Ares II: Tracing the Flaws of a (Storage) God

CHRYSSIS GEORGIOU, University of Cyprus, Cyprus
NICOLAS NICOLAOU, Algolysis Ltd, Cyprus
ANDRIA TRIGEORGI, University of Cyprus, Cyprus and Algolysis Ltd, Cyprus

Ares is a modular framework, designed to implement dynamic, reconfigurable, fault-tolerant, read/write and strongly consistent distributed shared memory objects. Recent enhancements of the framework have realized the efficient implementation of large objects, by introducing versioning and data striping techniques. In this work, we identify performance bottlenecks of the Ares's variants by utilizing distributed tracing, a popular technique for monitoring and profiling distributed systems. We then propose optimizations across all versions of Ares, aiming in overcoming the identified flaws, while preserving correctness. We refer to the optimized version of Ares as Ares II, which now features a piggyback mechanism, a garbage collection mechanism, and a batching reconfiguration technique for improving the performance and storage efficiency of the original Ares. We rigorously prove the correctness of Ares II, and we demonstrate the performance improvements by an experimental comparison (via distributed tracing) of the Ares II variants with their original counterparts.

Keywords: Distributed shared storage, Strong consistency, Reconfiguration, Distributed tracing, Optimization.

1 INTRODUCTION

Distributed Shared Memory Emulation. In an era where data is being generated at an unprecedented rate, effectively dealing with Big Data challenges has become a critical endeavor. To manage this large amount of data, organizations are increasingly adopting large-scale systems known as Distributed Storage Systems (DSS) which can divide the data across multiple servers for high availability, data redundancy, and recovery purposes. One of the fundamental structures to implement a DSS is a Distributed Shared Memory (DSM) emulation.

For three decades, a series of works (e.g., [9, 13, 16, 18, 27]) proposed solutions for building DSM emulations, allowing data to be shared concurrently offering basic memory elements, i.e. registers, with strong consistency guarantees. Linerazibility (atomicity) [23] is the most challenging, yet intuitive consistency guarantee that such solutions provide. Attiya, Bar-Noy and Dolev [9] present the first fault-tolerant emulation of atomic shared memory in an asynchronous message passing system, also known as ABD. Subsequent algorithms (eg., [13, 16, 18]) have built upon ABD, aiming to reduce communication overhead while imposing minimal computation overhead. The problem of keeping copies consistent becomes even more challenging when the set of servers need to be modified, leading to the development of dynamic solutions and reconfiguration services. Examples of reconfigurable storage algorithms are RAMBO [20], DynaStore [6], SM-Store [26], SpSnStore [14], and Ares [28].

ARES and its Extensions. Ares is a modular framework, designed to implement reconfigurable, fault-tolerant, read/write distributed linearizable (atomic) shared memory objects. Unlike other reconfigurable algorithms, Ares does not define the exact methodology to access the object replicas. Rather, it relies on *data access primitives* (DAPs), which are used for expressing the data access strategy (i.e., how they retrieve and update the object data) of different shared memory algorithms (e.g., ABD). One DAP implementation supports Erasure Coding, making Ares the first reconfigurable DSM providing a level of data striping. Recently, in [19], two extensions of Ares have been proposed. The first, called CoAres, extends Ares by providing *coverability* [29]. Coverability extends linearizability by ensuring that a write operation is performed on the latest "version" of the object (see Section 2). CoAresF extends CoAres by utilizing the fragmentation (striping) strategy of CoBFS [8], making Ares suitable for handling large objects (such as files). Section 3 overviews these three variants of Ares. Many versions of Ares are obtained, when these variants are used with different DAP implementations. Experimental evaluations [19, 34] of the various versions of Ares, while revealing interesting trade-offs, they also suggested that there is still room for improvement, especially when compared to commercial DSSs. An initial speculation was hindering that the reconfiguration mechanism of Ares, which is common to all versions, causes redundant communication. Thus, there was a need for a more thorough investigation of the communication deficiency and overall performance of Ares.

Distributed Tracing. Traditionally, in distributed shared memory emulations (e.g. [10, 11, 15–17, 22, 25, 32–34]), performance analysis and bottleneck detection have often focused on measuring the overall latency of operational or computational time, rather than examine the latency of individual components in detail. This approach provides a high-level view of the system's performance but may lack the level of detail required to precisely identify the underlying causes of the bottlenecks. In recent years, the realm of distributed systems has witnessed the emergence of innovative software monitoring tools. These tools, often referred to as tracing tools, are designed to trace and visualize the interactions within a distributed system. The technique behind the tools, known as *Distributed Tracing*, has gained significant traction and has been implemented by most of the major actors of Cloud-Computing for their own monitoring needs, e.g., Google [30], Twitter [7], Uber [35], Sigelman [2, 31]. Unlike logging, which merely records events and data, distributed tracing offers a comprehensive perspective on how requests flow through interconnected components, enabling a better understanding of performance and dependencies in distributed systems.

Contributions. In this work we bring distributed tracing into the realm of DSM and demonstrate its usefulness by turning the identified flaws of Ares into optimizations, yielding Ares II.
Distributed tracing. We identify performance bottlenecks in different versions of Ares using OpenTelemetry [3], an open-source tool for distributed tracing. We seamlessly integrate this tool with other observability tools, such as Jaeger and Grafana Jaeger, enabling us to combine its tracing data with metrics, logs, and visualizations. This integration allows us to trace requests as they traverse various components, providing a holistic view of the algorithms' behavior.
ARES II. Once bottlenecks are detected, we develop optimizations that ensure a performance boost while preserving the correctness conditions of the original algorithms, leading to Ares II. The optimizations mainly concern the reconfiguration mechanism, and are as follows: (i) to expedite configuration discovery we introduce piggy-back data on read/write messages; (ii) for service longevity and to expedite configuration discovery, we introduce a garbage collection mechanism that removes obsolete configurations and updates older configurations with newly established ones; and (iii) to expedite reconfiguration we introduce a batching mechanism where a single configuration is applied not on a single but multiple objects concurrently; this is particularly useful for the fragmented version of Ares II. We rigorously prove the correctness of Ares II, and we demonstrate the performance improvements by an experimental comparisons of the Ares II versions with their original counterparts.

2 SYSTEM SETTINGS AND DEFINITIONS

We explore an asynchronous message-passing system with processes communicating through reliable point-to-point channels, allowing potential message reordering.

Clients and servers. The system is a collection of crash-prone, asynchronous processors with unique identifiers (ids) from a totally-ordered set, composed of two main disjoint sets of processes: (a) a set $\mathcal{I}$ of client processes ids that may perform operations on a replicated object, and (b) a set $\mathcal{S}$ of server processes ids; servers host and maintain replicas of shared data. A *quorum* is defined as a subset of $\mathcal{S}$. A *quorum system* [36] is a collection of pair-wise intersecting quorums.
In this work, we deal with dynamic environments, where the *configuration* of the system may dynamically change over time due to servers removal or addition. A configuration is a data type

that describes the service setup, i.e., the finite set of servers, their grouping into intersecting sets (i.e., quorum system), and various parameters needed for the implementation of the atomic storage service (see Section 3 for the precise contains of Ares's configurations). Dynamic addition and removal of servers may lead the system from one configuration to another, with different set of servers and parameters. *Reconfiguration* is the process responsible to migrate the system from one configuration to the next. There are three distinct sets of client processes: a set $\mathcal{W}$ of writers, a set $\mathcal{R}$ of readers, and a set $\mathcal{G}$ of reconfiguration clients. Each writer is allowed to modify the value of a shared object, and each reader is allowed to obtain the value of that object. Reconfiguration clients attempt to introduce new configurations to the system in order to mask transient server errors or include new servers and to ensure the longevity of the service.

Executions, histories and operations. An execution ξ of a distributed algorithm A is a sequence of states and actions reflecting real-time evolution. The history H_ξ is the subsequence of actions in ξ. An operation π is invoked in ξ when its invocation appears in H_ξ, and it responds when the matching response is present. An operation is *complete* in ξ when both its invocation and matching response appear in H_ξ in order. H_ξ is sequential if it starts with an invocation, with each invocation immediately followed by its matching response; otherwise, it is concurrent. H_ξ is complete if every invocation has a matching response, ensuring that each operation in ξ is complete. π *precedes* (or *succeeds*) π' in real-time in ξ, denoted $\pi \rightarrow \pi'$, if π's response appears before π''s invocation in H_ξ. Two operations are concurrent if neither precedes the other.

Consistency. We consider *linearizability* [23] and *coverability* [29] of R/W objects. A complete history H_ξ is *linearizable* if there exists some total order on the operations in H_ξ s.t. it respects the real-time order $\rightarrow$ of operations, and is consistent with the semantics of operations. *Coverability* extends linearizability by ensuring that a write operation is performed on the latest *version* of the object. In particular, coverability is defined over a *totally ordered* set of versions, and introduces the notion of *versioned (coverable) objects*. Per [29], a coverable object is a type of a R/W object where each value written is assigned with a version, and a write succeeds only if its associated version is the latest one; otherwise, the write becomes a read operation.

Fragmented Objects and Fragmented coverability. As defined in [8], a *block object* is a concurrent R/W object with a bounded value domain. A *fragmented object* is a totally ordered sequence of *block* objects, initially containing an empty block. *Fragmented coverability* [8] is a consistency property defined over fragmented objects. It guarantees that concurrent write operations on different coverable blocks of the fragmented object would *all* prevail (as long as each write is tagged with the latest version of each block), whereas only one write on the same block eventually prevails (all other concurrent writes on the same block would become read operations). Thus, a fragmented object implementation satisfying this property may lead to higher access concurrency [8].

Tags. We use logical tags τ as pairs (ts, wid) to order operations, where $ts \in \mathbb{N}$ is a timestamp, and $wid \in \mathcal{W}$ is a writer ID. Let $\mathcal{T}$ be the set of all tags. For any $\tau_1, \tau_2 \in \mathcal{T}$, $\tau_2 > \tau_1$ if (i) $\tau_2.ts > \tau_1.ts$ or (ii) $\tau_2.ts = \tau_1.ts$ and $\tau_2.wid > \tau_1.wid$. Each tag is associated with a value of the object.

3 OVERVIEW OF ARES AND ITS EXTENSIONS

Ares [28] is a reconfigurable algorithm, designed as a modular framework to implement dynamic, fault-tolerant, read/write distributed linearizable (atomic) shared memory objects. We first present Ares, and then we overview two recently proposed extensions of Ares.

Main components. Ares consists of three major components: (i) Reconfiguration Protocol: Handles the introduction and installation of new configurations. Reconfiguration clients propose and install new configurations via the reconfig operation. (ii) Read/Write Protocol: Executes read and write operations invoked by readers and writers, respectively. (iii) DAP Implementation:

Implements Distributed Atomic Primitives (DAPs) for each installed configuration. These primitives respect certain properties and are used by reconfig, read, and write operations (more below).

Configurations in Ares. A *configuration* $c \in \mathcal{C}$, $\mathcal{C}$ being a set of unique identifiers, comprises: (i) $c.Servers \subseteq \mathcal{S}$: a set of server identifiers; (ii) $c.Quorums$: the set of quorums on $c.Servers$, s.t. $\forall Q_1, Q_2 \in c.Quorums, Q_1, Q_2 \subseteq c.Servers$ and $Q_1 \cap Q_2 \neq \emptyset$; (iii) $DAP(c)$: the set of primitives that clients in $\mathcal{I}$ may invoke on $c.Servers$; and (iv) $c.Con$: a consensus instance with the values from $\mathcal{C}$, implemented on servers in $c.Servers$. We refer to a server $s \in c.Servers$ as a *member* of configuration c.

DAPs. Ares allows for reconfiguration between completely different protocols in principle, as long as they can be expressed using three DAPs: (i) the get-tag, which returns the tag of an object, (ii) the get-data, which returns a $\langle tag, value \rangle$ pair, and (iii) the put-data($\langle tag, value \rangle$), which accepts a $\langle tag, value \rangle$ as an argument.
For the DAPs to be useful, they need to satisfy a property, referred in [28] as **Property 1**, which involves two conditions: **(C1)** if a put-data($\langle \tau, v \rangle$) precedes a get-data (or get-tag) operation that returns τ', then $\tau' \geq \tau$, and **(C2)** if a get-data returns $\langle \tau', v' \rangle$ then there exists put-data($\langle \tau', v' \rangle$) that precedes or is concurrent to the get-data operation.
In [28], two different atomic shared R/W algorithms were expressed in terms of DAPs. These are the DAPs for the ABD algorithm [9], and the DAPs for an erasure coded based approach presented for the first time in [28]. Both DAPs were shown to satisfy Property 1. In the rest of the manuscript we refer to the two DAP implementations as ABD-DAP and EC-DAP, respectively. ABD-DAP focuses on data consistency through replication, while EC-DAP offers fault tolerance and data protection by dividing data into smaller fragments with redundancy.

Configuration sequence and sequence traversal. A configuration sequence *cseq* in Ares is defined as a sequence of pairs $\langle c, status \rangle$ where $c \in \mathcal{C}$, and $status \in \{P, F\}$, where P stands for pending and F for finalized. Configuration sequences are constructed and stored in clients, while each server in a configuration c only maintains the configuration that follows c in a local variable $nextC \in \mathcal{C} \cup \{\bot\} \times \{P, F\}$. This way, Ares attempts to construct a global distributed sequence (or list) of configurations $\mathcal{G}_L$.
Any read/write/reconfig operation utilizes the *sequence traversal* mechanism which consists of three actions: (i) get-next-config(), to discover the next configuration, (ii) put-config(), which writes back *nextC* to a quorum of servers, to ensure that a state is discoverable by any subsequent operation, and (iii) read-config(), which finally returns the updated configuration sequence.

Reconfiguration operation. To perform a reconfiguration operation reconfig(c), a client r follows 4 steps: (i) It executes a sequence traversal to discover the latest configuration sequence *cseq*. (ii) It attempts to add $\langle c, P \rangle$ at the end of *cseq* by proposing c to a consensus mechanism. The outcome of the consensus may be a configuration c' (possibly different than c) proposed by some reconfiguration client. (iii) The client determines the maximum tag-value pair of the object, say $\langle \tau, v \rangle$ by executing get-data and transfers the pair to c' by performing put-data($\langle \tau, v \rangle$) on c'. (iv) Once the update of the value is complete, r *finalizes* the proposed configuration by setting $nextC = \langle c', F \rangle$ in a quorum of servers of the last configuration in its *cseq*.
In [28], it was shown that this reconfiguration procedure guarantees that if $cseq_p$ and $cseq_q$ are configuration sequences obtained by any two clients p and q, then either $cseq_p$ is a prefix of $cseq_q$, or vice versa.

Read/Write operations. A write (or read) operation π by a client p is executed by performing the following actions: (i) π invokes a read-config action to obtain the latest configuration sequence *cseq*, (ii) π invokes a get-tag (if a write) or get-data (if a read) in each configuration, starting from the last finalized to the last configuration in *cseq*, and discovers the maximum τ or $\langle \tau, v \rangle$ pair respectively, and (iii) repeatedly invokes put-data$(\langle \tau', v' \rangle)$, where $\langle \tau', v' \rangle = \langle \tau + 1, v' \rangle$ if π is a write and $\langle \tau', v' \rangle = \langle \tau, v \rangle$ if π is a read in the last configuration in *cseq*, and read-config to discover any new configuration, until no additional configuration is observed.

Extension 1: CoAres is a coverable version of Ares, introduced in [19]. Recall from Section 2 that coverability guarantees that writing to an object succeeds when connecting the new value with the "current" version of the object; otherwise, the write turns into a read operation that provides the latest version and its associated value. CoAres uses an *optimized* DAPs implementation presented in [19]. This optimization reduces unnecessary data transfers by having servers send only the $\langle tag, value \rangle$ pairs with higher tags than the client's tag in get-data. Also, put-data only occurs if the maximum tag exceeds their local one.

Extension 2: CoAresF, introduced in [19], is the fragmented version of CoAres that is suitable for fragmented objects and guarantees fragmented coverability (cf. Section 2). It is obtained by the integration of CoAres with the CoBFS framework presented in [8]. As a fragmented object is a sequence of blocks (fragments), write operations can modify individual blocks, leading to higher access concurrency. CoBFS is composed of two modules: (i) a Fragmentation Module (FM), and (ii) a Distributed Shared Memory Module (DSMM). In brief, the FM implements the fragmented object, while the DSMM implements an interface to a shared memory service that allows operations on individual block objects. To this respect, CoBFS is flexible enough to utilize any underlying DSM implementation, including CoAres. When CoAres is used as the DSMM with EC-DAP, we obtain a two-level striping (one level from fragmentation and one from Erasure Coding) reconfigurable DSM providing strong consistency and high access concurrency for large (fragmented) objects.

Remark: We note that both extensions of Ares mainly involve the read/write operations (applied on coverable or fragmented objects). All three versions share the same reconfiguration mechanism, thus any optimization introduced on the reconfiguration operation of Ares, follows naturally on its extensions as well (with some minor modifications). Thus, in Section 5, for simplicity of presentation, we present optimizations on the reconfiguration mechanism of the original Ares.

4 TRACING BOTTLENECKS

We first provide some additional background on distributed tracing and then we proceed to provide the details on how we have applied distributed tracing on the different version of Ares.

Distributed Tracing is a technique used to monitor and profile distributed systems by tracing individual requests or transactions as they move across multiple components and systems. It involves adding code to the distributed system to gather detailed data on the flow of requests and the behavior of each component. This data is then combined and analyzed to gain a better understanding of the system's overall performance and to identify any problems or bottlenecks. In particular, the distributed tracing method creates *traces*, which are records of the activity of individual requests as they pass through various microservices in a distributed system. These traces can be used to diagnose and debug problems in the system, as well as to gain insights into how the system is operating. Also, the individual units of work within a trace are called *spans*. Each span corresponds to a specific piece of work that is performed as a request passes through a microservice. Spans can be used to track the performance of individual components of a system and to identify bottlenecks or other issues that may be impacting system performance.

Distributed tracing is implemented by various open-source and commercial tools. Some of the popular implementations and tools for distributed tracing include: OpenTelemetry [2], Datadog [12], Google Cloud Trace [21] etc. OpenTelemetry is a popular open-source project maintained by Cloud Native Computing Foundation(CNCF) which resulted from merging two mature technologies: OpenTracing and OpenCensus. It is an observability framework that provides a standard way to

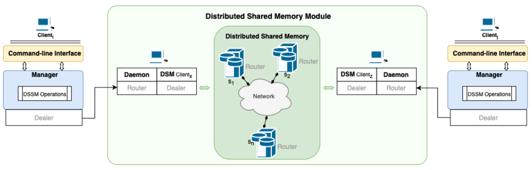


Fig. 1. The architecture of our implementation.

collect telemetry data from distributed systems, including tracing, metrics, and logs. It provides integration with various backends, such as Jaeger [35] and Zipkin [5]. Both collectors (Jaeger and Zipkin) can store the collected data in-memory, or persistently with a supported backend such as Apache Cassandra or Elasticsearch. For the visualization of collected data into meaningful charts and plots, a proposed tool *Grafana Jaeger* [1]. Grafana is a popular open-source platform for visualizing and analyzing data. It has built-in support for Jaeger, allowing for visualization of Jaeger tracing data through Grafana.

4.1 EXPERIMENTAL EVALUATION SETUP

In this section, we describe the setup of the distributed tracing used in this work.

OpenTelemetry embedding and Jaeger setup. Incorporating OpenTelemetry, we trace read/write/reconfig requests, track timing, and interactions within our implementation, exporting collected data to Jaeger [35]. Our Jaeger setup uses Cassandra as backend storage with an external lab URL accessible via web browsers. We also visualize traces through Grafana Jaeger integration and extend Jaeger's TTL from 2 days to "forever" to retain traces indefinitely in Cassandra storage, preventing automatic deletion.

Evaluated Algorithms. We have evaluated the performance of the following algorithms:

- AresABD: This is the version of Ares that uses the ABD-DAP implementation.
- CoAresABD: This is the version of CoAres that uses the optimized ABD-DAP implementation (see Section 3).
- CoAresABDF: This is the version of CoAresF that used the optimized ABD-DAP implementation; i.e., it is the fragmented version of CoAresABD.
- AresEC: This is a version of Ares that uses the EC-DAP implementation.
- CoAresEC: This is CoAres that uses the optimized EC-DAP implementation.
- CoAresECF: This is the two-level data striping algorithm obtained when CoAresF is used with the optimized EC-DAP implementation; i.e., it is the fragmented version of CoAresEC.

We have implemented all these algorithms using the same code and communication libraries, based on the architecture in Fig. 1. The system consists of two main modules: (i) a Manager (User Layer), and (ii) a Distributed Shared Memory Module (DSMM Layer). The Manager provides a client interface (CLI) for DSM access, with each client having its manager handling commands. In this setup, clients access the DSMM via the Manager, while servers maintain shared objects through the DSMM. The Manager uses the DSMM as an external service for read and write operations, allowing flexibility in utilizing various DSM algorithms. The algorithms are all written in Python, and we achieve asynchronous communication between layers using DEALER and ROUTER sockets from the ZeroMQ library [4].
In the remainder, for simplicity, we will refer to AresABD, CoAresABD and CoAresABDF as ABD-based algorithms and AresEC, CoAresEC and CoAresECF as EC-based algorithms.

Procedures of interest. Using OpenTelemetry Python Library [3], we monitor the *communication* and *computational* overheads of read, write and reconfig operations in both User and DSMM debug levels.

Distributed Experimental Setup on Emulab. We used 39 physical machines on a LAN with no delays or packet loss. The nodes had 2.4 GHz 64-bit Quad Core Xeon E5530 "Nehalem" processor and 12 GB RAM. A physical controller node orchestrated the experiments, while servers were on different machines. Clients were deployed in a round-robin fashion, with some machines hosting multiple client instances. For instance, with 38 machines, 11 servers, 5 writers, and 50 readers, servers used the first 11 machines, and the rest hosted readers, with 5 of them also running as writers.

Parameters of algorithms: In EC-based algorithms, quorum size is determined by $\lceil \frac{n+k}{2} \rceil$, while in ABD-based ones, it is $\lfloor \frac{n}{2} \rfloor + 1$. Here, n is the total server count, k is the number of encoded data fragments, and m represents parity fragments (i.e., $n-k$). In EC-based algorithms, higher k increases quorum size but results in smaller coded elements, while a high k and low m mean less redundancy and lower fault tolerance. When $k = 1$, it is equivalent to replication. The parameter δ in EC-based algorithms signifies the maximum concurrent put-data operations, i.e., the number of writers.

4.2 EXPERIMENTAL SCENARIOS AND RESULTS

We outline the scenarios and their settings. In each scenario, a writer initializes the system by creating a text file with a specific initial size. As writers update the file, its size grows. While these experiments use random byte strings in text files, our implementations accommodate various file types.
Readers and writers use stochastic invocation with random times in intervals of $[1...rInt]$ and $[1..wInt]$ (where $rInt, wInt = 3sec$). Each read/write client performs 50 operations, and reconfigurers, if present, perform 15 reconfigurations at intervals of $15sec$.

We present three types of scenarios:

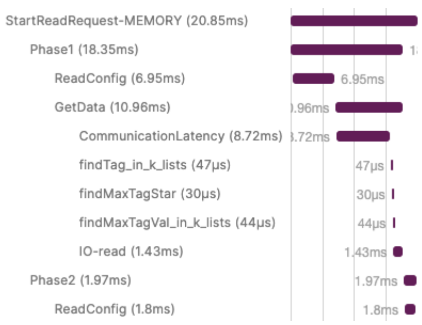
- File Size: examine performance when using different initial file sizes.
- Participation Scalability: examine performance as the number of service participants increases.
- Block Sizes: examine performance under different block sizes (only for fragmented algorithms).
- EC Parameter k: examine performance with varying k encoded data fragments (for EC-DAP algorithms).
- Longevity: examine performance with reconfigurers switching between DAPs and random server changes under various reconfigurer counts.

4.3 Results and Findings

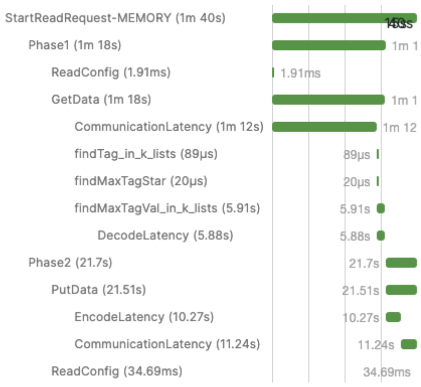
Below, we provide Grafana Jaeger-trace graphs, displaying sample traces with average duration for each scenario. Note that all scenarios underwent at least three executions for reliable results.

4.3.1 *File Size.*

The scenario is made to measure the performance of algorithms when we vary the size of the shared object, f_{size}, from 1 MB to 512 MB by doubling the size in each simulation run. The maximum, minimum and average block sizes (*rabin fingerprints* parameters), for the fragmented algorithms, were set to 1 MB, 512 kB and 512 kB respectively.For EC-based algorithms we used parity $m = 5$ yielding quorum sizes of 9 and for ABD-based algorithms we used quorums of size 6. We fixed the number of concurrent participants to $|\mathcal{W}| = 5, |\mathcal{R}| = 5, |\mathcal{S}| = 11$.



(a) algorithm:CoAresECF



(b) algorithm:AresEC

Fig. 2. READ Operation - S:11, W:5, R:5, fsize: 512MB

Results: The read latency in CoAresECF involves multiple *block read* requests, fragmenting the file into blocks and executing each block read on shared memory (DSMM). Fig. 2 shows the time it takes for two read operations to complete on the shared memory (one block in CoAresECF and the whole object in AresEC). From Fig. 2 we observe that the communication and computation latencies vary, while the latency of the read-config operation incurs a stable overhead regardless of the object size. It is worth noting that in this experiment the configuration remains unchanged, and thus time spent for configuration discovery is unnecessary. Due to read-config, fragmented algorithms also suffer a stable overhead for each block of the file. This overhead increases when handling large objects split into many blocks, ultimately defeating the purpose of fragmentation.

4.3.2 Participation Scalability. This scenario is constructed to analyze the read and write latency of the algorithms, as the number of readers, writers and servers increases. We varied the number of readers $|R|$ from the set $\{5, 15, 50\}$ and the number of writers from the set $\{5, 10, 15, 20\}$, while the number of servers $|S|$ varies from 3 to 11. We calculate all possible combinations of readers, writers and servers where the number of readers or writers is kept to 5. The size of the file used is 4 MB. The maximum, minimum and average block sizes were set to 1 MB, 512 kB and 512 kB respectively. To match the fault-tolerance of ABD-based algorithms, we used a different parity for EC-based algorithms (except in the case of 3 servers to avoid replication). With this, the EC client has to wait for responses from a larger quorums. The parity value of the EC-based algorithms is set to m=1 for $|\mathcal{S}| = 3$ and m=5 for $|\mathcal{S}| = 11$.
Results: In Figs. 3 and 4, we observe AresEC read operations in DSMM with varying server counts. A significant difference is seen between 3 and 11 servers, with the latter having a shorter read time due to reduced communication latency in $Phase1$. This is because more servers lead to smaller message sizes. However, $DecodeLatency$ and $EncodeLatency$ are slightly longer with 11 servers. CoAresEC shows a similar pattern but with smaller latencies due to optimization.
In contrast, the read latency of CoAresECF in the USER level remains consistent with increasing server count since the object is already divided at the USER level, and DSMM further divides each block, reducing data transfers and improving read latency.

4.3.3 Block Sizes. This scenario provides insights into the performance of the read and write operations at different block sizes, highlighting the impact of block sizes variations on the overall

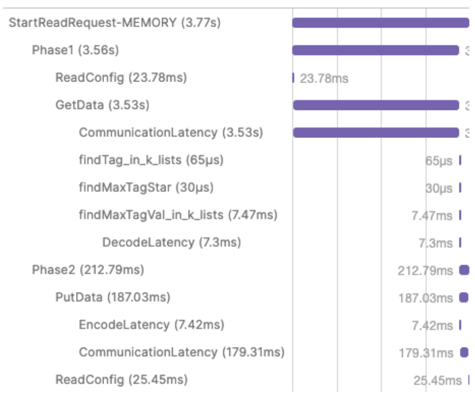


Fig. 3. READ Operation - algorithm: AresEC, S:3, W:5, R:50, fsize:4MB, Debug Level:DSMM

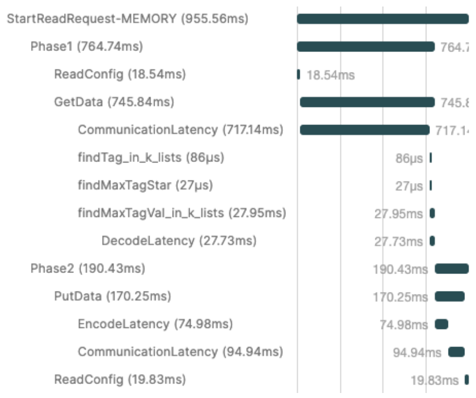


Fig. 4. READ Operation - algorithm: AresEC, S:11, W:5, R:50, fsize:4MB, Debug Level:DSMM

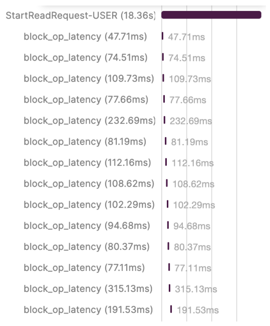


Fig. 5. READ Operation - algorithm: CoAresECF, S:11, W:5, R:5, fsize:512MB, Min/Avg Block Size:2MB, max Block Size:4MB, Debug Level:USER

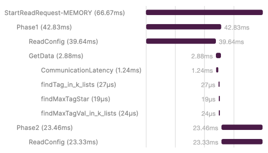


Fig. 6. READ Operation - algorithm: CoAresECF, S:11, W:5, R:5, fsize:512MB, Min/Avg Block Size:2MB, max Block Size:4MB, Debug Level:DSMM

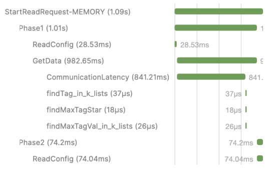


Fig. 7. READ Operation - algorithm: CoAresECF, S:11, W:5, R:5, fsize:512MB, Min/Avg Block Size:64MB, max Block Size:128MB, Debug Level:DSMM

duration of the operations. We varied the minimum and average b_{sizes} from 2 MB to 64 MB and the maximum b_{size} from 4 MB to 128 MB. The number of servers $|S|$ is fixed to 11, the number of writers $|W|$ and readers $|R|$ to 5. The parity value of CoAresECF is set to $m = 5$. The size of the initial file used was set to 512 MB.

Results: In previous experiments, we observed that CoAresF experiences higher read and write latencies with larger block sizes. The reason becomes clear when examining the details. For instance, in the read latency with minimal block sizes (Fig. 5), the reader must retrieve numerous small blocks, each taking a short time. In contrast, the read latency with maximum block sizes involves fewer but larger blocks, impacting the *CommunicationLatency*, *EncodeLatency*, and *DecodeLatency* in DSMM (not depicted due to fast reads).

4.3.4 EC *Parameter k.*

This scenario applies only to EC-based algorithms since we examine how the read and write latencies are affected as we modify the erasure-code fragmentation parameter k (a parameter of Reed-Solomon). We assume 11 servers and we increase k from 2 to 10. The number of writers (and hence the value of δ) are set to 5. The number of readers is fixed to 15. The size of

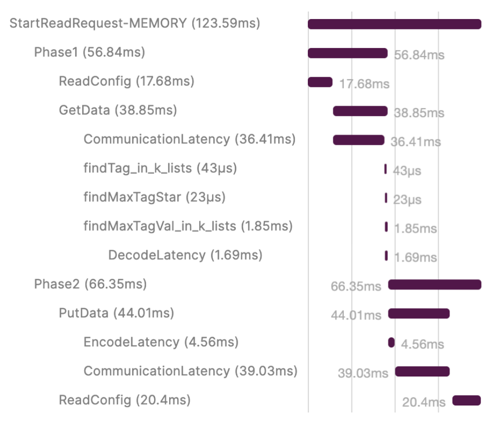


Fig. 8. READ Operation - algorithm: CoAresECF, S:11, k:1, W:5, R:5, fsize:4MB, Debug Level:DSMM

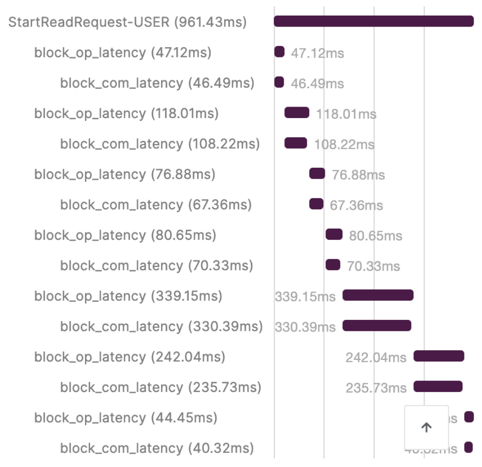


Fig. 9. READ Operation - algorithm: CoAresECF, S:11, k:10, W:5, R:5, fsize:4MB, Debug Level:DSMM

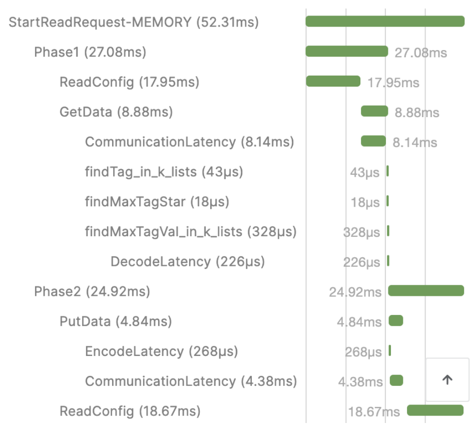


Fig. 10. READ Operation - algorithm: CoAresECF, S:11, k:1, W:5, R:5, fsize:4MB, Debug Level:USER

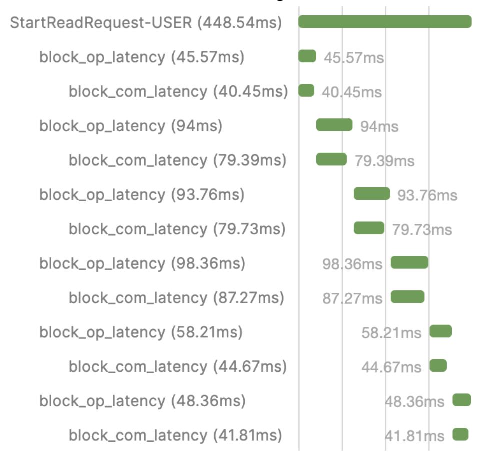


Fig. 11. READ Operation - algorithm: CoAresECF, S:11, k:10, W:5, R:5, fsize:4MB, Debug Level:USER

the object used is 4 MB. The maximum, minimum and average block sizes were set to 1 MB, 512 kB and 512 kB respectively.
Results: In DSMM (Figs. 8, 9), we observe reduced communication latency at $k = 10$. A higher k reduces read/write latency, while a lower k increases redundancy and fault tolerance at the cost of performance.
At $k = 1$ (Fig. 8), encoding latency ($EncodeLatency$) is significantly higher than decode latency due to more parity fragments (m) with lower k, increasing redundancy. Conversely, at $k = 10$ (Fig. 11) with $m = 1$, encoding and decoding are faster with simpler calculations.

4.3.5 Longevity. These scenarios examine the performance and verify the correctness of Ares when reconfigurations coexist with read/write operations. The reconfigurers changes the DAP alternatively and choose servers randomly between [3, 5, 7, 9, 11] servers. The parity value of the EC algorithm is set to $m = 1$ for $|S| = 3$, $m = 2$ for $|S| = 5$, $m = 3$ for $|S| = 7$, $m = 4$ for $|S| = 9$ and

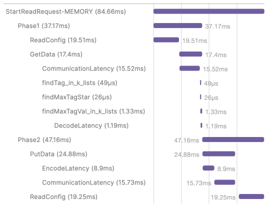


Fig. 12. READ Operation - Algorithm:CoAresF, S:11, W:5, R:15, G=1, fsize:4MB, Debug Level:DSMM

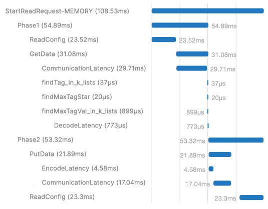


Fig. 13. READ Operation - Algorithm:CoAresF, S:11, W:5, R:15, G=5, fsize:4MB, Debug Level:DSMM

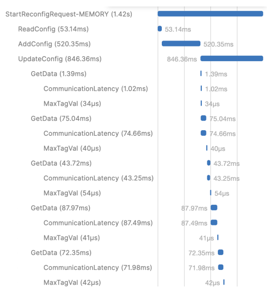


Fig. 14. RECON Operation - Algorithm:CoAresF, S:11, W:5, R:15, G=5, fsize:4MB, Debug Level:DSMM

$m = 5$ for $|S| = 11$. We varied the number of reconfigurers from 1 to 5. The numbers of writers and readers are fixed to 5 and 15 respectively. The size of the file used is 4 MB. We used the external implementation of Raft consensus algorithms, which was used for the service reconfiguration and was deployed on 6 physical nodes on Emulab.
Results: In Figs. 12-14, single read/write/recon operations may access various configurations, including both ABD and EC algorithms when concurrent with recon operations. Reconfiguration introduces higher delays due to more communication rounds. As the number of reconfigurers increases, all latencies rise, as evident in Figs. 12-13. In CoAresF, recon operations involve multiple $GetData$ and $PutData$ operations, optimizing reconfiguration latency by performing it on the DSMM level for the entire file, gathering and transferring blocks from previous configurations to the new one during a single reconfig.

4.3.6 Results Summary. A general observation from the tracing evaluation is that configuration discovery adds a stable overhead to the operations, as the client traverses the entire configuration sequence. A representative sample of our results appears in Fig. 2 showing the time it takes for two read operations to complete on the shared memory (one block in CoAresECF and the whole object in AresEC). The tracing mechanism is able to illustrate the various phases and actions in the two operations, and lay down the time it takes for their completion. This allows us to pinpoint potential bottlenecks of the two algorithms. From Fig. 2 we observe that the communication and computation latencies vary, while the latency of the read-config operation incurs a stable overhead regardless of the object size. It is worth noting that in this experiment the configuration remains unchanged, and thus time spent for configuration discovery is unnecessary. Due to read-config, fragmented algorithms also suffer a stable overhead for each block of the file. This overhead increases when handling large objects split into many blocks, ultimately defeating the purpose of fragmentation. Lastly, experiments involving concurrent recon operations (cf. Appendix 4.2), demonstrated increased latencies as operations have to access/traverse multiple configurations. This led to the conclusion that latency also suffers as the number of reconfigurations (and thus the number of concurrent configurations) increases.

5 FROM ARES TO ARES II

In this section, we introduce a new algorithm called Ares II, which enhances its predecessor, Ares, by addressing identified shortcomings found on Section 4, and mainly affecting the reconfiguration mechanism of the original algorithm. We present the following enhancements: (i) to expedite configuration discovery we introduce piggy-back data on read/write messages (Section 5.1), (ii) for service longevity and to expedite configuration discovery, we introduce a garbage collection mechanism that removes obsolete configurations and updates older configurations with newly established ones (Section 5.2), and (iii) to expedite reconfiguration we introduce a batching mechanism where a single configuration is applied not on a single but multiple objects concurrently (Section 5.3). Below we describe the modifications required for Ares, EC-DAP, and the reconfiguration protocol to support the above enhancements, resulting in Ares II and EC-DAP II. We apply similar changes to ABD-DAP and implement ABD-DAP II, which are evaluated in Section 7. In the pseudocode of the algorithms that follow, struck-out text annotates code that has been removed compared to Ares. The colored text annotates the changed code, and the colored box annotates newly added code. Each optimization has a different color: (i) **red**, (ii) **blue**, (iii) **green**.

5.1 Optimization 1: Configuration Piggyback

As explained in Section 3, during the read and write operations in Ares, a node learns about new configurations performing read-config actions. However, as seen in Section 4, the latency of a read-config operation produces a stable communication latency overhead, regardless the size of the object.

Description. In this optimization, we combine data retrieval with configuration discovery in an attempt to avoid the time spent during read-config. In particular, in Ares II, a server piggybacks the *nextC* variable on every get-data/put-data reply, allowing the client to both obtain/update the object value and discover any new configuration through a single message exchange. This optimization brings changes in the specification of read/write operations, the DAP implementations, and the *sequence traversal* (cf. Section 3).

Read/Write operations. Algorithm 1 specifies the read and write protocols of Ares II.
State variable *cseq* at each reader, writer, and reconfigurer process is used to hold a sequence of configurations as these are discovered at each client. We use the convention that the $cseq[i]$ of a client p stores a tuple $\langle c_i, *\rangle$ if p discovered the configuration with identifier i; otherwise we say that $cseq[i] = \perp$. Initially, *cseq* contains a single element, $\langle c_0, F\rangle$, which is an initial configuration known to every participant in the service. The local variable $Cs \subseteq C \times \{P, F\}$ is a set used by readers and writers to collect the *nextC* values received by servers, during any DAP action.
In contrast to Ares, both write and read operations, do not issue a read-config action to obtain the latest introduced configuration (cf. line Alg. 1:8 (resp. line Alg. 1:32). Instead, the reader/writer finds the last finalized entry in its local *cseq* denoted as μ and starts the read/write operation from $cs = cseq[\mu]$ (line Alg. 1:10 (res[. line] Alg. 1:34)).
Starting from $cs = cseq[\mu]$, a read/write operation then invokes get-data/get-tag operations respectively. In the case of a read operation, the reader collects the $\langle \tau_c, v_c\rangle$ pairs and discovers the maximum $\langle \tau_{max}, v_{max}\rangle$ pair among them, where $v_c \neq \perp$. Similarly in the case of a write operation, the writer collects only the tag values τ_c and discovers the maximum τ_{max} among them. At the same time, both operations, collect the set of configurations Cs, and discover using the $\text{find-next-config}(\text{cseq}, \text{Cs})$ action the next available configuration cs.
The process above is repeated until the client reaches $cs = \perp$ which indicates that all servers in a quorum responded with $nextC.cfg = \perp$ (lines Alg. 1:11–15 (resp. lines Alg. 1:35–39)). Once done, the writer associates a new tag with the value to be written, as done in Ares (cf. line Alg.1:16).
Then both read/write operations propagate the tag-value pairs to the servers. The propagation of $\langle \tau, v\rangle$ in write (lines Alg. 1:19–23) and $\langle \tau_{max}, v_{max}\rangle$ in read (lines Alg. 1:42–46) follows the same logic as in the first phase, involving this time the put-data actions.
It is worth noting that in Ares, the discovery of new configuration and the retrieval of the tags/pairs occurs in two separate communication rounds: first executing read-config to update the *cseq*, and then by get-tag or get-data to retrieve the tag/value pairs. Same goes for the propagation phase.

Algorithm 1 Write and Read protocols at the clients for Ares II.

```
    Write Operation:
2:    at each writer $w_i$
      State Variables:
4:    $cseq[\,]\ s.t.\ cseq[j] \in C \times \{F,P\}$
      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:     $cs \leftarrow cseq[\mu]$
        while $cs \neq \bot$ do
12:       $\tau_c, Cs \leftarrow cs.cfg.$get-tag()
          $\tau_{max} \leftarrow \max(\tau_c, \tau_{max})$
14:       $cs, cseq \leftarrow$find-next-config$(cseq, Cs)$
        end while
16:     $\langle \tau, v\rangle \leftarrow \langle\langle \tau_{max}.ts + 1, \omega_i\rangle, val\rangle$

        $\lambda \leftarrow \max(\{i : cseq[i] \neq \bot\})$
18:     $cs \leftarrow cseq[\lambda]$
        while $cs \neq \bot$ do
20:       $Cs \leftarrow cs.cfg.$put-data$(\langle\tau, v\rangle)$
          ~~$cseq \leftarrow$ read-config($cseq$)~~
22:       $cs, cseq \leftarrow$find-next-config$(cseq, Cs)$
        end while
24:   end operation

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

      operation read( )
32:     ~~$cseq \leftarrow$ read-config($cseq$)~~
        $\mu \leftarrow \max(\{j : cseq[j].status = F\})$
34:     $cs \leftarrow cseq[\mu]$
        while $cs \neq \bot$ do
36:       $\langle \tau_c, v_c\rangle, Cs \leftarrow cs.cfg.$get-data()
          $\langle \tau_{max}, v_{max}\rangle \leftarrow \max(\langle \tau_c, v_c\rangle, \langle \tau_{max}, v_{max}\rangle)$
            $\hookrightarrow$ s.t. $v_c \neq \bot$
38:       $cs, cseq \leftarrow$find-next-config$(cseq, Cs)$
        end while

40:     $\lambda \leftarrow \max(\{i : cseq[i] \neq \bot\})$
        $cs \leftarrow cseq[\lambda]$
42:     while $cs \neq \bot$ do
          $Cs \leftarrow cs.cfg.$put-data$(\langle \tau_{max}, v_{max}\rangle)$
44:       ~~$cseq \leftarrow$ read-config($cseq$)~~
          $cs, cseq \leftarrow$find-next-config$(cseq, Cs)$
46:     end while
        return $v_{max}$
48:   end operation
```

Sequence Traversal. sequence traversal II is presented in Algorithm 2.
find-next-config$(cseq, Cs)$*:* Given a batch of servers' replies Cs containing their $nextC$ values, it determines whether any of the replies contains $nextC.cfg \neq \bot$ (line Alg. 2:13). In cases where there are replies with $nextC.status = F$, we need to identify the $nextC$ with the highest ID (line Alg. 2:6), as servers may point to different next finalized configurations due to garbage collection in the reconfig operation (cf. Section 5.2). After identifying the $nextC$, the put-config action notifies a quorum of servers in $c.Servers$, where c the current configuration, to update the value of their $nextC$ variable to the found one (lines Alg. 2:15). Finally, the action returns the updated $cseq$ along with the found $nextC$. read-config$(cseq)$*:* The procedure remains the same as that in Ares, with modifications aimed at avoiding code duplication.
EC-DAP II Implementation. The piggyback optimization needed also to be supported by the DAP protocols. Here we present only the modifications done in the EC-DAP protocol. Similar modifications are applied to the ABD-DAP. EC-DAP II is presented in Algorithms 3 and 4.
Following [28], each server s_i stores a state variable, $List$, which is a set of up to $(\delta + 1)$ (tag, coded-element) pairs; δ is the maximum number of concurrent put-data operations. Also in any configuration c that implements EC-DAP II, we assume an $[n, k]$ MDS code [24], where $|c.Servers| = n$ of which no more than $\frac{n-k}{2}$ may crash. The $[n, k]$ MDS encoder slits the a value v into k equal fragments, which then uses to generate n coded elements $e_1, e_2, \ldots, e_n$, denoted as $\Phi(v)$. We denote the projection of Φ onto the ith output component as Φ_i, where $e_i = \Phi_i(v)$. We associate each coded

Algorithm 2 Sequence traversal at each process $p \in \mathcal{I}$ of Ares II.

```
procedure find-next-config($cseq, Cs$)
  $\lambda \leftarrow \max(\{j : cseq[j] \neq \bot\})$
  $lastC \leftarrow cseq[\lambda]$
  $Cs_F \leftarrow \{cs : cs \in Cs \ \wedge cs.status = F\}$
  if $Cs_F \neq \emptyset$ then
    $\boxed{next\lambda \leftarrow \max(cs.cfg.ID : cs \in Cs_F)}$
    $nextC \leftarrow cs \in Cs_F$ s.t. $cs.cfg.ID = next\lambda$
  else if $\exists cs \in Cs$ s.t. $cs \neq \bot \wedge cs.status = P$ then
    $next\lambda \leftarrow cs.cfg.ID$
    $nextC \leftarrow cs$
  else
    $nextC \leftarrow \bot$
  if $nextC \neq \bot$ then
    $cseq[next\lambda] \leftarrow nextC$
    put-config($lastC.cfg, cseq[next\lambda]$)
  return $cseq, nextC$
end procedure

procedure read-config($cseq$)
  $\mu \leftarrow \max(\{j : cseq[j].status = F\})$
  $cs \leftarrow cseq[\mu]$
  while $cs \neq \bot$ do
    $Cs \leftarrow$get-next-config($cs.cfg$)
    $cseq, cs \leftarrow$ find-next-config(cseq, Cs)
  end while
  return $cseq$
end procedure

procedure get-next-config($c$)
  send (READ-CONFIG) to each $s \in c.Servers$
  until $\exists Q, Q \in c.Quorums$ s.t. $rec_i$ receives $nextC_s$
      $\hookrightarrow$ from $\forall s \in Q$
  $Cs \leftarrow \{nextC_s :$ received $nextC_s$ from each $s \in Q\}$
  return $Cs$
end procedure

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

procedure gc-config($cseq$)
  $\mu \leftarrow \max(\{i : cseq[i].status = F\})$
  $CID \leftarrow \{i : cseq[i] \neq \bot \wedge i < \mu\}$
  for $id$ in $CID$ do
    send (GC-CONFIG, $next$) to each $s \in cseq[id].Servers$
    until $\exists Q, Q \in cseq[id].Quorums$ s.t. $rec_i$ receives ACK
    $\hookrightarrow$ from $\forall s \in Q$
    /* remove the {id, cseq[id]} */
    $cseq \leftarrow cseq \setminus \{id, cseq[id]\}$
  return $cseq$
end procedure
```

element e_i with server i, $1 \leq i \leq n$. Such erasure code algorithm enables recovery of a coded value v using any k out of the n coded elements, each representing a fraction $\frac{1}{k}$ of v.
We now proceed with the description of the three primitives of EC-DAP II.
Primitive c.get-tag()*:* The difference from the EC-DAP is that the client, in addition to requesting the servers' maximum τ, also queries their $nextC$ within the same request. So, it returns both a list with servers' τ values and the corresponding Cs values.
Primitive c.get-data()*:* Similarly with the get-tag action, the client, in addition to requesting the servers' $List$, also queries their $nextC$ within the same request. If the status of $nextC$ is F, the server creates a $List'$ which contains all the tags associated with the $\bot$ value, i.e., $\langle\tau, \bot\rangle$ (lines Alg. 4:11–12). As by the reconfiguration algorithm, a finalized configuration contains the latest data, we achieve communication efficiency by avoiding propagating the data from servers that point to a next finalized configuration, i.e. the status of their $nextC$ is finalized. In this case, the server responds with $List'$ and $nextC$. Otherwise, the server returns $List$ and $nextC$.
It is possible that the $List$ contains $\bot$ values, as a result of a garbage collection (GC) of reconfig operation (cf. Section 5.2). So, the client needs to check whether all fragments associated with $t^{\text{dec}}_{\max}$ (line Alg. 3:16) are not equal to $\bot$. If $\bot$ does not exist in fragments (Alg. 3:17), the client can decode v Otherwise, it sets the value to $\bot$.
Primitive c.put-data($\langle\tau, v\rangle$)*:* In line with EC-DAP, the client computes coded elements and dispatches the pair $(\tau, \Phi_i(v))$ to the relevant servers (s_i). When server s_i receive a (PUT-DATA, τ, e_i) message, s_i verifies if its $List$ does not include $e_i = \bot$ (line Alg. 4:18). If it is, it indicates that the configuration

Algorithm 3 EC-DAP II implementation

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

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

procedure c.get-data()
    send (QUERY-LIST) to each $s \in c.Servers$
    until $p_i$ receives $List_s, nextC_s$ from $\lceil \frac{n+k}{2} \rceil$ servers in $c.Servers$
    $\boxed{Cs \leftarrow \{nextC_s : \text{ received } nextC_s \text{ from } s\}}$
    $Tags_{dec}^{\geq k}$ = set of tags that appears in $k$ Lists
    $t_{max}^{dec} \leftarrow \max(Tags_{dec}^{\geq k})$
    if $Tags_{dec}^{\geq k} \neq \emptyset$ then
        $\boxed{fragments \leftarrow \{e : \langle \tau, e \rangle \in Lists \;\&\; \tau = t_{max}^{dec}\}}$
        if $\boxed{\nexists \bot \in fragments}$ then
            $v \leftarrow$ decode value for $t_{max}^{dec}$
        else
            $\boxed{v \leftarrow \bot}$
        return $\langle t_{max}^{dec}, v \rangle, Cs$
end procedure

procedure c.put-data($\langle \tau, v \rangle$))
    $code\text{-}elems = [(\tau, e_1), \ldots, (\tau, e_n)]$, $e_i = \Phi_i(v)$
    send (PUT-DATA, $\langle \tau, e_i \rangle$) to each $s_i \in c.Servers$
    until $p_i$ receives $nextC_s$ from each server $s \in \mathcal{S}_g$
        $\hookrightarrow$ s.t. $|\mathcal{S}_g| = \lceil \frac{n+k}{2} \rceil$
        and $\mathcal{S}_g \subset c.Servers$
    $\boxed{\begin{array}{l} Cs \leftarrow \{nextC_s \text{: received } nextC_s \text{ from each } s \in \mathcal{S}_g\} \\ \textbf{return } Cs \end{array}}$
end procedure
```

is garbage collected by *GC* operation in reconfig operation (cf. Section 5.2), and s_i does not need to update its *List*. Like in EC-DAP, s_i trims pairs with tags exceeding length $(\delta + 1)$ (line Alg.4:24), but it does not keep older tags of coded-elements with $\bot$ to reduce return message size (line Alg.4:25). Finally, the client returns a *Cs* list comprising all the *nextC* values received from the servers.

5.2 Optimization 2: Garbage Collection

For storage efficiency, longevity, and expediting configuration discovery, we introduce a garbage collection mechanism to eventually remove obsolete configurations. The main idea of this mechanism is to update older configurations to point to more recently established configurations. At the same time we save storage by removing the obsolete content of older configurations. This optimization brings changes in the specification of recon operations.
Description. The Garbage Collection (gc-config) runs in the end of the reconfiguration operation (line Alg. 5:12). The main idea is that the entries that appear in *cseq* before the last finalized entry can be garbage collected and point to the last finalized entry in *cseq*. Thus, the gc-config sends request to servers for each configuration *cs* in *cseq* that has smaller configuration id $cs.cfg.ID$ than the last finalized index μ, followed by removing the garbage collected configuration from its local *cseq* (lines Alg. 2:40–44). We pass $cseq[\mu]$ as a parameter to the server message. When a server receives the message, it checks if the index of received configuration is larger than the index of its local *nextC* (line Alg. 4:37). If that holds, the server updates the *nextC* to point to the last finalized configuration (line Alg. 4:38). Next, the server sets the values e of all $\langle \tau, e \rangle$ in *List* to $\bot$ (lines Alg. 4:39–41).

5.3 Optimization 3: Reconfiguration Batching

A reconfiguration operation in ARES is applied on a single atomic object. Thus, in systems where we need to manipulate multiple objects, e.g., the blocks of a fragmented object, whenever a reconfigurer wants to move the system from a configuration c to a configuration c' needs to execute a series of

Algorithm 4 The response protocols at any server $s_i \in \mathcal{S}$ of ARES II

```
    at each server $s_i \in \mathcal{S}$ in configuration $c_k$
2:  State Variables:
    $List \subseteq \mathcal{T} \times C_s$, initially $\{(t_0, \Phi_i(v_0))\}$
    $nextC \in C \times \{P, F\}$, initially $\langle \bot, P \rangle$

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

8:  Upon receive (QUERY-LIST) $_{s_i, c_k}$ from $q$
10:     if $nextC.status = F$ then
            for $\tau, v$ in $List$ do
12:             $List' \leftarrow List' \cup \{\langle \tau, \bot \rangle\}$
            Send $List', nextC$ to $q$
14:     Send $List$, $nextC$ to $q$
    end receive

16: Upon receive (PUT-DATA, $\langle \tau, e_i \rangle$) $_{s_i, c_k}$ from $q$
        $fragments \leftarrow \{e : \langle t, e \rangle \in List\}$
18:     if $\nexists \bot \in fragments$ then
            $List \leftarrow List \cup \{\langle \tau, e_i \rangle\}$
20:         if $|List| > \delta + 1$ then
                $\tau_{min} \leftarrow \min\{t : \langle t, * \rangle \in List\}$
22:             /* remove the coded value */
                $List \leftarrow List \backslash \{\langle \tau, e \rangle : \tau = \tau_{min} \wedge \langle \tau, e \rangle$
24:                 $\in List\}$
                ~~$List \leftarrow List \cup \{(\tau_{min}, \bot)\}$~~
26:     Send $nextC$ to $q$
    end receive

28: Upon receive (READ-CONFIG) $_{s_i, c_k}$ from $q$
        send $nextC$ to $q$
30: end receive

    Upon receive (WRITE-CONFIG, $cfgT_{in}$) $_{s_i, c_k}$ from $q$
32:     if $nextC.cfg = \bot \vee nextC.status = P \vee$
            $\hookrightarrow cfgT_{in}.cfg.ID > nextC.cfg.ID$ then
            $nextC \leftarrow cfgT_{in}$
34:     send ACK to $q$
    end receive
36: Upon receive (GC-CONFIG, $cfgT_{in}$) $_{s_i, c_k}$ from $q$
        if $cfgT_{in}.cfg.ID > nextC.cfg.ID$ then
38:         $nextC \leftarrow cfgT_{in}$
        for $\tau, e$ in $List$ do
40:         $List \leftarrow List \backslash \{\langle \tau, e \rangle\}$
            $List \leftarrow List \cup \{(\tau, \bot)\}$
42:     send ACK to $q$
    end receive
```

recon operations, one for each object. Running multiple recons however, means executing a different instance of consensus and maintaining different configuration sequence per object, resulting in significant overhead and performance degradation. In this section we examine an optimization that suggests the application of the reconfiguration operation over a domain of (one or all) objects. In the implementation below, instead of having an instance of consensus for each configuration (as defined in Section 3), we use an external consensus mechanism that maintains the values from C.

Description. We implement reconfiguration batching in ARES II by changing the specification of the reconfiguration protocol (Alg. 5), and in particular the update-config procedure. The latter procedure is responsible for moving the latest value of an object from an older configuration to the one a reconfigurer tries to install. While ARES was executing this procedure on a single object, ARES II executes the procedure on a given *domain* (or set) of one or more objects.

More precisely, assume that a reconfigurer rec_i wants to change the current configuration c to c' by invoking a reconfig(c', D). Then, rec_i collects the sequence $cseq$ of established configurations using the read-config procedure (Alg. 5:8), and then attempts to add the configuration c' at the end of $cseq$ using the consensus service (Alg. 5:9). Once the newly added configuration is established, given a domain D, rec_i invokes the update-config procedure. Using the get-data DAP, rec_i, gathers the tag-value pairs for each object o in D, from every configuration $cseq[i]$, for $\mu \leq i \leq \lambda$ (i.e., the last finalized configuration with index μ and the configuration with the largest index λ in $cseq$). Then it discovers the maximum of those pairs and transfers, using the put-data DAP, the pair for each object in D to the new configuration c'. For example, if $\langle t_{max}, v_{max} \rangle$ is the tag value

Algorithm 5 Reconfiguration protocol of Ares II.

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

6:  operation reconfig(c, D)
        if $c \neq \bot$ then
8:          $cseq \leftarrow$read-config$(cseq)$
            $cseq \leftarrow$ add-config$(cseq, c)$
10:         update-config$(cseq, D)$
            $cseq \leftarrow$ finalize-config$(cseq)$
12:         $\boxed{cseq \leftarrow \text{gc-config}(cseq)}$
    end operation
14: procedure add-config(cseq, c)
        $\lambda \leftarrow \max(\{j : cseq[j] \neq \bot\})$
16:     $c' \leftarrow cseq[\lambda].cfg$
        $d \leftarrow Con.propose(\lambda + 1, c)$
18:     $d.ID \leftarrow \lambda + 1$
        $cseq[\lambda + 1] \leftarrow \langle d, P\rangle$
20:     put-config$(c', \langle d, P\rangle)$
        return $cseq$
22: end procedure

    procedure update-config(cseq , D)
24:     $\mu \leftarrow \max(\{j : cseq[j].status = F\})$
        $\lambda \leftarrow \max(\{j : cseq[j] \neq \bot\})$
        M = [ ]
26:     for $o$ in $D$ do
28:         $M[o] \leftarrow \emptyset$
        for $i = \mu : \lambda$ do
30:         for $o$ in $D$ do
                $\langle t, v\rangle, \_ \leftarrow cseq[i].cfg$.get-data() for object $o$
32:             $\boxed{M[o] \leftarrow M[o] \cup \{\langle \tau, v\rangle\}}$
        for $o$ in $D$ do
34:         $\langle \tau, v\rangle \leftarrow \max_t\{\langle t, v\rangle : \langle t, v\rangle \in M[o]\}$
            $cseq[\lambda].cfg$.put-data$(\langle \tau, v\rangle)$ for object $o$
36: end procedure

    procedure finalize-config(cseq)
38:     $\lambda \leftarrow \max(\{j : cseq[j] \neq \bot\})$
        $cseq[\lambda].status \leftarrow F$
40:     put-config$(cseq[\lambda - 1].cfg, cseq[\lambda])$
        return $cseq$
42: end procedure
```

pair corresponding to the highest tag among the responses from all the $\lambda - \mu + 1$ configurations for a specific object o, then $\langle t_{max}, v_{max}\rangle$ is written to the configuration c' via the invocation of $cseq[\lambda].cfg$.put-data$(\langle \tau_{max}, v_{max}\rangle)$ of object o.

6 CORRECTNESS OF ARES II

6.1 Correctness of EC-DAP II

As seen in Section 3, Ares relies on three *data access primitives* (DAPs): (i) the get-tag, (ii) the get-data, and (iii) the put-data$(\langle \tau, v\rangle)$. For the DAPs to be useful, they need to satisfy a property, referred in [28] as *Property 1*. We slightly revised *Property 1* to accommodate the fact that get-data can return a tag associated with either a value from $\mathcal{V}$ or $\bot$. See the revised *Property 1*.

Property 1. *In an execution ξ we say that a DAP operation in 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$(\langle \tau_\phi, v_\phi\rangle)$, *$\langle \tau_\phi, v_\phi\rangle \in \mathcal{T} \times \mathcal{V}$, and π is c.*get-tag() *(or c.*get-data()*) that returns $\tau_\pi \in \mathcal{T}$ (or $\langle \tau_\pi, v_\pi\rangle \in \mathcal{T} \times \mathcal{V} \cup \{\bot\}$) and $\phi \to \pi$ in ξ, then $\tau_\pi \geq \tau_\phi$.*
- C2 *If ϕ is a c.*get-data() *that returns $\langle \tau_\pi, v_\pi\rangle \in \mathcal{T} \times \mathcal{V} \cup \{\bot\}$, then there exists π such that π is a c.*put-data$(\langle \tau_\pi, v_\pi\rangle)$ *and ϕ did not complete before the invocation of π. If no such π exists in ξ, then (τ_π, v_π) is equal to (t_0, v_0).*

Given that Ares II use DAP operations that satisfy the conditions in Property 1, this allows us to show in Section 6.2 that Ares II satisfies atomicity. In our implementation of Ares II we use two DAP algorithms: (i) EC-DAP II (see Section 5.1), and (ii) ABD-DAP II. The main differences of the enhanced algorithms compared to their original counterparts that appeared in [28], are the

following: (i) each of their read-data operation may return a $\perp$ value, and (ii) each DAP operation includes in its response a list *Cs* of the *nextC* values received from the server's replies (piggyback). As Property 1 focuses on the tags and not the values returned, then the proof that ABD-DAP II satisfies Property 1 is almost identical to the proof for ABD-DAP in [28]; so we refer the reader to [28] for that proof. The fact that DAP operations may return $\perp$ however, may affect decodability of the value (and thus termination of operations) in EC-DAP II. Hence in the rest of the section we focus in proving that EC-DAP II satisfies Property 1 as well. In particular, to prove the correctness of EC-DAP II, we need to show that it is *safe*, i.e., it ensures Property 1, and *live*, i.e., it allows each operation to terminate.
For the following proofs we fix the configuration to c as it suffices that the DAPs preserve Property 1 in any single configuration. Also we assume an $[n, k]$ MDS code [24], $|c.Servers| = n$ of which no more than $\frac{n-k}{2}$ may crash. We refer to δ as the maximum number of put-data operations concurrent with any get-data operation.

Lemma 1 (C2). *Let ξ be an execution of an algorithm A that uses the* EC-*DAP II. If ϕ is a c.*get-tag() *that returns $\tau_\pi \in \mathcal{T}$ or a c.*get-data() *that returns $\langle \tau_\pi, v_\pi \rangle \in \mathcal{T} \times \mathcal{V} \cup \perp$, then there exists π such that π is a c.*put-data($\langle \tau_\pi, v_\pi \rangle$) *and ϕ did not complete before the invocation of π. If no such π exists in ξ, then (τ_π, v_π) is equal to (t_0, v_0) or $(t_0, \perp)$.*

Proof. The proof of property $C2$ of EC-DAP II when $\pi = c$.get-tag() or $\pi = c$.get-data() and the return value is not $\perp$ is identical to that of EC-DAP (Theorem 2 in [28]). This similarity arises because the initial value of the *List* variable in each server s in $\mathcal{S}$ remains $(t_0, \Phi_s(v_0))$, and new tags are added to the *List* exclusively through put-data operations. However, get-data can return a $\perp$ value in two cases: (i) the value of elements in the *List* can be set to $\perp$ via a subsequent gc-config operation, or (ii) when the *nextC.status* of a server is F, the server creates a new *List'* during the get-data operation, which contains all the tags from *List* associated with the $\perp$ value (lines Alg. 4:11–12). As a result, during a get-data operation, each server includes all the tags from the *List* as before, but now some tags may now have associated $\perp$ value. If ϕ returns $(t_\phi, \perp)$, this means that at least one server returns $\perp$ with $t_\phi = t_0$ or $t_\phi > t_0$. If $t_\phi = t_0$, there is nothing to prove. If $t_\phi > t_0$, there is a put-data(t_π, v_π) operation π. The value v_π of the pair $\langle t_\pi, v_\pi \rangle$ later becomes $\perp$ for any of the two reasons mentioned before. To show ϕ cannot complete before π for any π, we argue by contradiction: if ϕ completes before π begins for every π, t_π cannot be returned by ϕ, contradicting the assumption. □

Lemma 2 (C1). *Let ξ be an execution of an algorithm A that uses the* EC-*DAP II. If ϕ is c.*put-data($\langle \tau_\phi, v_\phi \rangle$), *for $c \in C$, $\langle \tau_\phi, v_\phi \rangle \in \mathcal{T} \times \mathcal{V}$, and π is c.*get-data() *that returns $\langle \tau_\pi, v_\pi \rangle \in \mathcal{T} \times \mathcal{V} \cup \{\perp\}$ or π is c.*get-tag() *that returns $\tau_\pi \in \mathcal{T}$ and $\phi \to \pi$ in ξ, then $\tau_\pi \geq \tau_\phi$.*

Proof. Let p_ϕ and p_π denote the processes that invoke ϕ and π in ξ. Let $S_\phi \subset \mathcal{S}$ denote the set of $\lceil \frac{n+k}{2} \rceil$ servers that respond to p_ϕ, during ϕ, and by S_π the set of $\lceil \frac{n+k}{2} \rceil$ servers that respond to p_π, during π. Per Alg. 4:19, every server $s \in S_\phi$, inserts the tag-value pair received by p_ϕ in its local *List*. Note that once a tag-value pair is added to *List*, is removed only when the *List* exceeds the length $(\delta + 1)$ and the tag of the pair is the smallest in the *List* (Alg. 4:20–24). Except that the server may remove the tag-value pair from the *List*, it can also set the value of the pair to $\perp$ and retain the tag. Notice that as $|S_\phi| = |S_\pi| = \lceil \frac{n+k}{2} \rceil$, then $|S_\phi \cap S_\pi| \geq k$ reply to both π and ϕ. There are two cases to examine: (a) the pair $\langle \tau_\phi, * \rangle \in List$ of at least k servers in S_π, and (b) the $\langle \tau_\phi, * \rangle$ appeared in fewer than k servers in S_π.

Case a: In this case τ_ϕ was discovered in at least k servers in S_π. This happens since there are not enough concurrent write operations to remove the elements corresponding to tag τ_ϕ. In the case of π being a get-tag operation, the only difference is the inclusion of $nextC$ in every server reply and their aggregation in a set Cs at the client, then with similar reasoning as in Lemma 19 in [28], we can show that the lemma holds for the get-tag operation. In the case of π being a get-data operation, we can break this case in two subcases: (i) no server $s \in S_\pi$ returns $\perp$ associated τ_ϕ and (ii) at least one server $s \in S_\pi$ returns $\perp$ associated τ_ϕ. In case (i), π discovers τ_ϕ in at least k servers, ensuring that the value associated with τ_ϕ will be decodable, as there are no $\perp$ values associated with τ_ϕ (line Alg. 3:18). Hence, $t^{dec}_{max} \geq \tau_\phi$ and $\tau_\pi \geq \tau_\phi$. In case (ii), since at least one server $s \in S_\pi$ return $\perp$ value, it indicates that either a gc-config operation has occurred or $nextC = F$. In the first subcase, servers executing gc-config set $List$ values to $\perp$, while retaining tags. In the latter subcase, servers in S_π send all the $List$ with the tags associated with $\perp$ values (Alg.4:11-12). In both cases (possibly both applying), τ_ϕ was discovered by k servers in S_π, but there is at least one $\perp$ in their associated elements. Thus, according to Alg.3:20, the value is not decodable, and the client returns $\perp$ during get-data. Per Alg.3:13, t^{dec}_{max} is defined as the set of tags that appear in k Lists with values from the set $\mathcal{V} \cup \perp$. Therefore, in this case, $t^{dec}max \geq \tau_\phi$, and thus $\tau_\pi > \tau_\phi$.

Case b: In this case τ_ϕ was discovered in less than k servers in S_π. A server $s \in S_\phi \cap S_\pi$ will not include τ_ϕ iff $|Lists_s| = \delta + 1$, and therefore the local $List$ of s removed τ_ϕ as the smallest tag in the list. According to our assumption though, no more than δ put-data operations may be concurrent with a get-data operation. Thus, at least one of the put-data operations that wrote a tag $\tau' \in Lists_s$ must have completed before π. Since τ' is also written in $|S'| = \frac{n+k}{2}$ servers then $|S_\pi \cap S'| \geq k$. Thus, π will be able to find the τ'; whereas, when $\pi =$ get-data, π will be able to decode the value associated with τ' or return $\perp$. Hence $t^{dec}_{max} \geq \tau'$ and $\tau_\pi \geq \tau_\phi$, completing the proof of this lemma. □

Theorem 3 (Safety). *Let ξ be an execution of an algorithm A that contains a set Π of complete* get-tag/get-data *and* put-data *operations of Algorithm 3. Then every pair of operations $\phi, \pi \in \Pi$ satisfy Property 1.*

Proof. Follows directly from Lemmas 2 and 1. □

Theorem 4 (Liveness). *Let ξ be an execution of an algorithm A that utilises the* EC-*DAP II. Then any* get-tag, get-data *and* put-data *π invoked in ξ will eventually terminate.*

Proof. The only difference between the get-tag and put-data operations of EC-DAP II compared to those of EC-DAP is that they piggyback a list Cs with the servers' $nextC$ in their replies. Thus, our primary focus now shifts to ensuring the completion of the get-data operation, as its decodability is affected. Let p_π be the process executing a get-data operation π. Define S_π as the set of $\lceil \frac{n+k}{2} \rceil$ servers responding to p_π. Let T_1 denote the earliest time point when p_π receives all the $\lceil \frac{n+k}{2} \rceil$ responses. Additionally, let Λ encompass all put-data operations starting before T_1. Observe that, by algorithm design, the coded-elements corresponding 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. According to our assumption though, no more than δ put-data operations may be concurrent with a get-data operation. Since $List$ variable has length $\delta + 1$, at least one of the put-data operations that wrote a tag $\tau' \in Lists_s$ must have completed before T_1. Since τ' is written in $|S'| = \frac{n+k}{2}$ servers then $|S_\pi \cap S'| \geq k$ and hence π will be able to find the τ' in k Lists. Although the get-data can find the t^{dec}_{max} (the maximum tag π discovered in k Lists), there are two subcases to consider for its accosiated values: (a) π does not receive $\perp$ elements in its replies from any server $s \in S_\pi$ and (b) there is at least one server $s \in S_\pi$ that returns $\perp$ value. In case (a), since π does not receive $\perp$ elements, each server $s \in S_\pi$ includes t^{dec}_{max} associated with non-empty coded elements in its replies.

Consequently, t^{dec}_{max} must be in $Tag^{\geq k}_{dec}$, and its value is decodable. In case (b), at least one $\perp$ value associated with t^{dec}_{max} received from any $s \in S_\pi$ during the operation π. As seen above, a server can return $\perp$ in two cases: (i) if the $nextC.status$ of a server is F, or (ii) the value in its $List$ set to $\perp$ via a subsequent gc-config operation. While in case (i) it is clear that the π can detect the $\perp$ value from any server $s \in S_\pi$, we should prove it for the case (ii). Let p_γ be the process that invokes the gc-config operation γ . Define S_γ as the set of $\lceil \frac{n+k}{2} \rceil$ servers responding to p_γ in the gc-config operation during γ. Notably, at execution point T_0 in ξ, just before the completion of π, the garbage collection operation γ is initiated for the configuration consisting of the set of servers in S_γ. As per the algorithm design, the elements corresponding to the configuration of S_γ set the values in their $List$ to $\perp$ during garbage collection (lines Alg. 4:39–41). Hence, between execution points T_0 and T_1 in ξ, the elements of every active server $s \in S_\gamma$ may undergo garbage collection. Since $|S_\gamma| = |S_\pi| = \lceil \frac{n+k}{2} \rceil$ and $|S_\gamma \cap S_\pi| \geq k$, a $\perp$ value can be detected by p_π from at least one server. Since get-data detects at least one $\perp$ element, it will not attempt to decode the value. Instead, it returns a $\perp$ value (line Alg. 3:20). □

6.2 Correctness of Ares II

The correctness of Ares (Section 6 of [28]) highly depends on the way the configuration sequence is constructed at each client process. Also, atomicity is ensured if the DAP implementation in each configuration c_i satisfies Property 1.

This work involves modifications on the reconfiguration aspect of Ares, and in particular it changes the way the configuration sequence is constructed at each client process. In this section we show that the modifications proposed in Ares II do not violate the correctness of the algorithm.

The changes in Ares II have direct implications on the reconfiguration properties of Ares (cf. Section 6.1 of [28]). In Ares, the configuration sequence maintained in two processes is either the same or one is the prefix of the other. In Ares II this changes: the configuration sequence maintained in two processes is either the same or the one is *subsequence* of the other with respect to their indices.

In the following subsections we first present the configuration properties and prove their satisfaction by Ares II, and then we show that given those properties we can prove the correctness of Ares II. We proceed by introducing some definitions and notation we use in the proofs.

$\mathbf{c}^p_\sigma$	the value of the configuration sequence variable $cseq$ at process p in state σ, i.e. a shorthand of $p.cseq\|_\sigma$
$\mathbf{c}^p_\sigma[i]$	the element with the index i in the configuration sequence $\mathbf{c}^p_\sigma$
$\mu(\mathbf{c}^p_\sigma)$	last finalized configuration in $\mathbf{c}^p_\sigma$
$\lambda(\mathbf{c}^p_\sigma)$	the largest index i in the configuration sequence $\mathbf{c}^p_\sigma$, s.t. $\mathbf{c}^p_\sigma[i].cfg \in C$

Table 1. Notations.

Last, we define the notion of subsequence on two configuration sequences.

Definition 5 (Subsequence). A configuration sequence x is a *subsequence* of a sequence y if $\lambda(x) \leq \lambda(y)$. Additionally, for any index j, if $x[j].cfg \in C$ and $y[j].cfg \in C$, then $x[j].cfg = y[j].cfg$.

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

i **configuration uniqueness:** the configuration sequences in any two processes have identical configuration at any common index i,

ii **subsequence:** the configuration sequence observed by an operation is a subsequence of the sequence observed by any subsequent operation, and

iii **sequence progress:** if the configuration with index i is finalized during an operation, then a configuration j, for $j \geq i$, will be finalized in a succeeding operation.

Lemma 6. *For any reconfigurer $r \in \mathcal{G}$ that invokes an* reconfig(c) *operation 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 $Cons.propose(k, c)$ and $\lambda(\mathbf{c}^r_\sigma) = k-1$ where σ the state of r at the completion of* read-config *procedure.*

Proof. This Lemma follows directly from Alg. 5 and the read-config procedure in Alg. 2. Notice that the reconfigurer traverses to a configuration with index say $k - 1$ and then proposes the new configuration to be installed on the next index, i.e. k. □

Lemma 7. *If a server s sets $s.nextC$ to $\langle c, *\rangle$ with index $c.ID = i$ at some state σ in an execution ξ of the algorithm, then $s.nextC$ will set to $\langle c', *\rangle$ with an index $c'.ID \geq i$ for any state σ' that appears after σ in ξ.*

Proof. Notice that a server s updates its $s.nextC$ variable for some specific configuration c_k in a state σ when it receives: (i) a gc-config message or (ii) a write-confing message. In case (i), the $s.nextC.cfg = c$ of c_k has index $c.ID = i$ and the gc-config message received contains a tuple $\langle c', F\rangle$. This gc-config action is initiated by a reconfigurer r which wants to propagate c' before state σ'. At first, r executes read-config and among other configurations it detects the c (i.e., $r.cseq[i]$). By Alg 5:12, r sends a gc-config message to the servers of all configurations in its $r.cseq$ where their index is smaller than $c'.ID$ (including $r.cseq[i]$) in order to propagate to them the proposed $r.cseq[c'.ID]$. By Alg. 4:37, server $s \in c_k$ updates its local $nextC$ only when the received c' has larger index than the local $nextC$. Thus $c'.ID > i$. In case (ii), the write-config message is either the first one received at s for c_k (and thus $s.nextC = \perp$), or $s.nextC = \langle c, *\rangle$ and the message received contains a tuple $\langle c', F\rangle$. There are only two cases where s can modify its $s.nextC$: (i) its status is finalized by a reconfigurer, or (ii) a gc-config updates the $s.nextC$ to c' with index $c'.ID > c.ID$ and a DAP operation propagates c' through a write-config message to a quorum. In case (i), $c = c'$ and the $(.statusc) = P$ hence $c.ID = c'.ID$, while in case (ii) $c'.ID > c.ID$ as shown above and the status of c can be either P or F. □

Lemma 8 (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. $\mathbf{c}^p_{\sigma_1}[i].cfg, \mathbf{c}^q_{\sigma_2}[i].cfg \in C$.*

Proof. The lemma holds trivially for index $i = 0$ such that $\mathbf{c}^p_{\sigma_1}[0].cfg = \mathbf{c}^q_{\sigma_2}[0].cfg = c_0$. So in the rest of the proof we focus in the case where index $i > 0$. 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 c with index i in two cases: (a) either it received c as the result of invoking a propose operation on index i to the consensus external service (Alg. 5: 17), or (b) p receives $s.nextC.cfg = c$ from a server through a DAP operation (Alg. 1:12&20&36&43) or a read-config (Alg. 5:8). Note here that (a) is possible only when p is a reconfigurer and attempts to install a new configuration while its latest discovered configuration is $p.cseq(i-1).cfg$, by Lemma 6. On the other hand (b) may be executed by any process in any operation that reads the $nextC$ of some server $\in p.cseq(k).cfg.Servers$, where

$k \leq i-1$ (i.e., in either get-tag or get-data or put-data or read-config). By Lemma 7, the $nextC$ at each server is monotonic, thus the index of $s.nextC.cfg$ always increases or remains the same. We are going to prove 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 at index $i = 0$. As per Alg. 5, a reconfigurer proposes configurations at a specific index i by sending a request to a consensus external service and receiving the decided configuration c_{i+1} with index $i+1$ in response (line Alg. 5, 17). By Lemma 6, since both processes want to install a configuration in $i = 1$, then they have to run the external consensus service on the index 0. Since $\lambda(\mathbf{c}^p_{\sigma_1}) = \lambda(\mathbf{c}^q_{\sigma_2}) = 0$, both processes have to run the consensus on the same index $i = 0$. Therefore, by the agreement property, they have to decide on the same configuration with the index $i = 1$. Consequently, $c_p = c_q = c_1$ and $\mathbf{c}^p_{\sigma_1}[1].cfg = \mathbf{c}^q_{\sigma_2}[1].cfg = c_1$.
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, either the configuration that has been decided by the consensus instance on index $i = 0$ is propagated to the servers in $c_0.Servers$, or a configuration that is propagated to the $nextC$ of servers in $c_0.Servers$ by a gc-config operation. In the first case, 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 $\mathbf{c}^p_{\sigma_1}[1].cfg$ or $\mathbf{c}^q_{\sigma_2}[1].cfg$ to some received configuration, then $\mathbf{c}^p_{\sigma_1}[1].cfg = \mathbf{c}^q_{\sigma_2}[1].cfg = c_1$ in this case as well. In the second case, the gc-config is executed by a reconfigurer which propagates the finalized proposed configuration c to all the configurations before that. The configuration c can be c_1 or a subsequent finalized configuration. Thus if both p or q receives c_1, it holds that $s.nextC(c_0) = \langle c_1, F \rangle$, and hence $\mathbf{c}^p_{\sigma_1}[1].cfg = \mathbf{c}^q_{\sigma_2}[1].cfg = c_1$. However, if one of them receives a subsequent configuration $\neq c_1$, due to a change of $nextC$ to a configuration with a larger index (Lemma 7), it implies that its $cseq$ sequence at index 1 is denoted as $cseq[1].cfg = \perp \notin C$. This contracted our hypothesis, which assumes that $\mathbf{c}^p_{\sigma_1}[1].cfg, \mathbf{c}^q_{\sigma_2}[1].cfg \in C$.
Hypothesis: We assume that $\mathbf{c}^p_{\sigma_1}[k] = \mathbf{c}^q_{\sigma_2}[k] \neq \perp$ for some k, $k \geq 1$.
Induction Step: We need to show that the lemma holds for some index $i \geq k+1$, where i is the first index after the index k where $\mathbf{c}^p_{\sigma_1}[i] = \mathbf{c}^q_{\sigma_2}[i] \neq \perp$. Let's break down the problem into two subcases: (i) $i = k+1$ and (ii) $i > k+1$. *Case (i):* If both processes retrieve index i with $\mathbf{c}^p_{\sigma_1}[k+1].cfg$ and $\mathbf{c}^q_{\sigma_2}[k+1].cfg$ respectively through consensus, then both p and q run consensus on the previous index k. Since according to our hypothesis the index is k where $\mathbf{c}^p_{\sigma_1}[k] = \mathbf{c}^q_{\sigma_2}[k] \neq \perp$ then both processes will receive the same decided value for index $k+1$, say c_{k+1}, and hence $\mathbf{c}^p_{\sigma_1}[k+1].cfg = \mathbf{c}^q_{\sigma_2}[k+1].cfg = c_{k+1}$. Similar to the base case, a server in $c_k.Servers$ only receives the configuration c_{k+1} decided by the consensus run on index k.
Case (ii): Now, consider the case where p and q receive a configuration with index $i > k+1$. The two processes can receive i through $nextC$ of some server s in $c_j.Servers$ where $j < i$. This happens since a reconfiguration operation adds the index i, then executes the gc-config operation on every configuration with an index smaller than i and updates the $nextC$ of their servers to point to the configuration with index i (line Alg. 4:38). Thus if both processes p and q receive i, then $\mathbf{c}^p_{\sigma_1}[i].cfg = \mathbf{c}^q_{\sigma_2}[i].cfg = c_i$. However, if one of them does not receive index i, i.e. $cseq[i].cfg = \perp \notin C$ (due to a change of $nextC$ to a configuration with a larger index according to Lemma 7), it implies that its $cseq$ sequence at index i is denoted as $cseq[i].cfg = \perp \notin C$. This contracted our hypothesis, which assumes that $\mathbf{c}^p_{\sigma_1}[i].cfg, \mathbf{c}^q_{\sigma_2}[i].cfg \in C$.

□

Lemma 8 showed that any two operations store the same configuration in any cell k of their $cseq$ variable. However, whether the two processes discover the same maximum configuration id is still uncertain. In the following lemmas, we will demonstrate that if a process learns about a configuration in a cell k (and it is not its first configuration), it also learns about some configuration ids for some indices i such that $0 \leq i \leq k-1$. Notably, while the number of configurations may differ between processes, the subsequences must have the same maximum tag or a smaller one.

Lemma 9. *If at a state σ of an execution ξ of the algorithm $\lambda(\mathbf{c}^p_\sigma) = k$ for some process p, then for any element $0 \leq j < k$, $\exists Q \in \mathbf{c}^p_\sigma[j].cfg.Quorums$ such that $\forall s \in Q, s.nextC(\mathbf{c}^p_\sigma[j].cfg) = \mathbf{c}^p_\sigma[i]$, for some $i \in [j+1, k]$.*

Proof. Similar to Lemma 14 in [28] we can show that this lemma holds if for every $s \in Q$, $s.nextC(\mathbf{c}^p_\sigma[j].cfg) = \mathbf{c}^p_\sigma[j+1]$. As the gc-config changes the *nextC* of servers, we need to show that no server $s \in Q$ will have $s.nextC(\mathbf{c}^p_\sigma[j].cfg) = \mathbf{c}^p_\sigma[i]$ for $i < j+1$. This follows from the implementation of the gc-config as well as from Lemma 7. In particular, whenever a reconfigurer executes a gc-config propagates the latest finalized configuration to a quorum of each previous configuration in its sequence (line Alg 5:12). By Lemma 7, a server updates its *nextC* to point to a configuration with a larger index than the one it holds. Therefore, since a server s in a configuration $\mathbf{c}^p_{\sigma'}[j].cfg$ sets $s.nextC(\mathbf{c}^p_{\sigma'}[j].cfg) = \mathbf{c}^p_{\sigma'}[j+1]$ at some state σ' before σ then by Lemma 7 it can only have $s.nextC(\mathbf{c}^p_\sigma[j].cfg) = \mathbf{c}^p_\sigma[k]$, for $k \geq j+1$ in state σ. This completes the proof. □

Lemma 10 (Subsequence). *Let π_1 and π_2 be two completed read/write/reconfig operations 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 be the state after the response step of π_2. Then $\mathbf{c}^{p_1}_{\sigma_1} \sqsubseteq_p \mathbf{c}^{p_2}_{\sigma_2}$.*

Proof. By Lemma 8, 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 $\mathbf{c}^{p_1}_{\sigma_1}[i].cfg = \mathbf{c}^{p_2}_{\sigma_2}[i].cfg$. So it remains to show that $\lambda_1 \leq \lambda_2$.
Let $\lambda_1 = \lambda(\mathbf{c}^{p_1}_{\sigma_1})$ and $\lambda_2 = \lambda(\mathbf{c}^{p_2}_{\sigma_2})$. Since $\pi_1 \rightarrow \pi_2$, it follows that σ_1 appears before σ_2 in ξ. Let $\mu = \mu(\mathbf{c}^{p_2}_{\sigma'})$ be the last finalized element that p_2 established during operation π_2 at some state σ' before σ_2. It is easy to see that $\mu \leq \lambda_2$. If $\lambda_1 \leq \mu$, then $\lambda_1 \leq \lambda_2$, and the lemma follows. Thus, it remains to examine the case where $\mu < \lambda_1$. Notice that since $\pi_1 \rightarrow \pi_2$, then σ_1 appears before σ' in execution ξ. By Lemma 9, we know that by σ_1, there exists $Q \in \mathbf{c}^{p_1}_{\sigma_1}[j].cfg.Quorums$ for $0 \leq j < \lambda_1$ and for each $s \in Q$, s.nextC $= \mathbf{c}^{p_1}_{\sigma_1}[i]$, for some $i \geq j+1$, possibly different for every s. Since $\mu < \lambda_1$, 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 $= \mathbf{c}^{p_1}_{\sigma_1}[i]$, for $i \geq \mu+1$. Let p_2 as the outcome of an action (either get-next-config or DAP with piggyback) to read the configuration from a quorum $Q' \in \mathbf{c}^{p_2}_*[\mu].cfg$ during π_2,at a state σ''. Since by Lemma 8, $\mathbf{c}^{p_2}_*[\mu].cfg = \mathbf{c}^{p_1}_{\sigma_1}[\mu].cfg$, the Q and Q' belong to the same configuration and thus by definition $Q' \cap Q \neq \emptyset$. Therefore, there exists a server $s \in Q \cap Q'$ which by Lemma 7, replies to p_2 with either $s.nextC = \mathbf{c}^{p_1}_{\sigma_1}[\mu+1]$, or with $s.nextC = \mathbf{c}^r_*[j]$ for a $j > \mu+1$. If $j < \lambda_1$ then by a simple induction we can show that the process will be repeated in every configuration with index $k < \lambda_1$ until we reach at least λ_1. In that case $\lambda_2 \geq \lambda_1$. In case where $j \geq \lambda_1$ then p_2 will set $\mathbf{c}^{p_2}_{\sigma''}[j] = \mathbf{c}^r_*[j]$. Since σ_2 comes after σ'', then $\lambda_2 \geq j > \lambda_1$ and this completes the proof.

□

Lemma 11. *Let σ and σ' two states in an execution ξ such that σ appears before σ' in ξ. Then for any process p, that executes a* read-config *or a DAP action, 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 or DAP action from a configuration $\mathbf{c}^p_{\sigma'}[j].cfg$ such that $j \geq k$. The $\mathbf{c}^p_{\sigma'}[j].cfg$ is the last finalized configuration at σ' which is added by a subsequent reconfiguration. Hence, $\mu(\mathbf{c}^p_{\sigma'}) \geq \mu(\mathbf{c}^p_\sigma)$. □

Lemma 12 (Sequence Progress). *Let π_1 and π_2 two completed read/write/reconfig operations 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 10 it follows that $\mathbf{c}^{p_1}_{\sigma_1}$ is a subsequence of $\mathbf{c}^{p_2}_{\sigma_2}$. Thus, if $\lambda_1 = \lambda(\mathbf{c}^{p_1}_{\sigma_1})$ and $\lambda_2 = \lambda(\mathbf{c}^{p_2}_{\sigma_2})$, $\lambda_1 \leq \lambda_2$. Let $\mu_1 = \mu(\mathbf{c}^{p_1}_{\sigma_1})$, such that $\mu_1 \leq \lambda_1$, 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 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 11 $\mu_2 \leq \mu(\mathbf{c}^{p_2}_{\sigma_2})$ 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 or a DAP action.

Case (i): In case (i) p_1 propagates the $\langle \mathbf{c}^{p_1}_{\sigma_1}[\mu_1].cfg, F\rangle$ to a quorum of servers in every $\mathbf{c}^{p_1}_{\sigma_1}[j].cfg$, where $j \leq \mu_1 - 1$, before completing, using the gc-config. Thus, a quorum of servers in every $\mathbf{c}^{p_2}_{\sigma_2}[j].cfg$ ($j \leq \mu_1 - 1$), say Q, receives $\langle \mathbf{c}^{p_1}_{\sigma_1}[\mu_1].cfg, F\rangle$ from p_1. We know by Lemma 10 that since $\pi_1 \rightarrow \pi_2$ then $\mathbf{c}^{p_1}_{\sigma_1}$ is a subsequence of the $\mathbf{c}^{p_2}_{\sigma_2}$, it must be the case that $\mu_2 < \mu_1 \leq \lambda_2$. Thus, during π_2, p_2 starts from the configuration at index μ_2 and in some iteration performs get-next-config or DAP operation in a configuration $\mathbf{c}^{p_2}_{\sigma_2}[j]$, where $j \leq \mu_1 - 1$. 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 or DAP 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 or DAP is executed by p_2 there is already a quorum of servers in every $\mathbf{c}^{p_2}_{\sigma_2}[j].cfg$ ($j \leq \mu_1 - 1$) 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 configuration with index j, say Q', then by Lemma 7 there is a server $s \in Q \cap Q_{'}$, such that s replies to p_2 with $s.nextC = \langle \mathbf{c}^{p_1}_{\sigma_1}[\mu_1].cfg, F\rangle$ or a server $s \in Q_{'}$ replies to p_2 with $s.nextC = \langle \mathbf{c}^{r}_{*}[j].cfg, F\rangle$ where r a reconfigurer that propagated the configuration in a gc-config action and $j > \mu_1$. Hence, $\mu(\mathbf{c}^{p_2}_{\sigma_2}) \geq \mu_1$ in this case.

Case (ii): In this case p_1 sets the status of $\mathbf{c}^{p_1}_{\sigma_1}[\mu_1]$ to F 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 or a DAP action. Process p_1 propagates $\langle \mathbf{c}^{p_1}_{\sigma_1}[\mu_1].cfg, F\rangle$ using the action put-config to a quorum of servers in configuration $\mathbf{c}^{p_1}_{\sigma_1}.cfg$. Similar to Case (i) it holds that $\mu_2 < \mu_1 \leq \lambda_2$. So at σ_2, $\mathbf{c}^{p_2}_{\sigma_2}[\lambda_2]$ is the last configuration in $\mathbf{c}^{p_2}_{\sigma_2}$. By our algorithm p_2 discovers this configuration either by traversing the configurations from index μ_2 to λ_2, or it discovers $\mathbf{c}^{p_2}_{\sigma_2}[\lambda_2]$ from a configuration $\mathbf{c}^{p_2}_{\sigma_2}[j].cfg$ for $j < \lambda_2 - 1$. In the latter case, it must hold that $\mathbf{c}^{p_2}_{\sigma_2}[\lambda_2].status = F$ since the gc-config action only propagates finalized configurations. Hence, in this case $\mu_2 = \lambda_2 > \mu_1$. By Lemma 9 every configuration $\mathbf{c}^{p_2}_{\sigma_2}[j]$ has at least one quorum whose servers have at least $s.nextC = \mathbf{c}^{p_2}_{\sigma_2}[j+1]$. So in the case of traversing the configurations p_2 will either perform a get-next-config or DAP action on $\mathbf{c}^{p_2}_{\mu_1 - 1}.cfg$ or it will avoid μ_1 as it will discover a finalized configuration $k > \mu_1$. In the latter case $\mu_2 \leq k > \mu_1$. In the initial case p_2 will access $\mathbf{c}^{p_1}_{\sigma_1}.cfg$ since by Lemma 8, $\mathbf{c}^{p_1}_{\sigma_1}[\mu_1 - 1] = \mathbf{c}^{p_2}_{\sigma_2}[\mu_1 - 1]$ if both are non empty. We know that p_1 propagated $\langle \mathbf{c}^{p_1}_{\sigma_1}[\mu_1].cfg, F\rangle$ to $\mathbf{c}^{p_1}_{\sigma_1}[\mu_1 - 1].cfg$ before completing (Algo.2:15). p_2 executes get-next-config or DAP on $\mathbf{c}^{p_1}_{\sigma_1}[\mu_1 - 1].cfg$. Thus, some server in Q will reply to p_2 with $s.nextC = \mathbf{c}^{p_1}_{\sigma_1}\mu_1$ which is finalized. Thus, $\mu_2 \geq \mu_1$ in this case as well. □

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

Theorem 13. *Let π_1 and π_2 two completed* read-config *or DAP 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) ***Configuration Consistency:*** $\mathbf{c}^{p_2}_{\sigma_2}[i].cfg = \mathbf{c}^{p_1}_{\sigma_1}[i].cfg$, *for* $1 \leq i \leq \nu(\mathbf{c}^{p_1}_{\sigma_1})$,

(b) ***Subsequence:*** $\mathbf{c}^{p_1}_{\sigma_1} \sqsubseteq_p \mathbf{c}^{p_2}_{\sigma_2}$, *and*
(c) ***Sequence Progress:*** $\mu(\mathbf{c}^{p_1}_{\sigma_1}) \leq \mu(\mathbf{c}^{p_2}_{\sigma_2})$

Proof. Statements (a), (b) and (c) follow from Lemmas 8, 10, and 12. □

6.2.2 Atomicity Property of Ares II. Ares II is correct if it satisfies *liveness* (termination) and *safety* (i.e., linearizability).

Lemma 14. *Every read/write/reconfig operation terminates in any execution ξ of* Ares II.

Proof. Termination holds since read, write and reconfig operations on Ares II always complete given that the DAP completes. □

Lemma 15. *In any execution ξ of* Ares II, *if π_ρ a read operation invoked by some process p that appears in ξ, then π_ρ returns a value $v \neq \bot$.*

Proof. In this proof we need to show that any read operation π_ρ in ξ will decode and return some value $v \neq \bot$. But the read π_ρ, may receive $\bot$, as the returned values of get-data actions, which the read calls repeatedly get-data until it cannot discover a new configuration.
Let's assume by contradiction that p_ρ may return a value $\bot$. The value returned by $p_r d$ is specified by Line 37 when the reader discovers the max $\langle \tau_{max}, v_{max} \rangle$ pair s.t. $v_{max} \neq \bot$ out of all pairs returned by the get-data operations it invoked. So in order for p_ρ to return $\bot$ then it must be the case that all get-data actions invoked returned a $\bot$ value.
A c.get-data action, invoked in a configuration c, returns a $\bot$ value only if it receives a $\bot$ value in some server reply from a quorum Q in $c.\mathcal{S}$. A server $s \in Q$ in turn, may return a $\bot$ value in two cases: (i) its variable $s.nextC(c).status = F$ Line 10, or (ii) it received a message from a gc-collect action Lines Alg. 4:39–41. Notice, that in both cases $s.nextC(c).status = F$, as the gc-collect action propagates a finalized configuration. So, assuming that c has an index i, then the $s.nextC(c) = \langle c', F \rangle$ for some c' with index $j > i$.
Action c.get-data will include c' in the set of configurations that it returns. Therefore, the set Cs observed by the reader will contain at least c', and hence the reader will repeat c'.get-data in c'. Let w.l.o.g c' be the last finalized configuration in the introduced list of configurations $\mathcal{G}_L$. There are two cases to examine: (i) there is no other configuration after c', or (ii) there exists some pending configuration c'' after c'.
In the case where c' is the last configuration then for every server $s' \in c'.\mathcal{S}$, $s'.nextC = \bot$. Thus, every s' replies to c'.get-data with a value $v \neq \bot$ (v will be at least v_0). Since we cannot have more than δ concurrent put-data similar to Lemma 2 we will find and return a decodable value say v'. So the reader will find $v' \neq \bot$ and this case contradicts our initial assumption.
So it remains the case where there exists a pending configuration c'' after c'. In this case the Cs set at the reader will not be empty and it will repeat c''.get-data operation in c''. As c' is the last finalized configuration then no server $s'' \in c''.\mathcal{S}$ will have a $nextC$ variable with $s''.nextC(c'').status = F$. As c'' is still pending then there exists a concurrent reconfigurer that tries to install c''. So we have two cases when c''.get-data is invoked: (i) the reconfigurer managed to write the latest tag-value pair in a quorum in $c''.Quorums$, or (ii) not. In the first case the c''.get-data action will find the written value in at least k lists and thus will decode and return a value v'' to the reader. In the second case the servers in $c''.\mathcal{S}$ will return at least the initial value v_0 so the action c''.get-data will be able to decode and return at least $\langle \tau_0, v_0 \rangle$. In any of those cases the reader will return either v', v'', or v_0 based on which tag of those values is greater. Since any of those values belong in $\mathcal{V}$ they are different than $\bot$ and they also contradict our initial assumption. This completes our proof. □

Theorem 16. *In any execution ξ of* Ares II, *if in every configuration* $c \in G_L$, c.get-data(), c.put-data(), *and* c.get-tag() *satisfy Property 1, then* Ares II *satisfies atomicity.*

Proof. As shown in [28], Ares implements a linearizable object given that the DAP used satisfy Property 1. In Ares II, the read and write operations have similar behaviour with that of Ares, even though it unites the read-config operations with the get-tag, get-data and put-data operations into one communication round. One main difference introduced by this change in execution is that the get-tag and get-data are executed until an empty configuration is found. Additionally, the algorithm transfers only the Cs with next configurations in DAP operations without the tag/data (similar to how the read-config does) when the status of the next configuration is finalized. In contrast, in Ares, after completing the read-config and reaching $\perp$, it then proceeds with the get-tag/get-data. The main challenge to the proof is to show that Ares II satisfies the linearizability despite, the changes in DAPs (i.e., piggy-back), the new addition of a garbage collection and the optimization in the reconfiguration operation.
By Ares II, before a read/write/reconfig operation completes it propagates the maximum tag, or the new tag in the case of a write it discovered, by executing the put-data action in the last configuration of its local configuration sequence (Lines Alg.5:20, Alg.1:20 & 43). When a subsequent operation is invoked, it reads the latest configuration sequence by beginning from the last finalized configuration in its local sequence and invoking get-tag/get-data to all the configurations until the end of that sequence.
Lemma 15 shows that a read operation retrieves a tag-value pair, where the value is from $\mathcal{V}$. In addition the reconfiguration properties help us show that the consistency of operations is preserved. Finally, the batching mechanism cannot be distinguished from multiple reconfigurations on different objects, hence the correctness of read/write operations is not affected. □

7 OPTIMIZATION RESULTS

In this section, we analyze the performance improvements yielded from optimizations in Section 5.

7.1 Piggyback

In this experiment we use the same scenario as in Section 4. To demonstrate the effectiveness of the *piggyback* optimization from Section 5.1, we conducted a comparison between the original algorithms and their optimized counterparts.

alg./f_{size}	AresABD	AresABD *PB*	AresEC	AresEC *PB*	CoAresABD	CoAresABD *PB*	CoAresEC	CoAresEC *PB*	CoAresABDF	CoAresABDF *PB*	CoAresECF	CoAresECF *PB*
1MB	343ms	334ms	257ms	251ms	308ms	302ms	136ms	127ms	284ms	278ms	149ms	142ms
256MB	72s	72s	54s	53s	71s	63s	25s	27s	9s	5s (44%)	9.65s	3.82s (60%)
512MB	264s	192s	109s	106s	176s	164s	55.6s	52.4s	21.8s	15.2s (30%)	23.2s	10.9s (53%)

Table 2. READ Operation - File Size - S:11, W:5, R:5:

In Table 2, a comprehensive comparison between the original algorithms and their optimized counterparts with the *piggyback* optimization is presented for READ operations with three different object sizes (1 MB, 256 MB, and 512 MB). In non-fragmented algorithms, no improvements are observed. Handling only one relatively medium-sized object, the removal of read-config which happened only one time does not substantially impact the latencies. In contrast, CoAresF exhibits higher drops in the case of 256 MB and 512 MB. In this experiment, CoAresF uses 4 communication rounds, while its *PB*-optimized counterpart completes it in 2 rounds. Due to the DAP optimization (outlined in Section 3) for get-data and put-data, often no data transfer occurs (e.g., in CoAresECF, fast reads average 13.8 ms, while slow reads take 74.1 ms. With *PB*, fast reads are 5.84 s, and slow reads are 67.1 ms.). Also, as there is only one configuration, read-config transfers an empty *nextC*. Thus, reducing the rounds to 2 with *PB* optimization drastically reduces the latency of fast reads. This demonstrates that when combined with DAP optimization, *PB* significantly reduces latencies in fragmentation (for the red-highlighted times in Table 2, see the improvement percentage).

7.2 Garbage Collection

The scenario below is made to measure the performance of algorithms when we apply the garbage collection optimization. So, we evaluate the algorithms Ares, CoAres, and CoAresF, along with their counterparts with *PB* and *PB* with *GC*. We used two different sizes of the object, 1 MB and 64 MB. The maximum, minimum, and average block sizes (parameters for *rabin fingerprints*) are set to 1 MB, 512 kB, and 512 kB respectively. We set $|\mathcal{W}| = 1, |\mathcal{R}| = 10, |\mathcal{G}| = 4, |\mathcal{S}| = 11$. For EC-based algorithms we used parity $m = 5$ yielding quorum sizes of 9 and for ABD-based algorithms we used quorums of size 6. Initially, a writer invokes a write operation, and subsequently, the 4 reconfigurers change the configuration using round-robin fashion three times each, with the set of servers remaining the same in all the configurations. The reconfigurers only modify the DAP switching between the ABD and the EC, starting from the EC. Finally, each reader reads the object in serial order.

alg./f_{size}	Ares	Ares *PB*	Ares *PB&GC*	CoAres	CoAres *PB*	CoAres *PB&GC*	CoAresF	CoAresF *PB*	CoAresF *PB&GC*
11 Pending Reconfiguration & 1 Finalized									
1MB	159ms	494ms	107ms	162ms	506ms	110ms	181ms	191ms	127ms
64MB	5.57s	27.4s	5.58s	5.81s	26.8s	5.73s	6.78s	6.62s	6.61s
12 Finalized Reconfiguration									
1MB	159ms	166ms	119ms	163ms	167ms	122ms	186ms	193ms	135ms
64MB	5.80s	5.76s	5.71s	5.88s	5.98s	5.82s	6.92s	6.73s	6.74s

Table 3. READ Operation - Rreconfigurations - S:11, W:1, R:10:, G:4

The Table 3 is divided into two scenarios. In the first scenario, the first 11 reconfigurations remain pending, and the last reconfiguration is completed as finalized. As we can see, in this first scenario the version with worst read latency between the three (without optimizations, with *PB*, and with *PB* and *GC*), is the algorithms that have *PB*. This happens since as we can see in Fig. 15 the *PB* version of algorithms transfers the data along with the next configuration (*nextC*) in all round trips. This issue is then solved by the *GC* optimization of the last reconfiguration, which changes the pointers of all the configurations before the proposed one to point to the finalized configuration. In this way, when the reader starts its read (from c_0) after the last finalized reconfiguration, it finds c_{12} as the next configuration. The read operation in algorithms without optimizations involves performing 12 read-config operations followed by 1 get-data operation to fetch the data. This is why the difference between the two algorithms is evident in the smaller object size (1 MB). However, in the fragmented algorithms, the read operation finds the last finalized configuration during the first block, and subsequent blocks start from that configuration. Thus, the difference between the versions of the fragmented algorithm will only become apparent if a reconfiguration occurs between every block read operation. Additionally, when (64 MB), the *CoAresF* with *PB* does not outperform the original algorithm as it did in the other cases. This is because it has a larger number of smaller blocks, which has to migrate to the new configuration.

In the second scenario, all 12 reconfigurations are completed and finalized (Fig. 16). The non-fragmented original algorithms and the ones with *PB* have negligible differences between them, as

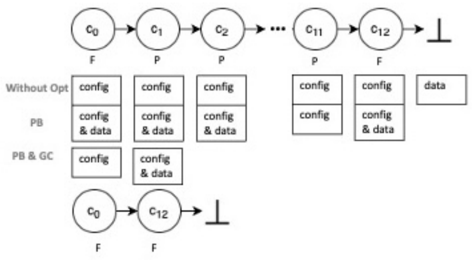


Fig. 15. 11 Pending Reconfigurations & 1 Finalized

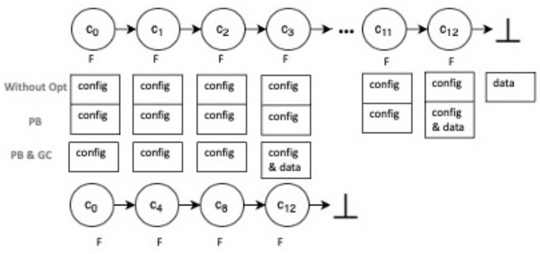


Fig. 16. 12 Finalized Reonfigurations

the only distinction lies in the one extra round trip required by the former to fetch the last finalized configuration, while the latter fetches the data with the last finalized configuration. However, the algorithms with *GC* reduce the number of communication rounds required to traverse the configuration sequence, as they skip every 4 configurations. Once again, the benefits of fragmented algorithms with *GC* are apparent only in the first block.

8 CONCLUSIONS

In this work, we emphasize the significance of identifying and addressing performance bottlenecks within DSM, specifically with Ares, and introduce the use of Distributed Tracing as a methodology for pinpointing these bottlenecks. By injecting checkpoints and monitoring the performance of individual procedures, tracing enables the detection of system inefficiencies. We then turned the identified inefficiencies into optimizations, without jeopardizing correctness, yielding Ares II.
For future work, it would be interesting to devise strategies on *when* new configurations are introduced, such that performance is optimized while prolonging liveness.

REFERENCES

[1] . Grafana. https://grafana.com. Accessed: [14/02/2024].

[2] . Opentelemetry. https://opentelemetry.io. Accessed: [14/02/2024].

[3] . OpenTelemetry-Python: A vendor-agnostic framework for distributed tracing, metrics, and logging. https://github.com/open-telemetry/opentelemetry-python.

[4] . ZeroMQ. https://zeromq.org. Accessed: [14/02/2024].

[5] . Zipkin. https://zipkin.io. Accessed: [14/02/2024].

[6] M.K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. 2009. Dynamic atomic storage without consensus. In *Proceedings of the 28th ACM symposium on Principles of distributed computing (PODC '09).* ACM, New York, NY, USA, 17–25.

[7] C. Aniszczyk. 2012. Distributed Systems Tracing with Zipkin. https://blog.twitter.com/2012/distributed-systems-tracing-with-zipkin Google Scholar.

[8] A.F. Anta, C. Georgiou, T. Hadjistasi, E. Stavrakis, and A. Trigeorgi. 2021. Fragmented Object : Boosting Concurrency of Shared Large Objects. *In Proc.of SIROCCO* (2021), 1–18.

[9] H. Attiya, A. Bar-Noy, and D. Dolev. 1995. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM (JACM)* 42, 1 (1995), 124–142.

[10] A. Berger, I. Keidar, and A. Spiegelman. 2018. Integrated Bounds for Disintegrated Storage. In *Proceedings of the 32nd International Symposium on Distributed Computing (DISC).* Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 11:1–11:18. https://doi.org/10.4230/LIPIcs.DISC.2018.11

[11] A. Carpen-amarie. 2012. BlobSeer as a Data-Storage Facility for Clouds: Self-Adaptation, Integration, Evaluation, PhD Thesis, France. (2012).

[12] Datadog. . Datadog Overview. https://www.datadog.com/overview Accessed: [14/02/2024].

[13] P. Dutta, R. Guerraoui, R.R. Levy, and A. Chakraborty. 2004. How Fast can a Distributed Atomic Read be? *In Prof. of PODC* (2004), 236–245.

[14] E. Gafni and D. Malkhi. 2015. Elastic configuration maintenance via a parsimonious speculating snapshot solution. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9363 (2015), 140–153. https://doi.org/10.1007/978-3-662-48653-5_10

[15] C. Georgiou, R. Gustafsson, A. Lindhé, and E. M. Schiller. 2019. Self-stabilization Overhead: A Case Study on Coded Atomic Storage. In *Networked Systems*, Mohamed Faouzi Atig and Alexander A. Schwarzmann (Eds.). Springer International Publishing, Cham, 131–147.

[16] C. Georgiou, T. Hadjistasi, N. Nicolaou, and A. A. Schwarzmann. 2022. Implementing Three Exchange Read Operations for Distributed Atomic Storage. *J. Parallel Distributed Comput.* 163 (2022), 97–113. https://doi.org/10.1016/j.jpdc.2022.01.024

[17] C. Georgiou, P. M. Musiał, and A. A. Shvartsman. 2005. Developing a consistent domain-oriented distributed object service. *Proceedings - Fourth IEEE International Symposium on Network Computing and Applications, NCA 2005* 2005, 149–158. https://doi.org/10.1109/NCA.2005.16

[18] C. Georgiou, N. Nicolaou, and A.A. Shvartsman. 2009. Fault-tolerant Semifast Implementations of Atomic Read/Write Registers. *J. Parallel and Distrib. Comput.* 69, 1 (2009), 62–79.

[19] C. Georgiou, N. Nicolaou, and A. Trigeorgi. 2022. Fragmented ARES: Dynamic Storage for Large Objects. In *Proceedings of the 36th International Symposium on Distributed Computing (DISC).* 25:1–25:24. Also at arXiv:2201.13292..

[20] S. Gilbert, N. A. Lynch, and A. A. Shvartsman. 2010. RAMBO: A Robust, Reconfigurable Atomic Memory Service for Dynamic Networks. *Distributed Comput.* 23, 4 (2010), 225–272. https://doi.org/10.1007/s00446-010-0117-1

[21] Google Cloud Trace. . Google Cloud Trace Overview. https://cloud.google.com/trace Accessed: [14/02/2024].

[22] T. Hadjistasi, N. Nicolaou, and A. A. Schwarzmann. 2017. Oh-RAM! One and a Half Round Atomic Memory. In *Proc. of NETYS 2017 (Lecture Notes in Computer Science, Vol. 10299).* 117–132. https://doi.org/10.1007/978-3-319-59647-1_10

[23] M. P. Herlihy and J. M. Wing. 1990. Linearizability: a Correctness Condition for Concurrent Objects. *ACM TOPLAS* 12, 3 (1990), 463–492.

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

[25] L. Jehl and H. Meling. 2017. The Case for Reconfiguration without Consensus: Comparing Algorithms for Atomic Storage. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 70)*, Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 31:1–31:17. https://doi.org/10.4230/LIPIcs.OPODIS.2016.31

[26] L. Jehl, R. Vitenberg, and H. Meling. 2015. Smartmerge: A new approach to reconfiguration for atomic storage. In *International Symposium on Distributed Computing.* Springer, 154–169.

[27] N.A. Lynch and A.A. Shvartsman. 1997. Robust Emulation of Shared Memory Using Dynamic Quorum-Acknowledged Broadcasts. *In Proc. of FTCS* (1997), 272–281.

[28] N. Nicolaou, V. Cadambe, N. Prakash, A. Trigeorgi, K. M. Konwar, M. Medard, and N. Lynch. 2022. ARES: Adaptive, Reconfigurable, Erasure coded, Atomic Storage. *ACM Transactions on Storage (TOS)* (2022). Accepted. Also in https://arxiv.org/abs/1805.03727.

[29] N. Nicolaou, A. Fernández Anta, and C. Georgiou. 2016. Coverability: Consistent Versioning in Asynchronous, Fail-Prone, Message-Passing Environments. In *Proc. of IEEE NCA 2016*. IEEE.

[30] B. Sigelman, Luiz A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. 2010. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Google Inc.

[31] B. Sigelman and contributors. 2016. OpenTracing. https://opentracing.io/ Accessed: [14/02/2024].

[32] A. Spiegelman, Y. Cassuto, I. Keidar, and G. Chockler. 2016. Space Bounds for Reliable Storage: Fundamental Limits of Coding. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. ACM. https://doi.org/10.1145/2933057.2933104

[33] A. Spiegelman, I. Keidar, and D. Malkhi. 2017. Dynamic Reconfiguration: Abstraction and Optimal Asynchronous Solution. In *Proceedings of the 31st International Symposium on Distributed Computing (DISC)*, Andréa W. Richa (Ed.). Leibniz International Proceedings in Informatics, Dagstuhl Publishing, Germany, 40:1–40:15. https://doi.org/10.4230/LIPIcs.DISC.2017.40 A full version of the paper is available at https://alexanderspiegelman.github.io/alexanderspiegelman.github.io/DynamicTasks.pdf..

[34] A. Trigeorgi, N. Nicolaou, C. Georgiou, T. Hadjistasi, E. Stavrakis, V. Cadambe, and B. Urgaonkar. 2022. Invited Paper: Towards Practical Atomic Distributed Shared Memory: An Experimental Evaluation. In *Stabilization, Safety, and Security of Distributed Systems: 24th International Symposium, SSS 2022, Clermont-Ferrand, France, November 15–17, 2022, Proceedings* (Clermont-Ferrand, France). Springer-Verlag, Berlin, Heidelberg, 35–50. https://doi.org/10.1007/978-3-031-21017-4_3

[35] Uber Technologies, Inc. 2023. Jaeger Distributed Tracing. https://www.jaegertracing.io/ Accessed: [14/02/2024].

[36] M. Vukolic. 2012. *Quorum Systems: With Applications to Storage and Consensus*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00402ED1V01Y201202DCT009