Invited Paper: Towards Practical Atomic Distributed Shared Memory: An Experimental Evaluation

Andria Trigeorgi[1,2](✉), Nicolas Nicolaou[1], Chryssis Georgiou[2], Theophanis Hadjistasi[1], Efstathios Stavrakis[1], Viveck Cadambe[3], and Bhuvan Urgaonkar[3]

[1] Algolysis Ltd., Limassol, Cyprus
[2] Department of Computer Science, University of Cyprus, Nicosia, Cyprus
`trigeorgi.andria@ucy.ac.cy`
[3] Pennsylvania State University, State College, PA, USA

Abstract. Distributed Shared Storage Services may serve as building blocks to yield complex, decentralized, cloud applications in emerging technologies (e.g., IoT, VR/AR), as they offer a transparent cloud storage space where distributed applications can store, retrieve, and coordinate over shared data. Ideally, distributed applications would like to communicate through a "cloud" memory layer that may provide similar guarantees as a centralized sequential memory. Atomic Distributed Shared Memory (ADSM) provides the illusion of a sequential memory space despite asynchrony, network perturbations, and device failures. A plethora of algorithmic solutions along with proven correctness guarantees have been proposed to provide ADSM in a message passing system. None of them, however, has been adopted in a real working solution: commercial solutions avoid the use of ADSM algorithms, mainly due to their communication overhead. *But what is exactly the performance overhead of an ADSM algorithm over existing commercial solutions?* In this work we want to provide a first answer to this question by performing an in-depth experimental comparison of the state-of-the-art dynamic ADSM algorithm ARES, with two well-established open-source distributed storage solutions, Cassandra and Redis. The results show that ARES's performance is comparable with the commercial systems, with respect to scalability, object size and throughput.

Keywords: Distributed storage · Strong consistency · Erasure code · Reconfiguration · Fault-tolerance

1 Introduction

Motivation and Prior Work. Emulating a shared memory over a set of distinct, often geographically dispersed devices, is a fundamental problem in

Supported by the EU's NGIAtlantic.eu cascading grant agreement no. OC4-347; https://projects.algolysis.com/ares-ngi/.

distributed computing, and an important tool for the development of dependable and robust distributed applications [6,14]. A Distributed Shared Memory (DSM) service promises to provide an available, accessible, and survivable shared memory space over an asynchronous, fail prone, message passing environment. To preserve these properties, data are replicated in multiple devices, referred to as servers or replica hosts, raising the challenge on how to preserve *consistency* between the replica copies. Different consistency guarantees were rigorously defined over the years [16]. *Atomicity* is a venerable notion of consistency, introduced by Lamport [19]. To this day it remains the most natural type of consistency because it provides an illusion of equivalence with the serial object type that software designers expect. For more than two decades, a series of works, e.g., [4,7,9–11,13,20,21], suggested solutions for building Atomic DSM (ADSM) emulations, for both static, i.e., where replica participation does not change over time, and dynamic (reconfigurable) environments, i.e., where failed replicas may retire and new replicas may join the service in a non-blocking manner.

It is apparent that those solutions cannot be found readily and were not adopted by commercial distributed storage applications. Commercial Distributed Storage Systems (DSS), such as Dropbox, HDFS, Cassandra and Redis, avoid providing strong consistency guarantees (such as atomicity) as they are considered costly and difficult to implement in an asynchronous, fail prone, message passing environment. Hence, such solutions either choose to offer weaker or tunable guarantees to achieve better performance when atomicity is not preserved.

Indeed, initial implementations of ADSM had high demands in communication, storage, and sometimes computation. Recent works, however, e.g., [12,21], invest in algorithms that may reduce the overheads on the aforementioned parameters. ARES [21] is a recent ADSM algorithm, which proposes a modular approach for providing a *dynamic* shared memory space. ARES may use any ADSM algorithm at its core, providing the flexibility to adjust its performance based on the application demands. Fragmented ARES [12] is an extension of ARES that supports versioning and fragmentation for efficiently handling large objects, such as files.

Experimental results presented in [12,21], demonstrated a promising performance of the algorithm under various environmental conditions and data loads. *But how such an algorithm may compare to commercially used solutions?* That is, no evidence exists to date to examine what are the gains from commercial solutions to adopt less than intuitive guarantees. In this work we set to put ADSM and chosen open-source, commercial solutions in a head-to-head comparison in order to answer the question: *Is it worth to trade consistency for performance?*

Contributions. In this work we perform an in-depth experimentation on ARES [21] and we present extensive comparison with two open-source widely used distributed storage solutions: (*i*) Cassandra [1], and (*ii*) Redis [2]. To this respect, we have developed our own implementation of ARES, and we have utilized the open source code of Cassandra and Redis.

Our experimental study focuses on measuring the average operation latency (communication and computation), in the following three test categories:

- **Scalability Tests:** Aim to test the ability of the service while the set of service participants grows.
- **Stress Tests:** Aim to test the performance of the service under various loads, concurrency patterns, and topology deployments.
- **Fault-Tolerance Tests:** Aim to test the tolerance of the service to node failures and its performance in necessary reconfigurations.

We deployed our experiments in real testbeds, distributed in the European Union (EU) and the USA. Such deployment helped us obtain real-condition results and evaluate the algorithms over cross-Atlantic setups. To the best of our knowledge, this is the first work to conduct such comparison. Our experimentation results suggest, perhaps surprisingly, that ARES has a similar or sometimes better performance than the competition, even without any optimization.

2 Algorithms Overview

In this section we provide a high-level description of the algorithms we examine in this work, highlighting their main differences.

2.1 ARES

ARES [21] is a modular framework, designed to implement dynamic, reconfigurable, fault-tolerant, read/write distributed atomic shared memory objects. Similar to traditional implementations, ARES uses $\langle tag, value\rangle$ pairs to order the operations on a shared object. In contrast to existing solutions, ARES does not define the exact methodology to access the object replicas. Rather, it relies on three, so called, *data access primitives* (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($\langle tag, v\rangle$), which accepts a $\langle tag, value\rangle$ as an argument.

DAPs. As detailed in [21], these DAPs may be used to express the data access strategy, i.e., how they retrieve and update the object data, of different shared memory algorithms (e.g., [6]). Using the DAPs, ARES achieves a modular design, agnostic of the data access strategies, and enables the use of different DAP implementation per configuration (something impossible for other solutions). For the DAPs to be useful, they need to satisfy *Property 1* [21], which informally states that a get-data (or get-tag) DAP returns a value (or tag) at least as recent as the one written by a put-data.

To demonstrate the flexibility that DAPs provide, the authors in [21] expressed two different atomic shared R/W algorithms in terms of DAPs. These are the DAPs for the well celebrated ABD [7] algorithm, and the DAPs for an erasure coded based approach presented for the first time in [21]. In the rest of the manuscript we refer to the two DAP implementations as ABD-DAP and EC-DAP. In EC-DAP, an $[n, k]$-MDS erasure coding algorithm (e.g., Reed-Solomon [25]) encodes k object fragments into n coded elements, which consist of the k encoded data fragments and m encoded parity fragments. The n coded fragments are distributed among a set of n different servers. Any k of the n coded fragments can then be used to reconstruct the initial object value. As servers maintain a fragment instead of the whole object value, EC-based approaches claim significant storage benefits. To reduce the communication overhead and yet preserve atomicity, servers maintain the last δ values they have seen, such that $\delta = |W|$ the set of writers, and thus the number of concurrent write operations. By utilizing the EC-DAP, ARES became *the first* erasure coded dynamic algorithm to implement an atomic R/W object. We refer as ARES-ABD and ARES-EC the versions of ARES using ABD-DAP and EC-DAP, respectively.

We now provide a high-level description of the two main functionalities supported by ARES: (i) the reconfiguration of the servers, and (ii) the read/write operations on the shared object.

Reconfiguration. Reconfiguration is the process of changing the set of servers. In high-level, ARES maintains a sequence of configuration ids. Whenever a server wants to introduce a new configuration, it performs the following steps: (1) it parses the configuration sequence to find the last configuration id proposed, (2) it proposes a new configuration to extend the sequence via an external *consensus* service, and (3) if its proposal is accepted, it moves the value of the object from the old configurations to the new, and then appends the id of the new configuration to the end of the sequence. The reconfiguration protocol ensures that the sequence remains connected, does not have any gaps, and it is the same for any participant in the system. The whole process is *non-blocking*, that it, the reconfiguration does not block the read/write operations on the object.

Reads/Writes. Read and write operations act as follows: (1) parse the sequence to find the latest configuration (read-config), (2) read the "latest" (based on the *tag*) value (if it is a read) or only the tag (if it is a write) of the object from that configuration (using DAPs), (3) get in a loop to propagate the latest (if its a read) or the new (if its a write) value to the latest configuration in the sequence (using DAPs and read-config), (4) terminate if no new configuration is discovered. The last two steps serve to propagate the value to new configurations as they become available. Essentially read and writes catch up with the latest configuration. Detailed analysis appears in [21].

Implementation. As we already mentioned, for the purposes of this study we have developed our own implementation of ARES. Our implementation is based on the architecture depicted in Fig. 1. This includes the modules composing the infrastructure as well as the communication layer between these modules. The system is composed of two main modules: (i) a Manager, and (ii) a Distributed Shared Memory Module (DSMM). The manager provides an interface to each client for accessing the DSM (in our case a command line interface - CLI). Following this architecture, clients may access the file system through the Manager, while the shared objects are maintained by the servers through the DSMM. Notice that the Manager uses the DSMM as an external service to write and read objects to the shared memory. To this respect, our architecture is flexible enough to utilize any underlying DSM algorithm to implement the DSMM. In our case

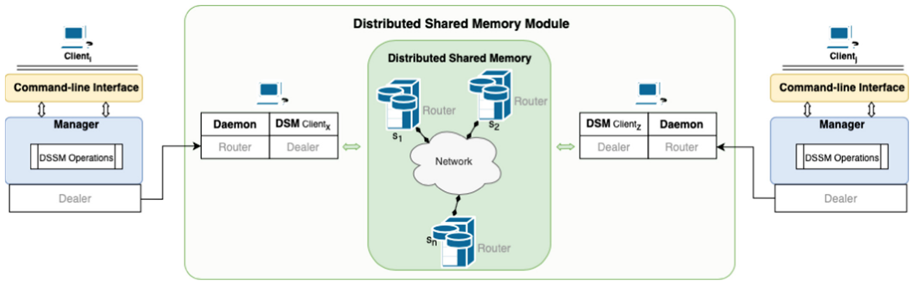


Fig. 1. The architecture of our ARES implementation.

we implemented two algorithms. At first, we integrated algorithm ABD to our DSM Module. Next, we implemented algorithm ARES with two different DAPs (ABD and EC) and then we integrated that implementation to the DSM Module. Python was chosen as the programming language and ZeroMQ [27] messaging library written in Python (the Dealer-Router paradigm) for the underlying communication. For the EC algorithm, we use the standard Reed-Solomon implementation provided by liberasurecode from the PyEClib Python library [23]. Notice that the implementation of ARES requires a consensus algorithm to be implemented as well. So, we implemented the RAFT [22] consensus algorithm, utilizing an open-source implementation of RAFT, also written in Python [24].

2.2 Cassandra

Cassandra [1] is a NoSQL distributed database offering continuous availability, high performance, horizontal scalability, and a flexible approach with tunable parameters. It was initially developed by Facebook for their inbox search feature. Today, it is an open-source application of Apache Hadoop. Cassandra uses peer-to-peer communication where each node is connected to all other nodes. The protocol used to achieve this communication is gossip, in which nodes periodically exchange state information about themselves. All the nodes in a cluster can serve read and write requests. Thus, when a request is sent to any node, this node acts as the coordinator. The coordinator distributes execution around the cluster, gathers the responses from the replicas, and responds back to the client. By default, Cassandra guarantees *eventual consistency*, which implies that all updates reach all replicas eventually. However, Cassandra offers tunable consistency for read and write operations, so that the system can guarantee weaker or stronger consistency, as required by the client application. The required consistency can be achieved by tuning the consistency level (CL) and the replication factor (RF) parameters. RF specifies how many copies of a store object (i.e., a row in Cassandra's Database) is kept among the participants. Given the value of the RF, the CL controls how many responses the coordinator waits for before the operation is considered complete. Finally, Cassandra allows the removal and addition of a single node at a time, in contrast to ARES that allows a complete modification of the configuration (reconfiguration) in a single operation.

Implementation. We deployed the Apache Cassandra 4 on multiple nodes with Ubuntu 18.04.1 LTS or 20.04 LTS. In order to guarantee atomicity, as in ARES and ABD, we set the CL parameter of Cassandra to "quorum". This means that a majority of nodes of the replicas must respond. Thus, if n is the total number of available replicas, and RF is n, then $n/2$ +1 must respond. To send read and write request we created a script using the Cassandra-driver Python library. First, the script creates connections to the cluster nodes, giving their IPs and ports. Then we specify a keyspace (a namespace that defines data replication on nodes) and create a table (a list of key-value pairs). Once that is done, the client can send write and read requests, using the *insert* and *select* statements, respectively. A writer inserts a tuple $(fileid, value)$, where the value is a byte string of type blobs (binary large objects) in Cassandra. A reader selects the value providing the file's id.

2.3 Redis

Redis [2] is an open source, in-memory key-value store. The read/write response time for Redis is extremely fast since all the data is in memory. Redis is based on a Master-Slave architecture, i.e., it enables replication of master Redis instances in replica Redis instances. The use of Redis is rather easy; Redis will internally store the key and value when users execute commands like set key value. Redis returns the value with a simple get key command from the user. The data size cannot exceed the main memory limit because all the data are in main memory. Redis has two persistence mechanisms: RDB (Redis Database Backup) and AOF (Append Only File). RDB persistence provides point-in-time snapshots of the database at specified intervals. AOF persistence logs every write operation. When the database server starts, Redis reads the AOF log to reconstruct the database. RDB is perfect for backup, but if RDB stops working all data changes since the last snapshot are lost. In comparison, AOF has better durability, although adopting AOF persistence may result in performance loss. Redis has a command called "WAIT" in order to implement synchronous replication. This command blocks the current client until all the previous write commands are successfully transferred and acknowledged by at least the specified number of replicas. Redis provides eventual consistency. Even though a write may wait until all replicas reply, reads do not wait and always terminate as soon as they receive messages from the master. So, we consider Redis as a benchmark providing eventual consistency, however, due to the use of the "WAIT" function, in most scenarios (as claimed in [2]), it may provide atomic consistency.

Implementation. We deployed Redis 5 on multiple nodes with Ubuntu 18.04.1 LTS or 20.04 LTS. We implement two variants of Redis, with and without the WAIT command during a write operation, i.e., Redis_W and Redis, respectively. For the Redis_W, we specified the number of waiting write acknowledgments with a majority, i.e., $n/2+1$, to match the ABD algorithm. To send read and write requests we created a script using the Redis-driver Python library. First, the script creates a connection to Redis, giving the IP and port of the master node. Once connected to Redis, the client can write and read with Redis command functions, *set* and *get* respectively. A writer assigns a file's byte string value to the Redis key; it uses the file's id as the key, while a reader gets the value giving the file's id. We note that the number of reader clients can dynamically increase or decrease. However, if the Master crashes, the writes will be blocked, as the replica nodes are read only, until a new replica becomes the new master; with this respect, reconfiguration in Redis is blocking.

3 Experimental Evaluation

In this section we provide a description of our experiments and the results we have obtained in this study. Section 3.1 presents the setup of the distributed system we considered and the tools we used for the experiment deployment. Section 3.2 presents the different scenarios we examined and the purpose of each scenario. We conclude with our results and their analysis in Sect. 3.3. The collected data are available in [3], in case one would like to validate our analysis.

3.1 Experimentation Setup

Our main goal was to conduct real-life experiments, exposed to the perturbations, delays, and uncertainty of network communication. We picked devices both in the EU and the USA, thus, examining the impact of long (cross-Atlantic) communication on the performance of each algorithm. We used two main tools to deploy and execute our experiments: (*i*) jFed [18], and (*ii*) Ansible [5].

Experiment Deployment. jFed is a GUI tool that was developed within the Fed4FIRE+ project and was used to get access and reserve virtual and physical machines in various experimental testbeds. Through the tool we were able to define our node deployment strategy, and specify the connectivity between the reserved nodes, their external interfaces, the resources and the OS image to use, and launch those machines in their respective testbeds, *for all algorithms*.

We used machines from four different testbeds (in the EU and the USA), that are supported by JFed: (*i*) imec Virtual Wall 1/2 [26] (Belgium – EU), (*ii*) Cloudlab [8] (Utah – USA), (*iii*) InstaGENI [17] (NYU, UCLA, and Utdallas – USA) and (*iv*) Grid5000 [15] (France – EU). In total, we used 39 nodes, where the InstaGENI ones are XEN VMs with Ubuntu 18.04.1 LTS and routable IPs, and the rest are physical machines with Ubuntu 20.04 LTS. Due to the similarity on machine specifications and the high demands in those testbeds we did not use a specific set of spec configuration but rather we were reserving random available nodes for each experiment. A reserved machine can either act as a client or a server in any given experimental run. We avoided having a machine with both roles, preventing giving a communication advantage to clients residing in the same machine with a server. Each server is deployed on a different machine, and clients are all deployed in the remaining machines in a round robin fashion (i.e., a machine may execute multiple client instances). For example, with 10 machines, 4 servers, 6 writers and 6 readers, servers would have been deployed on the first 4 machines and each other machine would contain one writer and one reader.

Experiment Execution. Ansible was mainly used for the execution of the experiments as it is a tool to automate different IT tasks, such as cloud provisioning, configuration management, application deployment, and intra-service orchestration. There are two main steps to run an experiment: (*i*) booting up the client (either writer or reader) and the server nodes, and (*ii*) executing each scenario using Ansible Playbooks, scripts written in the YAML language. The scripts get pushed to target machines, they are executed, and then get removed. In our experiments, one instance node was dedicated as a controller to orchestrate the experiments. For the execution of the experiment, Ansible automated the provision of the executables in each machine, the execution of the operations in the experiment, and the collection of the logs for our analysis.

Operations. In throughput experiments, operations are invoked without any delay (i.e., an operation is invoked once the previous operation by the same client is completed), and the clients perform 1000 operations each. For all other experiments we use a stochastic invocation scheme: each client waits a **random interval** each time it terminates an operation and before invoking the next one. Reads and writes are scheduled at a random interval between $[1 \ldots 3]$ s. In total, each writer performs 50 writes and each reader 50 reads. Each reconfigurer invokes one operation every 15 s and performs a total of 15 reconfigurations.

Performance Metric. The performance of the algorithms is measured in terms of the time it takes for their operations to terminate. Thus, for each algorithm, we measure the *average operation latency*, starting at the invocation to the response, and taking into account both the communication as well as the computation overhead. Notice that the operation latency is computed as the average of all clients' average operation latencies. Note that in the case of Cassandra, we omitted to account some "unsuccessful operations", i.e., operations where the client invoking them did not receive replies from a majority of servers.

3.2 Scenarios

Scenarios aim to capture the performance of the algorithms in the three performance parameters (tests) we mentioned in Sect. 1. Our scenarios are:

Scalability Test – Participation (All Algorithms). This scenario is constructed to compare the read and write latencies of the algorithms, as the number of the service participants increases. We varied the number of readers $|R|$ from 5 to 250 and the number of writers $|W|$ from 5 to 20. The number of servers $|S|$ is set to two different values, 3 and 11. To reduce the amount of combinations, we fixed the number of writers to 5 when testing all possible values of readers, and the readers to 5 when testing all possible combinations of writers. The size of the object is 1 MB. We used a different parity for ARES-EC, m, based on the number of servers used: m is set to $m = 1$ for $|S| = 3$ and $m = 5$ for $|S| = 11$.

Stress Test – Topology (All Algorithms). This scenario aims to measure how the performance of the algorithms is affected under different topologies and server participation. In this case we measure the throughput (average number of operations per second) of each algorithm. To avoid any delays due to operation contention, we chose to use 2 clients (1 reader and 1 writer), the minimum number of servers to form a majority, i.e. 3, and a simple object of 32 B. As we deployed machines on both EU and USA, our servers are split in such a way to either force all of them or their majority to be in a single continent. In particular, the 3 servers selected based the following topologies: $0E+3U$, $1E+2U$, $2E+1U$, $3E+0U$, where xY means that x servers are deployed in Y continent for $E = EU$ and $U = USA$. Similarly we deployed the clients either close (i.e., to the same continent) or away from the server majority. Last, we tested the throughput of the algorithms when the number of servers is growing from 3 to 15. In this case, for every server deployed in EU, we deployed 2 servers in the USA.

Stress Test – Object Size (All Algorithms). This scenario is made to evaluate how the read and write latencies are affected by the size of the shared object. The file size doubled from 64 kB to 8 MB. The number of servers is fixed to 11. The number of writers and the number of readers is fixed to 5. For ARES, there are two separated runs, one for ARES-ABD and one for ARES-EC. The parity value of ARES-EC is set to $m = 5$, and thus the fragmentation parameter is $k = 6$. The quorum size of the ARES-EC is $\left\lceil \frac{|S|+k}{2} \right\rceil = \left\lceil \frac{11+6}{2} \right\rceil = 9$, while the quorum size of ARES-ABD is $\left\lfloor \frac{|S|}{2} \right\rfloor + 1 = \left\lfloor \frac{11}{2} \right\rfloor + 1 = 6$. For Cassandra, we set the consistency level (CL) to the majority, i.e., 6. The writers of Redis_W also wait for a majority (6) servers to reply.

Stress Test – Fragmentation Parameter k (Only ARES-EC). This scenario applies only to ARES-EC 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 the object used is 4 MB.

Fault-Tolerance Test – Node Crashes (Only ARES). In this scenario, we introduced server fail-crashes in the ARES algorithm to verify the fault-tolerance guarantees and the responsiveness of the system, *especially with respect to reconfigurations*. The number of servers $|S|$ is set to 11 with $m = 5$. The number of writers and readers are fixed to 5 and 15, respectively. The size of the file used is 1 MB. We execute 2 crashes during each experimental run, server $s0$ crashes 100 s within the experiment and $s3$ crashes 200 s after. Both failed servers are from the imec Virtual Wall 2 testbed (EU), since we observed that they are included in the most quorum replies. We assign a unique id to each quorum. However, the quorum of each DAP differs in size. The size of each quorum (majority) in ARES-ABD is 6, while the quorum size of ARES-EC is 9. In total, ARES-ABD has 462 quorums and ARES-EC has 55. For ease of visualization, we categorize the quorums of the two DAPs into three groups: (i) one which includes all quorums, (*ii*) one which excludes quorums involving $s0$; and (*iii*) one which excludes quorums involving either $s0$ or $s3$. During the same scenario we tested the reconfiguration ability of the algorithm. In particular, we varied the number of reconfigurers with values in $\{1, 3, 5\}$ and each reconfiguration was switching between the two DAPs.

3.3 Experimental Results

Our analytical results aim to expose how a strongly consistent, reconfigurable service like ARES, compares in performance with the two commercial storages of our choice, namely Cassandra and Redis. Moreover, it helps us identify bottlenecks and shortcomings of ARES for future optimizations, and, in some scenarios, we demonstrate the ability of ARES to utilize erasure-coding and to cope with failures and dynamic reconfiguration.

Table 1 provides a comprehensive list of the variables we used in our scenarios. Experiments were conducted for a selection of those parameters. In this section we highlight some representative outcomes in each scenario. *More results may be found in the website of the project*[1] *presented in interactive plots where the user may choose the parameters to apply.* The results shown are compiled as averages over 3 samples per each scenario and 5 samples for the topology scenario.

Table 1. Experimental variables

Variable	Possible values	Description
Topology	{ *0E+3U, 1E+2U, 2E+1U, 3E+0U* }	Distribution of servers in EU and US. For the scenarios with more than 3 servers we use two servers in US for every server in EU
ClientContinent	{ *EU, US* }	Location of the clients (for throughput scenario)
$\mathcal{S}$	{ 3, 5, 7, 9, 11 }	The number of servers
$\mathcal{W}$	{ 0, 1, 5, 10, 15, 20 }	The number of writers
$\mathcal{R}$	{ 0, 1, 5, 15, 50, 100, 150, 250 }	The number of readers
$\mathcal{G}$	{ 0, 1, 3, 5 }	The number of reconfigurers
k	{ 1, 2, 3, 4, 5, 6, 7, 8, 9 }	Erasure-coding data fragments
fsize	{ 64 kB, 128 kB, 256 kB, 512 kB, 1 MB, 2 MB, 4 MB, 8 MB }	The size of the file (object)
Recontype	{ *sameDAP, switchingDAP, switchingDAP & andomServers* }	The way the reconfigurers work: (i) reconfiguring to the same DAP, (ii) reconfiguring the DAP alternately, (iii) reconfiguring the DAP alternatively and servers randomly

[1] https://projects.algolysis.com/ares-ngi/results/.

Scalability Tests. Some of the results obtained while increasing the number of participants in the system appear in Figs. 2, 3 and 4. At a first glance, Cassandra seems to struggle to keep up as the readers grow in all cases, while Redis_W does not seem to be affected. Similar observation can be made for the two ABD based algorithms (ABD and ARES-ABD) as they remain at low levels as $|\mathcal{R}|$ increases. ARES-EC exposes an interesting behavior as it is the worst performing algorithm when few servers are used, and becomes faster when more servers are deployed. This can be seen in Figs. 3 and 4. The more the servers the more the encoded elements to be distributed and the bigger can be the fragmentation parameter k. Thus, each object fragment becomes smaller, resulting in tremendous benefits on the communication delays. Worth observing is that the latency of the write operation of ARES-EC matches the one of Redis_W when $|\mathcal{S}| = 11$.

Similar findings can be seen as the number of writers $|\mathcal{W}|$ grows. Cassandra has the larger write latency despite the fact that it shows a more stable behavior, and the read latency of ARES-EC is the worst when $|\mathcal{S}| = 3$.

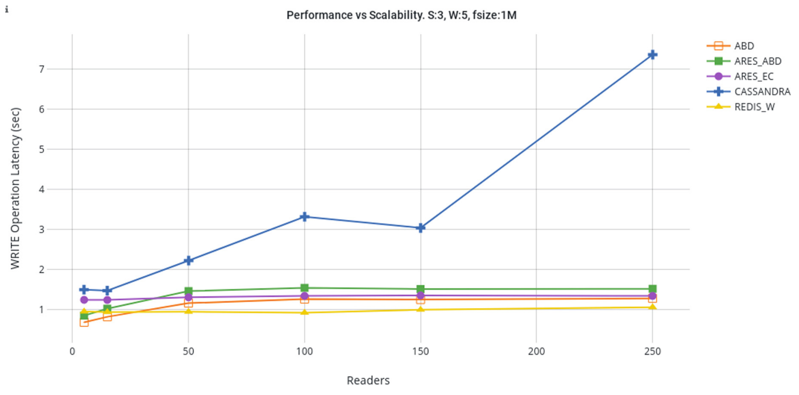


Fig. 2. Readers scalability vs write latency, $|\mathcal{S}| = 3$.

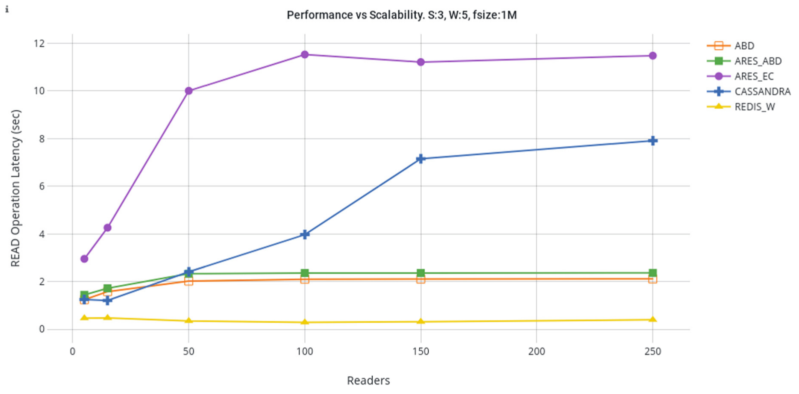


Fig. 3. Readers scalability vs read latency, $|\mathcal{S}| = 3$.

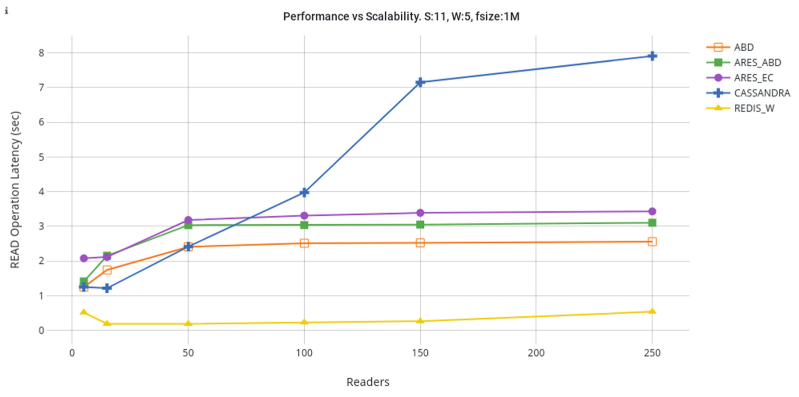
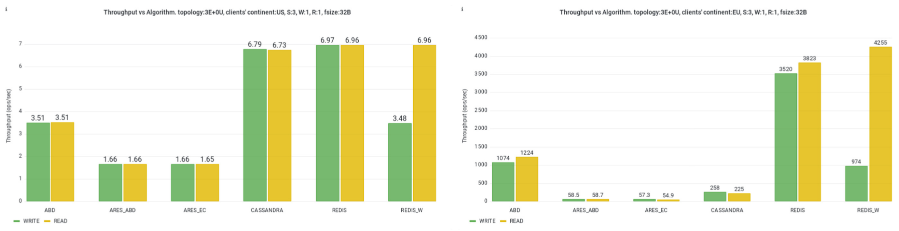


Fig. 4. Readers scalability vs read latency, $|\mathcal{S}| = 11$.



(a) (b)

Fig. 5. Throughput vs algorithm. Topology: $3E + 0U$

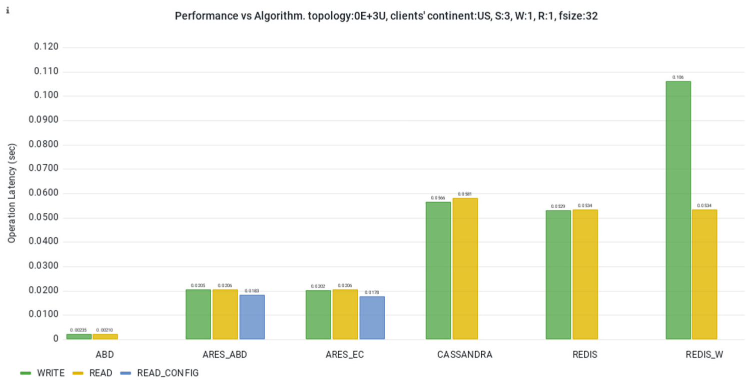


Fig. 6. Performance vs algorithm. Topology: $0E + 3U$ (Color figure online)

Stress Tests – Topology. Some results from these experiments appear in Figs. 5 and 6. Overall the topology played a major role on the performance, and in particular throughput, of all the algorithms we studied. All of the algorithms (including the ADSM algorithms we implemented, i.e., ABD, ARES-ABD, and

ARES-EC), achieve their maximum read and write throughput when the servers and the clients are deployed in the same continent.

For the ADSM algorithms, there appears to be no difference when the experiment contains non-concurrent or concurrent operations. The small $fsize$ (32 B), amplified the impact of the stable overhead of read-config operations, and they constitute a significant percentage of the total operation latency (see blue bar in Fig. 6). From the same figure we interestingly observe that the setup where all servers and clients are deployed in the USA, favored the ADSM algorithms over both Cassandra and Redis.

On the other hand, Cassandra shows different behavior. It achieves the maximum read throughput when both servers and clients are deployed in the EU. It demonstrates a small lead over the ADSM algorithms in most cases on both operations. However, it shows some performance degradation when write and read operations are invoked concurrently.

Finally, Redis and Redis_W outperform the rest of the algorithms in most scenarios. Redis shows consistent performance for both reads and writes due to the weaker consistency requirements and thus smaller communication footprint. The impact of the communication overhead is obvious in Redis_W, where the writer waits before completing.

Stress Tests – Object Size. The results for the write performance in these experiments are captured in Fig. 7. We observe that the write latencies of all operations, except ARES-EC and Redis_W, grow significantly, as the $fsize$ increases. The fragmentation applied by the ARES-EC benefits its write operations, which follow a slower increasing curve like the Redis_W. The write latencies of all other algorithms are close to each other. Results show that the read operations of ARES-EC suffer the most delays until 4 MB. The first phase of the read operation does decoding, which is slower than the first phase of the write, which simply finds the maximum tag, contributed to this overhead. However, at larger file sizes (8 MB) Cassandra has the slowest read operations. As

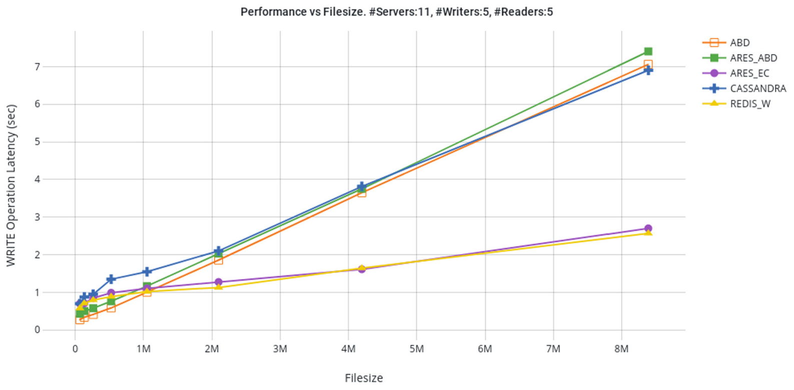


Fig. 7. Filesize results.

expected, the REDIS_W read operations provide the best results, and its write operations with the WAIT command have higher latency compared to the read operations. However, both of them remain at low levels as the $fsize$ increases.

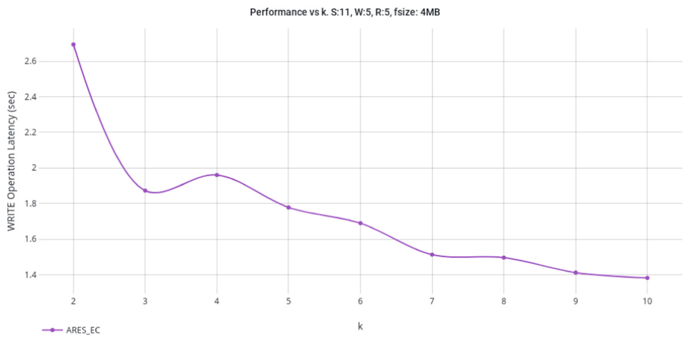


Fig. 8. k scalability results.

Stress Tests – Fragmentation Parameter k. From Fig. 8 we can infer that when smaller k are used, the write and read latencies reach their highest values. In both cases, small k results in the generation of a smaller number of data fragments and thus bigger sizes of the fragments and higher redundancy. For example, we can see that for $RS(11, 7)$ and $RS(11, 6)$ we have the same size of quorum, equal to 9, whereas the latter has more redundant information. As a result, with a higher number of m (i.e., smaller k) we achieve higher levels of fault-tolerance. The write latency seems to be less affected by the value of k since the write operation does only encoding, and not decoding, while the read operation does both. In conclusion, there appears to be a trade-off between operation latency and fault-tolerance in the system: the further increase of k (and thus lower fault-tolerance), the smaller the latency of read/write operations.

Fault-Tolerance. Figure 9 shows to which quorum group (0, 1, or 2) the responding servers belong when only 1 reconfigurer exists. That is, Fig. 9 shows

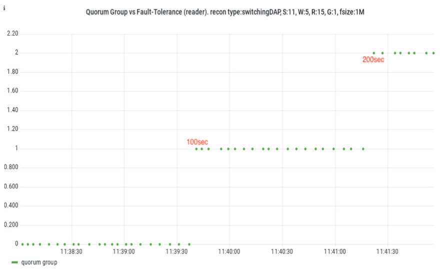


Fig. 9. Quorum replies to reader6.

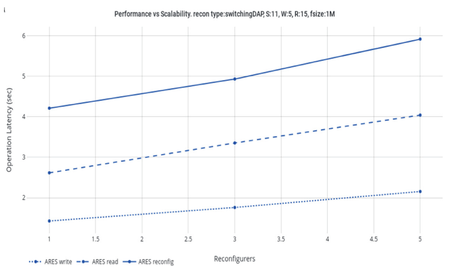


Fig. 10. Reconfiguring DAP alternately and 2 server fails.

the quorum group that sends to reader6 every 1 s interval. Until the first 100 s of each operation, quorum group is 0, e.g., all quorums were active. From that moment on, the clients receive responses only from group 1, e.g., quorums excluded server0. With the second kill, after 200 s, only the quorums included in group 2 remain active. Figure 10 shows the read, write, and reconfig operation latency as the number of reconfigurers increases. During each experiment, the two server failures took place, but our system kept running without interruptions.

4 Conclusions

As a general finding, achieving strong consistency is more costly than providing weaker semantics as we experienced with Redis and Redis_W. However, the performance gap is not prohibitively large and future optimizations of ARES may close it enough so as to substantiate trading performance for consistency. Compared to the atomic version of Cassandra, ADSM algorithms seem to scale better, but lack behind in the throughput when dealing with small objects. Both approaches seem to be affected by the object size, but ARES-EC suggests that fragmentation may be the solution to this problem. Finally, we demonstrated that ARES may handle efficiently failures in the system, and reconfiguring from one DAP to another without service interruptions. Also, by examining the fragmentation parameter, we exposed trade-offs between operation latency and fault-tolerance in the system: the further increase of the parity (and thus higher fault-tolerance), the larger the latency.

ARES, an algorithm that always offers provable guarantees, competes closely and in many cases outperforms existing DSS solutions (even when offering weaker consistency guarantees). It would be interesting to study how optimizations may improve the performance of ARES. For example, fragmentation techniques as presented in [12] may have a positive impact on the performance of the algorithm.

References

1. Cassandra. https://cassandra.apache.org/_/index.html
2. Redis. https://redis.io
3. Data repository. https://github.com/nicolaoun/ngiatlantic-public-data
4. Aguilera, M.K., Keidar, I., Malkhi, D., Shraer, A.: Dynamic atomic storage without consensus. J. ACM **58**(2), 7:1–7:32 (2011)
5. Ansible. https://www.ansible.com/overview/how-ansible-works
6. Attiya, H.: Robust simulation of shared memory: 20 years after. Bull. EATCS **100**, 99–114 (2010)
7. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42**(1), 124–142 (1995)
8. CloudlabUtah. Cloudlab Utah. https://www.cloudlab.us
9. Dutta, P., Guerraoui, R., Levy, R., Chakraborty, A.: How fast can a distributed atomic read be? In: Proceedings of PODC, pp. 236–245 (2004)

10. Georgiou, C., Nicolaou, N., Shvartsman, A.: Fault-tolerant semifast implementations of atomic read/write registers. JPDC **69**(1), 62–79 (2009)
11. Georgiou, C., Hadjistasi, T., Nicolaou, N., Schwarzmann, A.A.: Implementing three exchange read operations for distributed atomic storage. JPDC **163**, 97–113 (2022)
12. Georgiou, C., Nicolaou, N., Trigeorgi, A.: Fragmented ARES: dynamic storage for large objects. In: Proceedings of DISC (2022, To appear). arXiv:2201.13292
13. Gilbert, S., Lynch, N.A., Shvartsman, A.A.: RAMBO: a robust, reconfigurable atomic memory service for dynamic networks. Dist. Comp. **23**(4), 225–272 (2010)
14. Gramoli, V., Nicolaou, N., Schwarzmann, A.A.: Consistent Distributed Storage. Synthesis Lectures on DC Theory. Morgan & Claypool Publishers, San Rafael (2021)
15. GRID5000. https://www.grid5000.fr/w/Grid5000:Home
16. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. ACM TOPLAS **12**(3), 463–492 (1990)
17. InstaGENI. https://groups.geni.net/geni/wiki/GeniAggregate
18. jFed. https://jfed.ilabt.imec.be
19. Lamport, L.: On interprocess communication, parts I and II. Distrib. Comput. **1**(2), 77–101 (1986)
20. Lynch, N., Shvartsman, A.: Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In: Proceedings of FTCS, pp. 272–281 (1997)
21. Nicolaou, N., et al.: ARES: adaptive, reconfigurable, erasure coded, atomic storage. ACM Trans. Storage (TOS) (2022, To appear). arXiv:1805.03727
22. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proceedings of USENIX ATC, pp. 305–320 (2014)
23. PyEClib. https://github.com/openstack/pyeclib
24. PySyncObj. https://github.com/bakwc/PySyncObj
25. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. J. Soc. Ind. Appl. Math. **8**, 300–304 (1960)
26. VirtualWall. https://doc.ilabt.imec.be/ilabt/virtualwall/
27. ZeroMQ. https://zeromq.org