return τ_{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 $\forall s \in Q$
  $\tau_{max} \leftarrow \max(\{\tau_s : r_i \text{ received } \langle \tau_s, v_s \rangle \text{ from } s\})$
  return $\{\langle \tau_s, v_s \rangle : \tau_s = \tau_{max} \wedge \ p \text{ received } \langle \tau_s, v_s \rangle \text{ from } s\}$
end procedure

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

Upon receive (QUERY-TAG) **from** q
  send τ 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 35. *Let ϕ be a c.put-data($\langle \tau, v \rangle$) action invoked by p_1 and γ be a c.get-tag() action invoked by p_2 in a configuration c, such that $\phi \rightarrow \gamma$ in an execution ξ of the algorithm. Then γ 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 ϕ and γ actions. By Lemma 34 and since s received a tag τ, then s replies to p_2 with a tag $\tau_s \geq \tau$. Since γ 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 36. *Let π be a c.put-data($\langle \tau, v \rangle$) action invoked by p_1 and ϕ be a c.get-data() action invoked by p_2 in a configuration c, such that $\pi \rightarrow \phi$ in an execution ξ of the algorithm. Then ϕ 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 37. *If ϕ is a c.*get-data() *that returns $\langle \tau_\pi, v_\pi \rangle \in \mathcal{T} \times \mathcal{V}$, then there exists π such that π is a c.*put-data($\langle \tau_\pi, v_\pi \rangle$) *and $\phi \not\to \pi$.*

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 ϕ 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, π. Thus, π can proceed or be concurrent with ϕ, and hence $\phi \not\to \pi$. □

LDR **Algorithm.** The LDR algorithm [10] was designed with large objects in mind, and for that reason it decouples the meta-data associated with each atomic object from the actual object value. So the algorithm considers the existence of two sets of servers: (a) the directory servers that maintain the information about the latest tags in the system, and (b) the replica servers that maintain the replica values (also associated with particular tags). So a configuration c must include two sets of servers: $Directories(c) \subset c.Servers$ and $Replicas(c) \subset c.Servers$. Automaton 13 shows the specification along with the handlers of each DAP. With this specification LDR can be transformed to the template of algorithm A_2 where essentially the c.get-data() primitive encapsulates the specification of the read operation as defined in LDR.

B TREAS Correctness and Liveness

Lemma. 5 *The data-access primitives, i.e.,* get-tag, get-data *and* put-data *primitives, implemented in the* TREAS *algorithm satisfy the consistency properties.*

Proof. As mentioned above we are concerned with only configuration c, and therefore, in our proofs we will be concerned with only one configuration. Let α be some execution of TREAS, then we consider two cases for π for proving property $C1$: π is a get-tag operation, or π is a get-data primitive.

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

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

Algorithm 13 Implementation of DAP for LDR at process p using configuration c

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

```
procedure c.get-tag()
    send (QUERY-TAG-LOCATION) to each s ∈ Dir(c)
    until ∃Q, Q ∈ Majority(Dir(c)) s.t.
            p receives ⟨t_s, loc_s⟩ from ∀s ∈ Q
    τ_max ← max({τ_s : p, p ∈ Q received ⟨τ_s, loc_s⟩ from s})
    return τ_max
end procedure

procedure c.put-data(⟨τ, v⟩))
    send (PUT-DATA, ⟨τ, v⟩) to 2f + 1 servers in Rep(c)
    until p receives ACK from a set U of f + 1 servers in Rep(c)

    send (PUT-METADATA, ⟨τ, U⟩) to all servers in Dir(c)
    until p receives ACK from a majority servers in Dir(c)
end procedure

procedure c.get-data()
    send (QUERY-TAG-LOCATION) to each s ∈ Dir(c)
    until ∃Q, Q ∈ Majority(Dir(c)) s.t.
            p receives ⟨τ_s, loc_s⟩ from ∀s ∈ Q
    (τ_max, U_max) ← max_{τ_s}({(τ_s, loc_s) : p received ⟨τ_s, loc_s⟩ from s})

    send (PUT-METADATA, ⟨τ_max, U_max⟩) to all servers in Dir(c)
    until p receives ACK from a majority servers in Dir(c)

    send (GET-DATA, τ_max) to any f + 1 servers in U_max
    until p receives (τ_max, v) from any server in U_max
    return ⟨τ_max, v⟩
end procedure
```

Primitive Handlers at each Directory server $s \in Dir(c)$**:**

```
Upon receive (QUERY-TAG-LOCATION) from q
    send ⟨τ, loc⟩ to q
end receive

Upon receive (PUT-METADATA, ⟨τ_in, loc_in⟩) from q
    if τ_in > τ then
        ⟨τ, loc⟩ ← ⟨τ_in, loc_in⟩
    send ACK to q
end receive
```

Primitive Handlers at each Replica server $s \in Rep(c)$**:**

```
Upon receive (GET-DATA) from q
    send ⟨τ, v⟩ to q
end receive

Upon receive (PUT-DATA, ⟨τ_in, v_in⟩) from q
    if τ_in > τ then
        ⟨τ, v⟩ ← ⟨τ_in, v_in⟩
    send ACK to q
end receive
```

Next, we prove the $C2$ property of DAP for the TREAS 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_π 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_π, v_π) operation ϕ. To show that for every π it cannot be that ϕ completes before π, we adopt by a contradiction. Suppose for every π, ϕ completes before π begins, then clearly t_π cannot be returned ϕ, a contradiction. □

Theorem. 9 *[Liveness] Let β denote a well-formed execution of* TREAS*, with $[n, k]$, where n is the number of servers and $k > n/3$, and δ be the maximum number of write operations concurrent with any valid read operation then the read and write operations β are live.*

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 π, specifically, the get-data operation completes.

Let α be and execution of TREAS and let c_{σ^*} and c_π be the clients that invokes the write operation σ^* and read operation c_π, respectively.

Let S_{σ^*} be the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to c_{σ^*}, in the put-data operations, in σ^*. Let S_{σ^π} be the set of $\lceil \frac{n+k}{2} \rceil$ servers that responds to c_π during the get-data step of π. Note that in α at the point execution T_1, just before the execution of π, none of the the write operations in Λ is complete. Observe that, by algorithm design, the coded-elements corresponding to t_{σ^*} are garbage-collected from the $List$ variable of a server only if more than δ 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_{σ^*} is not garbage collected in α, at least until execution point T_2 in any of the servers in S_{σ^*}.

Therefore, during the execution fragment between the execution points T_1 and T_2 of the execution α, the tag and coded-element pair is present in the $List$ variable of every in S_{σ^*} that is active. As a result, the tag and coded-element pairs, $(t_{\sigma^*}, \Phi_s(v_{\sigma^*}))$ exists in the $List$ received from any $s \in S_{\sigma^*} \cap S_\pi$ during operation π. Note that since $|S_{\sigma^*}| = |S_\pi| = \lceil \frac{n+k}{2} \rceil$ hence $|S_{\sigma^*} \cap S_\pi| \geq k$ and hence $t_{\sigma^*} \in Tags_{dec}^{\geq k}$, the set of decodable tag, i.e., the value v_{σ^*} can be decoded by c_π in π, which demonstrates that $Tags_{dec}^{\geq k} \neq \emptyset$. Next we want to argue that $t^*_{max} = t^{dec}_{max}$ via a contradiction: we assume $\max Tags_*^{\geq k} > \max Tags_{dec}^{\geq k}$. Now, consider any tag t, which exists due to our assumption, such that, $t \in Tags_*^{\geq k}$, $t \notin Tags_{dec}^{\geq k}$ 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_π. Note that since $k > n/3$ hence $|S_{\sigma^*} \cap S_\pi| \geq \lceil \frac{n+k}{2} \rceil + \lceil \frac{n+1}{3} \rceil \geq 1$, i.e., $S_{\sigma^*} \cap S_\pi \neq \emptyset$. Then t must be in some servers in S_{σ^*} at T_2 and since $t > t^{dec}_{max} \geq t_{\sigma^*}$. 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 Λ) to garbage-collect the coded-elements corresponding to tag t, which also holds for tag t^*_{max}. In that case, t must be in $Tag_{dec}^{\geq k}$, a contradiction. □

B.1 Storage and Communication Costs

Lemma 38. *The worst-case total storage cost of* TREAS *algorithm 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 TREAS. Once again, note that we ignore the communication cost arising from exchange of meta-data.

Lemma 39. *The communication cost associated with a successful write operation in* TREAS *is at most* $\frac{n}{k}$.

Proof. During read operation, in the get-tag phase the servers responds 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 40. *The communication cost associated with a successful read operation in* TREAS *is at most* $(\delta + 2)\frac{n}{k}$.

Proof. During read operation, in the get-data phase the servers responds 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. 3 *The* TREAS *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 to $(\delta+2)\frac{n}{k}$.*

C ARES Safety

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 σ of an execution ξ. If server s crashes at a state σ_f in an execution ξ then $s.var|_\sigma \triangleq s.var|_{\sigma_f}$ for any state variable var and for any state σ that appears after σ_f in ξ.

Consensus instance in a configuration We assume that the servers in each configuration c implements a consensus service $c.Con$, where any client is allowed to proposed values from the set $\mathcal{C}$, and $c.Con$satisfies the following properties

Definition 41. *For a configuration $c \in \mathcal{C}$, $c.Con$ must satisfy the following properties:*

Agreement: *No two processes that participate in $c.Con$ decide a different value.*

Validity: *If any process decides a value c' then c' was proposed by some process.*

Termination: *Every correct process decides.*

Definition 42 (Tag of a configuration)**.** *Let $c \in \mathcal{C}$ be a configuration, σ be a state in some execution ξ then we define the tag of c at state σ as $tag(c)|_\sigma \triangleq \min_{Q\in c.Quorums}\max_{s\in Q}\ (s.tag|_\sigma)$. We often drop the suffix $|_\sigma$, and simply denote as $tag(c)$, when the state in the execution is clear from the context.*

Definition 43. *Let $\mathbf{c}^p_\sigma = p.cseq|_\sigma$. Then we define as $\mu(\mathbf{c}^p_\sigma) \triangleq \max\{i : \mathbf{c}^p_\sigma[i].status = F\}$ and $\nu(\mathbf{c}^p_\sigma) \triangleq |\mathbf{c}^p_\sigma|$, where $|\mathbf{c}^p_\sigma|$ is the number of elements in configuration vector $\mathbf{c}^p_\sigma$ that are not equal to $\perp$.*

Definition 44 (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 $\mathbf{x}[j] \neq \perp$.*

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

Lemma 45. *For any reconfigurer r that invokes an* reconfig(c) *action in an execution ξ 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. $\square$

Lemma 46. *If a server s sets $s.nextC$ to $\langle c, F\rangle$ at some state σ in an execution ξ of the algorithm, then $s.nextC = \langle c, F\rangle$ for any state σ' that appears after σ in ξ.*

Proof. Notice that a server s updates the $s.nextC$ variable for some specific configuration c_k in a state st if: (i) s did not receive any value for c_k before (and thus $nextC = \bot$), or (ii) s received a tuple $\langle c, P\rangle$ and before σ received the tuple $\langle c', F\rangle$. By Observation 41 $c = c'$ as s updates the $s.nextC$ of the same configuration c_k. Once the tuple becomes equal to $\langle c, F\rangle$ then s does not satisfy the update condition for c_k, and hence in any state σ' after σ it does not change $\langle c, F\rangle$. $\square$

Lemma 47 (Configuration Uniqueness)**.** *For any processes $p, q \in \mathcal{I}$ and any states σ_1, σ_2 in an execution ξ, it must hold that $\mathbf{c}^p_{\sigma_1}[i].cfg = \mathbf{c}^q_{\sigma_2}[i].cfg$, $\forall i$ s.t. $\mathbf{c}^p_{\sigma_1}[i].cfg, \mathbf{c}^q_{\sigma_2}[i].cfg \neq \bot$.*

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 σ_1 appears before σ_2 in ξ.

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 45, since both processes want to install a configuration in $i = 1$, then they have to run $Cons[0]$ instance over the configuration stored in their local $cseq[0].cfg$ variable. Since $p.cseq[0].cfg = q.cseq[0].cfg = c_0$ then both $Cons[0]$ instances run over the same configuration c_0 and according to Observation 41 return the same value, say c_1. Hence $c_p = c_q = c_1$ and $p.cseq[1].cfg = q.cseq[1].cfg = c_1$.

Let us examine the case now where p or q assign a configuration c they received from some server $s \in c_0.Servers$. According to the algorithm only the configuration that has been decided by the consensus instance on c_0 is propagated to the servers in $c_0.Servers$. If c_1 is the decided configuration, then $\forall s \in c_0.Servers$ such that $s.nextC(c_0) \neq \bot$, it holds that $s.nextC(C_0) = \langle c_1, *\rangle$. So if p or q set $p.cseq[1].cfg$ or $q.cseq[1].cfg$ to some received configuration, then $p.cseq[1].cfg = q.cseq[1].cfg = c_1$ in this case as well.

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

Induction Step: We need to show that the lemma holds for $i = k+1$. If both processes retrieve $p.cseq[k+1].cfg$ and $q.cseq[k+1].cfg$ through consensus, then both p and q run consensus over the previous configuration. Since according to our hypothesis $\mathbf{c}^p_{\sigma_1}[k] = \mathbf{c}^q_{\sigma_2}[k]$ then both process will receive the same decided value, say c_{k+1}, and hence $p.cseq[k+1].cfg = q.cseq[k+1].cfg = c_{k+1}$. Similar to the base case, a server in $c_k.Servers$ only receives the configuration c_{k+1} decided by the consensus instance run over c_k. So processes p and q can only receive c_{k+1} from some server in $c_k.Servers$ so they can only assign $p.cseq[k+1].cfg = q.cseq[k+1].cfg = c_{k+1}$ at Line A5:7. That completes the proof. $\square$

Lemma 47 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 48. *In any execution ξ of the algorithm , If for any process $p \in \mathcal{I}$, $\mathbf{c}^p_\sigma[i] \neq \perp$ in some state σ in ξ, then $\mathbf{c}^p_{\sigma'}[i] \neq \perp$ in any state σ' that appears after σ in ξ.*

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 $\perp$ (A5:7), 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 A5:L7. Hence, if $\mathbf{c}^p_\sigma[i] \neq \perp$ at state σ then it cannot become $\perp$ in any state σ' after σ in execution ξ. □

Lemma 49. *Let σ_1 be some state in an execution ξ 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 $\mathbf{c}^p_{\sigma_1}[j+1] \neq \perp$. Suppose w.l.o.g. that $j = k-1$ and that σ_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 45 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 $\mathbf{c}^p_{\sigma_1}[k] = c_k$. 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 σ' before σ_1 in ξ, 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 σ_1. So if state σ' is before σ_1 in ξ, then by Lemma 48, 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 we can show that $\mathbf{c}^p_{\sigma_1}[j] \neq \perp$, for $0 \leq j \leq k$. □

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

Lemma 50. *If at a state σ of an execution ξ of the algorithm, if $\mu(\mathbf{c}^p_\sigma) = k$ 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 A4:9 and A5:17), it then propagates this value to a quorum of the previous configuration (Lines A4:8 and A5:18). So if a process p assigned c_j to an element $\mathbf{c}^{p}_{\sigma'}[j]$ in some state σ' in ξ, then p may assign a value to the $j+1$ element of $\mathbf{c}^{p}_{\sigma''}[j+1]$ only after put-config($\mathbf{c}^{p}_{\sigma'}[j-1].cfg, \mathbf{c}^{p}_{\sigma'}[j]$) occurs. During put-config action, p propagates $\mathbf{c}^{p}_{\sigma'}[j]$ in a quorum $Q \in \mathbf{c}^{p}_{\sigma'}[j-1].cfg.Quorums$. Hence, if $\mathbf{c}^{p}_{\sigma}[k] \neq \perp$, then p propagated each $\mathbf{c}^{p}_{\sigma'}[j]$, for $0 < j \leq k$ to a quorum of servers $Q \in \mathbf{c}^{p}_{\sigma'}[j-1].cfg.Quorums$. And this completes the proof. $\square$

Lemma 51 (Configuration Prefix)**.** *Let π_1 and π_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 ξ. Let σ_1 be the state after the response step of π_1 and σ_2 the state after the response step of π_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 47 for any i such that $\mathbf{c}^{p_1}_{\sigma_1}[i] \neq \perp$ and $\mathbf{c}^{p_2}_{\sigma_2}[i] \neq \perp$, then $\mathbf{c}^{p_1}_{\sigma_1}[i].cfg = \mathbf{c}^{p_2}_{\sigma_2}[i].cfg$. Also from Lemma 49 we know that for $0 \leq j \leq \nu_1, \mathbf{c}^{p_1}_{\sigma_1}[j] \neq \perp$, and $0 \leq j \leq \nu_2, \mathbf{c}^{p_2}_{\sigma_2}[j] \neq \perp$. 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 π_2 (Line A5:2) at some state σ' before σ_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 $\pi_1 \rightarrow \pi_2$ then σ_1 appears before σ' in execution ξ. By Lemma 50, we know that by σ_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 47, 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 π_2. Since $\mathbf{c}^{p_1}_{\sigma_1}[\mu+1] \neq \perp$ then p_2 assigns $\mathbf{c}^{p_2}_{*}[\mu+1] = \mathbf{c}^{p_1}_{\sigma_1}[\mu+1]$, and repeats the process in the configuration $\mathbf{c}^{p_2}_{*}[\mu+1].cfg$. Since every configuration $\mathbf{c}^{p_1}_{\sigma_1}[j].cfg$, for $\mu \leq j < \nu_1$, 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 σ'' before σ_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. $\square$

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 52. *Let σ and σ' two states in an execution ξ such that σ appears before σ' in ξ. 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 σ, 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 σ' and hence $\mu(\mathbf{c}^{p}_{\sigma'}) \geq \mu(\mathbf{c}^{p}_{\sigma})$. $\square$

Lemma 53 (Configuration Progress)**.** *Let π_1 and π_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 ξ. Let σ_1 be the state after the response step of π_1 and σ_2 the state after the response step of π_2. Then $\mu(\mathbf{c}^{p_1}_{\sigma_1}) \leq \mu(\mathbf{c}^{p_2}_{\sigma_2})$.*

Proof. By Lemma 51 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 A4:2 during π_2 such that

$\mathbf{c}_{\sigma'}^{p_2}[\mu_2].status = F$ in some state σ' before σ_2. If $\mu_2 \geq \mu_1$, and since σ_2 is after σ', then by Lemma 52 $\mu_2 \leq \mu(\mathbf{c}_{\sigma_2}^{p_2})$ and hence $\mu_1 \leq \mu(\mathbf{c}_{\sigma_2}^{p_2})$ as well.

It remains to examine the case where $\mu_2 < \mu_1$. Process p_1 sets the status of $\mathbf{c}_{\sigma_1}^{p_1}[\mu_1]$ to F in two cases: (i) either when finalizing a reconfiguration, or (ii) when receiving an $s.nextC = \langle \mathbf{c}_{\sigma_1}^{p_1}[\mu_1].cfg, F\rangle$ from some server s during a read-config action. In both cases p_1 propagates the $\langle \mathbf{c}_{\sigma_1}^{p_1}[\mu_1].cfg, F\rangle$ to a quorum of servers in $\mathbf{c}_{\sigma_1}^{p_1}[\mu_1 - 1].cfg$ before completing. We know by Lemma 51 that since $\pi_1 \rightarrow \pi_2$ then $\mathbf{c}_{\sigma_1}^{p_1}$ is a prefix in terms of configurations of the $\mathbf{c}_{\sigma_2}^{p_2}$. So it must be the case that $\mu_2 < \mu_1 \leq \nu(\mathbf{c}_{\sigma_2}^{p_2})$. Thus, during π_2, p_2 starts from the configuration at index μ_2 and in some iteration performs get-next-config in configuration $\mathbf{c}_{\sigma_2}^{p_2}[\mu_1 - 1]$. According to Lemma 47, $\mathbf{c}_{\sigma_1}^{p_1}[\mu_1 - 1].cfg = \mathbf{c}_{\sigma_2}^{p_2}[\mu_1 - 1].cfg$. Since π_1 completed before π_2, then it must be the case that σ_1 appears before σ' in ξ. However, p_2 invokes the get-next-config operation in a state σ'' which is either equal to σ' or appears after σ' in ξ. Thus, σ'' must appear after σ_1 in ξ. From that it follows that when the get-next-config is executed by p_2 there is already a quorum of servers in $\mathbf{c}_{\sigma_2}^{p_2}[\mu_1 - 1].cfg$, say Q_1, that received $\langle \mathbf{c}_{\sigma_1}^{p_1}[\mu_1].cfg, F\rangle$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 46), then there is a server $s \in Q_1 \cap Q_2$, such that s replies to p_2 with $s.nextC = \langle \mathbf{c}_{\sigma_1}^{p_1}[\mu_1].cfg, F\rangle$. So, $\mathbf{c}_{\sigma_2}^{p_2}[\mu_1]$ gets $\langle \mathbf{c}_{\sigma_1}^{p_1}[\mu_1].cfg, F\rangle$, and hence $\mu(\mathbf{c}_{\sigma_2}^{p_2}) \geq \mu_1$ in this case as well. This completes our proof. □

Theorem 54. *Let π_1 and π_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 ξ. Let σ_1 be the state after the response step of π_1 and σ_2 the state after the response step of π_2. Then the following properties hold:*

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

(b) $\mathbf{c}_{\sigma_1}^{p_1} \preceq_p \mathbf{c}_{\sigma_2}^{p_2}$, *and*

(c) $\mu(\mathbf{c}_{\sigma_1}^{p_1}) \leq \mu(\mathbf{c}_{\sigma_2}^{p_2})$

Proof. Statements (a), (b) and (c) follow from Lemmas 47, 51, and 52. □

Lemma. 45 *For any reconfigurer r that invokes an* reconfig(c) *action in an execution ξ 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. 46 *If a server s sets $s.nextC$ to $\langle c, F\rangle$ at some state σ in an execution ξ of the algorithm, then $s.nextC = \langle c, F\rangle$ for any state σ' that appears after σ in ξ.*

Proof. Notice that a server s updates the $s.nextC$ variable for some specific configuration c_k in a state st if: (i) s did not receive any value for c_k before (and thus $nextC = \perp$), or (ii) s received a tuple $\langle c, P\rangle$ and before σ received the tuple $\langle c', F\rangle$. By Observation 41 $c = c'$ as s updates the $s.nextC$ of the same configuration c_k. Once the tuple becomes equal to $\langle c, F\rangle$ then s does not satisfy the update condition for c_k, and hence in any state σ' after σ it does not change $\langle c, F\rangle$. □

Lemma. 47 *[Configuration Uniqueness] For any processes $p, q \in \mathcal{I}$ and any states σ_1, σ_2 in an execution ξ, it must hold that $\mathbf{c}_{\sigma_1}^{p}[i].cfg = \mathbf{c}_{\sigma_2}^{q}[i].cfg$, $\forall i$ s.t. $\mathbf{c}_{\sigma_1}^{p}[i].cfg, \mathbf{c}_{\sigma_2}^{q}[i].cfg \neq \perp$.*

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 σ_1 appears before σ_2 in ξ.

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 45, since both processes want to install a configuration in $i = 1$, then they have to run $Cons[0]$ instance over the configuration stored in their local $cseq[0].cfg$ variable. Since $p.cseq[0].cfg = q.cseq[0].cfg = c_0$ then both $Cons[0]$ instances run over the same configuration c_0 and according to Definition 41 return the same value, say c_1. Hence $c_p = c_q = c_1$ and $p.cseq[1].cfg = q.cseq[1].cfg = c_1$.

Let us examine the case now where p or q assign a configuration c they received from some server $s \in c_0.Servers$. According to the algorithm only the configuration that has been decided by the consensus instance on c_0 is propagated to the servers in $c_0.Servers$. If c_1 is the decided configuration, then $\forall s \in c_0.Servers$ such that $s.nextC(c_0) \neq \perp$, 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 Alg. 5:9. That completes the proof. □

Lemma. 48 *In any execution ξ of the algorithm , If for any process $p \in \mathcal{I}$, $\mathbf{c}^p_{\sigma}[i] \neq \perp$ in some state σ in ξ, then $\mathbf{c}^p_{\sigma'}[i] \neq \perp$ in any state σ' that appears after σ in ξ.*

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 $\perp$ (A5:7), 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 A5:L9. Hence, if $\mathbf{c}^p_{\sigma}[i] \neq \perp$ at state σ then it cannot become $\perp$ in any state σ' after σ in execution ξ. □

Lemma. 49 *Let σ_1 be some state in an execution ξ 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 $\mathbf{c}^p_{\sigma_1}[j+1] \neq \perp$. Suppose w.l.o.g. that $j = k-1$ and that σ_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 45 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 A4:9. 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 σ' before σ_1 in ξ, 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 σ_1. So if state σ' is before σ_1 in ξ, then by Lemma 48, 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 we can show that $\mathbf{c}^p_{\sigma_1}[j] \neq \perp$, for $0 \leq j \leq k$. $\square$

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. 50 *If at a state σ of an execution ξ of the algorithm, if $\mu(\mathbf{c}^p_\sigma) = k$ 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 A4:7 and A5:17), it then propagates this value to a quorum of the previous configuration (Lines A4:8 and A5:18). So if a process p assigned c_j to an element $\mathbf{c}^p_{\sigma'}[j]$ in some state σ' in ξ, then p may assign a value to the $j+1$ element of $\mathbf{c}^p_{\sigma''}[j+1]$ only after put-config($\mathbf{c}^p_{\sigma'}[j-1].cfg, \mathbf{c}^p_{\sigma'}[j]$) occurs. During put-config action, p propagates $\mathbf{c}^p_{\sigma'}[j]$ in a quorum $Q \in \mathbf{c}^p_{\sigma'}[j-1].cfg.Quorums$. Hence, if $\mathbf{c}^p_\sigma[k] \neq \perp$, then p propagated each $\mathbf{c}^p_{\sigma'}[j]$, for $0 < j \leq k$ to a quorum of servers $Q \in \mathbf{c}^p_{\sigma'}[j-1].cfg.Quorums$. And this completes the proof. $\square$

Lemma. 51 *(Configuration Prefix) Let π_1 and π_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 ξ. Let σ_1 be the state after the response step of π_1 and σ_2 the state after the response step of π_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 47 for any i such that $\mathbf{c}^{p_1}_{\sigma_1}[i] \neq \perp$ and $\mathbf{c}^{p_2}_{\sigma_2}[i] \neq \perp$, then $\mathbf{c}^{p_1}_{\sigma_1}[i].cfg = \mathbf{c}^{p_2}_{\sigma_2}[i].cfg$. Also from Lemma 49 we know that for $0 \leq j \leq \nu_1, \mathbf{c}^{p_1}_{\sigma_1}[j] \neq \perp$, and $0 \leq j \leq \nu_2, \mathbf{c}^{p_2}_{\sigma_2}[j] \neq \perp$. 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 π_2 (Line A5:2) at some state σ' before σ_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 $\pi_1 \rightarrow \pi_2$ then σ_1 appears before σ' in execution ξ. By Lemma 50, we know that by σ_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 47, 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 π_2. Since $\mathbf{c}^{p_1}_{\sigma_1}[\mu+1] \neq \perp$ then p_2 assigns $\mathbf{c}^{p_2}_{*}[\mu+1] = \mathbf{c}^{p_1}_{\sigma_1}[\mu+1]$, and repeats the process in the configuration $\mathbf{c}^{p_2}_{*}[\mu+1].cfg$. Since every configuration $\mathbf{c}^{p_1}_{\sigma_1}[j].cfg$, for $\mu \leq j < \nu_1$, 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 σ'' before σ_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. $\square$

Lemma. 52 *Let σ and σ' two states in an execution ξ such that σ appears before σ' in ξ. 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 σ, 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 σ' and hence $\mu(\mathbf{c}^{p}_{\sigma'}) \geq \mu(\mathbf{c}^{p}_{\sigma})$. $\square$

Lemma. 53 *[Configuration Progress] Let π_1 and π_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 ξ. Let σ_1 be the state after the response step of π_1 and σ_2 the state after the response step of π_2. Then $\mu(\mathbf{c}^{p_1}_{\sigma_1}) \leq \mu(\mathbf{c}^{p_2}_{\sigma_2})$.*

Proof. By Lemma 51 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 A4:2 during π_2 such that $\mathbf{c}^{p_2}_{\sigma'}[\mu_2].status = F$ in some state σ' before σ_2. If $\mu_2 \geq \mu_1$, and since σ_2 is after σ', then by Lemma 52 $\mu_2 \leq \mu(\mathbf{c}^{p_2}_{\sigma_2})$ 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 51 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 π_2, p_2 starts from the configuration at index μ_2 and in some iteration performs get-next-config in configuration $\mathbf{c}^{p_2}_{\sigma_2}[\mu_1 - 1]$. According to Lemma 47, $\mathbf{c}^{p_1}_{\sigma_1}[\mu_1 - 1].cfg = \mathbf{c}^{p_2}_{\sigma_2}[\mu_1 - 1].cfg$. Since π_1 completed before π_2, then it must be the case that σ_1 appears before σ' in ξ. However, p_2 invokes the get-next-config operation in a state σ'' which is either equal to σ' or appears after σ' in ξ. Thus, σ'' must appear after σ_1 in ξ. 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 46), 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. $\square$

Lemma. 17 *Let π be a complete reconfiguration operation by a reconfigurer rc in an execution ξ of* ARES. *if σ_1 is the state in ξ following the termination of the* read-config *action during π, then π invokes a* finalize-config$(\mathbf{c}^{rc}_{\sigma_2})$ *at a state σ_2 in ξ, 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 ρ be a reconfiguration operation $\mathsf{reconfig}(c)$. At first, ρ invokes a read-config to retrieve a latest value of the global configuration sequence, $\mathbf{c}^{rc}_{\sigma_1}$, in a state σ_1 in ξ. During the add-config action, ρ proposes the addition of c, and appends at the end of $\mathbf{c}^{rc}_{\sigma_!}$ the decision d of the consensus protocol. Therefore, if $\mathbf{c}^{rc}_{\sigma_1}$ is extended by $\langle d, P\rangle$ (Line Alg. 5:17), 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 σ_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. $\square$

Lemma. 18 *Suppose ξ is an execution of* ARES*. For any state σ in ξ, if $\mathbf{c}^{p}_{\sigma}[j].status = F$ for some process $p \in \mathcal{I}$, then there exists a* reconfig *action ρ by a reconfigurer $rc \in \mathcal{G}$, such that (i) rc invokes* finalize-config$(\mathbf{c}^{rc}_{\sigma'})$ *during ρ at some state σ' in ξ, (ii) $\nu(\mathbf{c}^{rc}_{\sigma'}) = j$, and (iii) $\mu(\mathbf{c}^{rc}_{\sigma'}) < j$.*

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 A5:33), 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 A6:10). So, (ii) is possible only if c is finalized during a reconfig operation.

Let, w.l.o.g., ρ be the first reconfiguration operation that finalizes $\mathbf{c}^{p}_{\sigma}[j].cfg$. To do so, process rc invokes finalize-config$(\mathbf{c}^{rc}_{\sigma'_1})$ during ρ, at some state σ' that appears before σ in ξ. By Lemma 47, $\mathbf{c}^{p}_{\sigma}[j].cfg = \mathbf{c}^{rc}_{\sigma'}[j].cfg$. Since, rc finalizes $\mathbf{c}^{rc}_{\sigma'}[j]$, then this is the last entry of $\mathbf{c}^{rc}_{\sigma'}$ and hence $\nu(\mathbf{c}^{rc}_{\sigma'}) = j$. Also, by Lemma 18 it follows that the read-config action of ρ returned a configuration $\mathbf{c}^{rc}_{\sigma''}$ in some state σ'' that appeared before σ' in ξ, such that $\nu(\mathbf{c}^{rc}_{\sigma''}) < \nu(\mathbf{c}^{rc}_{\sigma'})$. Since by definition, $\mu(\mathbf{c}^{rc}_{\sigma''}) \leq \nu(\mathbf{c}^{rc}_{\sigma''})$, then $\mu(\mathbf{c}^{rc}_{\sigma''}) < j$. However, since only $\langle c, P\rangle$ is added to $\mathbf{c}^{rc}_{\sigma''}$ to result in $\mathbf{c}^{rc}_{\sigma'}$, then $\mu(\mathbf{c}^{rc}_{\sigma''}) = \mu(\mathbf{c}^{rc}_{\sigma'})$. Therefore, $\mu(\mathbf{c}^{rc}_{\sigma'}) < j$ as well and the lemma follows. $\square$

Lemma. 19 *Let π_1 and π_2 be two completed read/write/reconfig operations invoked by processes p_1 and p_2 in $\mathcal{I}$, in an execution ξ of* ARES*, such that, $\pi_1 \to \pi_2$. If c_1.*put-data$(\langle \tau_{\pi_1}, v_{\pi_1}\rangle)$ *is the last* put-data *action of π_1 and σ_2 is the state in ξ, after the completion of the first* read-config *action of π_2, then there exists a c_2.*put-data$(\langle \tau, v\rangle)$ *action in some configuration $c_2 = \mathbf{c}^{p_2}_{\sigma_2}[i].cfg$, for $\mu(\mathbf{c}^{p_2}_{\sigma_2}) \leq i \leq \nu(\mathbf{c}^{p_2}_{\sigma_2})$, such that (i) it completes in a state σ' before σ_2 in ξ, and (ii) $\tau \geq \tau_{\pi_1}$.*

Proof. Note that from the definitions of $\nu(\cdot)$ and $\mu(\cdot)$, we have $\mu(\mathbf{c}^{p_2}_{\sigma_2}) \leq \nu(\mathbf{c}^{p_2}_{\sigma_2})$. Let σ_1 be the state in ξ after the completion of c_1.put-data$(\langle \tau_{\pi_1}, v_{\pi_1}\rangle)$ and σ'_1 be the state in ξ following the response step of π_1. Since any operation executes put-data on the last discovered configuration then c_1 is the last configuration found in $\mathbf{c}^{p_1}_{\sigma_1}$, and hence $c_1 = \nu^c(\mathbf{c}^{p_1}_{\sigma_1})$. By Lemma 52 we have $\mu(\mathbf{c}^{p_1}_{\sigma_1}) \leq \mu(\mathbf{c}^{p_1}_{\sigma'_1})$ and by Lemma 53 we have $\mu(\mathbf{c}^{p_1}_{\sigma'_1}) \leq \mu(\mathbf{c}^{p_2}_{\sigma_2})$, since π_2 (and thus its first read-config action) is invoked after σ'_1 (and thus after the last read-config action during π_1). Hence, combining the two implies that $\mu(\mathbf{c}^{p_1}_{\sigma_1}) \leq \mu(\mathbf{c}^{p_2}_{\sigma_2})$. Now from the last implication and the first statement we have $\mu(\mathbf{c}^{p_1}_{\sigma_1}) \leq \nu(\mathbf{c}^{p_2}_{\sigma_2})$. Therefore, we consider two cases: (a) $\mu(\mathbf{c}^{p_2}_{\sigma_2}) \leq \nu(\mathbf{c}^{p_1}_{\sigma_1})$ and (b) $\mu(\mathbf{c}^{p_2}_{\sigma_2}) > \nu(\mathbf{c}^{p_1}_{\sigma_1})$. Let for appreviation denote by $\mu_i = \mu(\mathbf{c}^{p_i}_{\sigma_i})$, and $\nu_i = \nu(\mathbf{c}^{p_i}_{\sigma_i})$.

Case (a)**:** Since $\pi_1 \to \pi_2$ then, by Lemma 51, $\mathbf{c}^{p_2}_{\sigma_2}$ value returned by read-config at p_2 during the execution of π_2 satisfies $\mathbf{c}^{p_1}_{\sigma_1} \preceq_p \mathbf{c}^{p_2}_{\sigma_2}$. Therefore, $\nu(\mathbf{c}^{p_1}_{\sigma_1}) \leq \nu(\mathbf{c}^{p_2}_{\sigma_2})$, and in this case $\mu(\mathbf{c}^{p_2}_{\sigma_2}) \leq \nu(\mathbf{c}^{p_1}_{\sigma_1}) \leq \nu(\mathbf{c}^{p_2}_{\sigma_2})$. Since c_1 is the last configuration in $\mathbf{c}^{p_1}_{\sigma_1}$, then it has index $\nu(\mathbf{c}^{p_1}_{\sigma_1})$. 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 as: (i) it completes in state σ_1 which appears before σ_2, and (ii) it puts a pair $\langle \tau, v \rangle$ such that $\tau = \tau_{\pi_1}$.

Case (b)**:** Note that there exists a reconfiguration client rc that invokes reconfig operation ρ, 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 σ be the state immediately after the read-config of ρ. Now, we consider two sub-cases: (i) σ appears before σ_1 in ξ, or (ii) σ appears after σ_1 in ξ.

Subcase $(b)(i)$: Since read-config at σ completes before the invocation of last read-config of operation π_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 51. Suppose $\mathbf{c}_\sigma^{rc} \prec_p \mathbf{c}_{\sigma_1}^{p_1}$, then according to Lemma 17 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 σ_1 is the state after the last put-data during π_1, then if σ_1' is the state after the completion of the last read-config of π_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 ρ at state σ_ρ after σ_1' and the update-config operation is invoked at state σ_ρ' after σ_ρ with a configuration sequence $\mathbf{c}_{\sigma_\rho'}^{rc}$. During the update operation ρ 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 ρ 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, ρ 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 $\nu^c(\mathbf{c}_{\sigma_1}^{p_1})$, and completed in a state σ_1 before σ_ρ', then by property **C1** of Definition 31, it follows that the get-data action will return a tag $\tau \geq \tau_{\pi_1}$. Therefore, the maximum tag that ρ discovers is $\tau_{max} \geq \tau \geq \tau_{\pi_1}$. Before invoking the finalize-config action, ρ invokes $\nu^c(\mathbf{c}_{\sigma_\rho'}^{rc})$.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 47, 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 π_2 observed configuration $\mu(\mathbf{c}_{\sigma_2}^{p_2})$, then it must be the case that σ_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 ρ. Thus, in this case both properties are satisfied and the lemma follows.

Subcase $(b)(ii)$: Suppose in this case that σ occurs in ξ after σ_1. In this case the last put-data in π_1 completes before the invocation of the read-config in ρ in execution ξ. Now we can argue recursively, ρ taking the place of operation π_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 ξ before π_2 is invoked, and hence the statement of the lemma can be shown to hold by a sequence of inequalities. □

Lemma. 20 *Let π_1 and π_2 denote completed read/write operations in an execution ξ, from processes $p_1, p_2 \in \mathcal{I}$ respectively, such that $\pi_1 \to \pi_2$. If τ_{π_1} and τ_{π_2} are the local tags at p_1 and p_2 after the completion of π_1 and π_2 respectively, then $\tau_{\pi_1} \leq \tau_{\pi_2}$; if π_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 π_1. Also, let σ_2 be the state in ξ that follows the completion of the first read-config action during π_2. Notice that π_2 executes a loop after the first read-config operation and performs c.get-data (if π_2 is a read) or c.get-tag (if π_2 is a write) from all $c = \mathbf{c}_{\sigma_2}^{p_2}[i].cfg$, for $\mu(\mathbf{c}_{\sigma_2}^{p_2}) \leq i \leq \nu(\mathbf{c}_{\sigma_2}^{p_2})$. By Lemma 19, there exists a c'.put-data($\langle\tau, v\rangle$) action by some operation π' on some configuration $c' = \mathbf{c}_{\sigma_2}^{p_2}[j].cfg$, for $\mu(\mathbf{c}_{\sigma_2}^{p_2}) \leq j \leq \nu(\mathbf{c}_{\sigma_2}^{p_2})$, that completes in some state σ' that appears before σ_2 in ξ. Thus, the get-data or get-tag invoked by p_2 on $\mathbf{c}_{\sigma_2}^{p_2}[j].cfg$, occurs after state σ_2 and thus after σ'. Since the DAP primitives used satisfy properties **C1** and **C2** of Definition 31, then the get-tag action will return a tag τ'_{π_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 π_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 π_2 is a write, then before propagating the new value the writer increments the maximum timestamp discovered (Line A7:13) generating a tag $\tau_{\pi_2} > \tau_{max}$. Therefore the operation π_2 propagates a tag $\tau_{\pi_2} > \tau_{\pi_1}$ in this case. □

D Latency Analysis

In this section we examine closely the latencies of each operation in ARES, and question the termination of each operation under various environmental conditions. For our analysis we assume that each operation inflicts constant computation overhead and delays are only introduced due to the message exchange among the processes. We will measure delays in time units of some global clock T. No process has access to T and the clock can only be accessed by an external viewer. Let d denote the minimum message delivery delay between any two processes in the service; let D be the maximum delivery delay. Also, let $T(\pi)$ denote the communication delay of an operation (or action) π.

Given the message delivery bounds d and D, and through inspection of the algorithm we can provide the delay bounds of the operations and actions used by ARES as follows:

Lemma 55. *If any message send from a process p_1 to a process p_2, s.t. $p_1, p_2 \in \mathcal{I} \cup \mathcal{S}$, takes at least d and at most D time units to be delivered, then the following operations may terminate within the following time intervals: (i) $2d \leq T(\text{put-config}) \leq 2D$ and (ii) $2d \leq T(\text{read-next-config}) \leq 2D$.*

From Lemma 55 we can derive the delay of a read-config action.

Lemma 56. *For any* read-config *action ϕ such that it accepts an input seq and returns seq', if $\mu = \mu(seq)$ and $\nu = \nu(seq')$ then ϕ takes:*

$$4d(\nu - \mu + 1) \leq T(\phi) \leq 4D(\nu - \mu + 1) \tag{1}$$

From Lemma 56 it is apparent that the latency of a read-config action highly depends on the number of configurations installed, following the last finalized configuration as that was known by the process at the invocation of the read-config action. Let $\lambda = \nu - \mu$ denote the number of newly installed configurations. Now let us examine when a new configuration gets inserted in the configuration sequence by a reconfig operation. By ARES a reconfig operation has four phases: (i) reads the latest configuration sequence, (ii) adds the new configuration at the end of the sequence, (iii) transfers the knowledge to the added configuration, and (iv) 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 will observed 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.

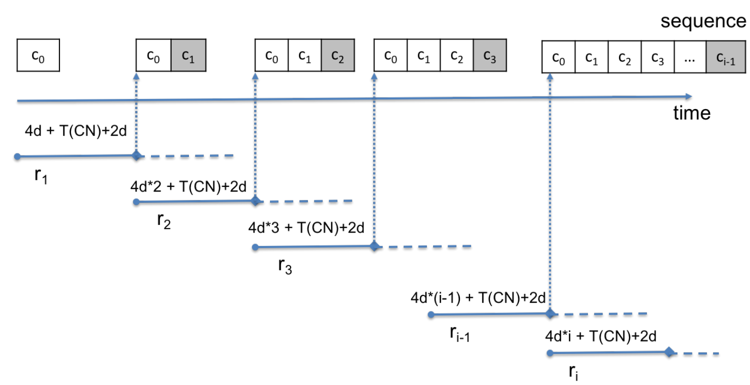

Figure 2: Successful reconfig operations.

More formally we can build an execution where all reconfig operations append their configuration in the configuration sequence. Consider a partial execution ξ that ends in a state σ. Suppose that every process $p \in \mathcal{I}$ knows the same configuration sequence, $\mathbf{c}_\sigma^p = \mathbf{c}_\sigma$. 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 $\mathbf{c}_{\sigma_0}^p$. We extend ξ_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 th reconfiguration operations $rc_1, rc_2, \ldots, rc_i$. We can now compute the minimum latency we need to add k new configurations in the configuration sequence starting from the state σ of execution ξ. For simplicity of our analysis we assume that any consensus instance takes the same time to terminate and that is $T(CN)$.

Lemma 57. *Starting from the last state of ξ, σ, and given that d is the minimum communication delay, then k configurations can be appended to $\mathbf{c}_\sigma$, in time: $T(k) \geq 4d\sum_{i=1}^{k} i + k\,(T(CN) + 2d)$ in our execution construction.*

Proof. Figure 2 shows the timings of each reconfiguration operation. In particular, consider the first reconfiguration rc_1. During its read-config rc_1 does not discover new configurations and thus, if seq_1 is the input and seq'_1 the output configuration, $\mu(seq_1) = \nu(seq'_1)$. Thus, by Lemma 56, the read-config takes at least time $4d$. Since the consensus algorithm takes $T(CN)$ and the put-config action at least $2d$ (see Lemma 55), then rc_i takes time at least:

$$T(1) \geq 4d + T(CN) + 2d$$

to install configuration c_1. Reconfiguration rc_2, executes two iterations during its read-config action, and $\nu(seq'_2) = \mu(seq_2) + 1$. Thus, by Lemma 56, the read-config of rc_2 takes at least time $4d * 2$. Until rc_2 installs c_2 it needs also time $T(CN)$ for the consensus algorithm and $2d$ for the put-config action. Hence, both configurations c_1 and c_2 are appended in time at least:
$T(2) \geq T(1) + 8d + T(CN) + 2d$
$\geq 12d + 2\,(T(CN) + 2d)$

So in general to install the configuration k it takes time at least $4d * k + T(CN) + 2d$, and thus to append the sequence with all the configurations $c_1, \ldots, c_k$ it takes time at least:

$$T(k) \geq T(k-1) + 4kd + T(CN) + 2d \geq 4d\sum_{i=1}^{k} i + k\,[T(CN) + 2d]$$

And that completes our proof. □

Finally, we can compute the maximum time that a read/write operation needs before completing. From close examination of the algorithm, the DAPs used by ARES have an impact on the delay of a read and write operation. For our analysis we assume that all get-data, get-tag and put-data primitives use two phases of communication as this capture the DAP we present in this work as well as the most common implementations of atomic registers.

Lemma 58. *If any message send from a process p_1 to a process p_2, s.t. $p_1, p_2 \in \mathcal{I} \cup \mathcal{S}$, takes at least d and at most D time units to be delivered, then the DAPs may terminate in: (i) $2d \leq T(\textsf{put-data}) \leq 2D$; (ii) $2d \leq T(\textsf{get-tag}) \leq 2D$; and (iii) $2d \leq T(\textsf{get-data}) \leq 2D$.*

Having the delays for the DAPs we can now compute the delay of a read/write operation π.

Lemma 59. *Let σ_s and σ_e be the states before the invocation and after the completion step of a read/write operation π by p respectively, in some execution ξ of* ARES,*. Then π takes time at most:*

$$T(\pi) \leq 6D\left[\nu(\mathbf{c}^p_{\sigma_e}) - \mu(\mathbf{c}^p_{\sigma_s}) + 2\right]$$

Proof. 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 σ_1 be the state when the first read-config, denoted by read-config$_1$, action terminates. By Lemma 56 the action will take time:

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

The get-data action that follows the read-config (Lines A7:34- A7:35) is also executed at most $(\nu(\mathbf{c}^p_{\sigma_1}) - \mu(\mathbf{c}^p_{\sigma_s}) + 1)$ given that no new finalized configuration was discovered by the read-config action. Finally, the put-data and the second read-config actions of π 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 π can be at most:

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

And the lemma follows. □

It remains now to examine if a read/write operation may "catch up" with any ongoing reconfigurations. For the sake of a worst case analysis we will assume that reconfiguration operations may communicate respecting the minimum delay d, whereas read and write operations suffer the maximum delay D in each message exchange. We will split our analysis into three directions, with respect to the number of configurations installed k, and the bound on the minimum delay d: (i) k is finite, and d may be unbounded small; (ii) k is infinite, and d may be unbounded small; (iii) k is infinite, and d can be bounded.

k is finite, and d may be unbounded small. In this case we assume a finite number of installed configurations. Also, as the d is unbounded, it follows that reconfigurations may be installed almost instantaneously. Let us first examine what is the maximum delay bound of a any read/write operation.

k is infinite, and d is bounded. We will compute the bound on d with respect to the D and the number of configurations to be installed k if we want to allow a read/write operation to reach ongoing reconfigurations.

Lemma 60. *A read/write operation π may terminate in any execution ξ of* ARES *given that k configurations are installed during π, if $d \geq \frac{3D}{k} - \frac{T(CN)}{2(k+2)}$*

Proof. By Lemma 57, k configurations may be installed in at least: $T(k) \geq 4d\sum_{i=1}^{k} i + k\left(T(CN) + 2d\right)$. Also by Lemma 59, we know that operation π 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 π, then π may terminate before a $k+1$ configuration is added in the configuration sequence if $6D(k+2) \leq 4d\sum_{i=1}^{k} i + k\left(T(CN) + 2d\right)$ then we have $d \geq \frac{3D}{k} - \frac{T(CN)}{2(k+2)}$. And that completes the lemma. $\square$

5 Efficient state transfer during reconfiguration

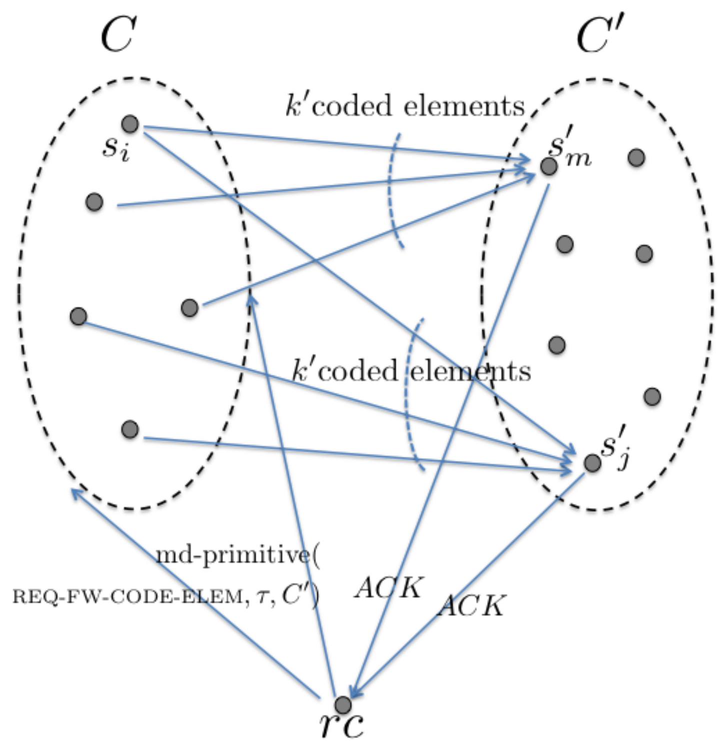

Figure 3: An illustration of the recon client rc transferring data from servers of configuration C to those in C'.