Dependable Distributed Shared Memory Suitable for Large, Strongly Consistent Objects

Candidate

Andria Trigeorgi

Supervisor

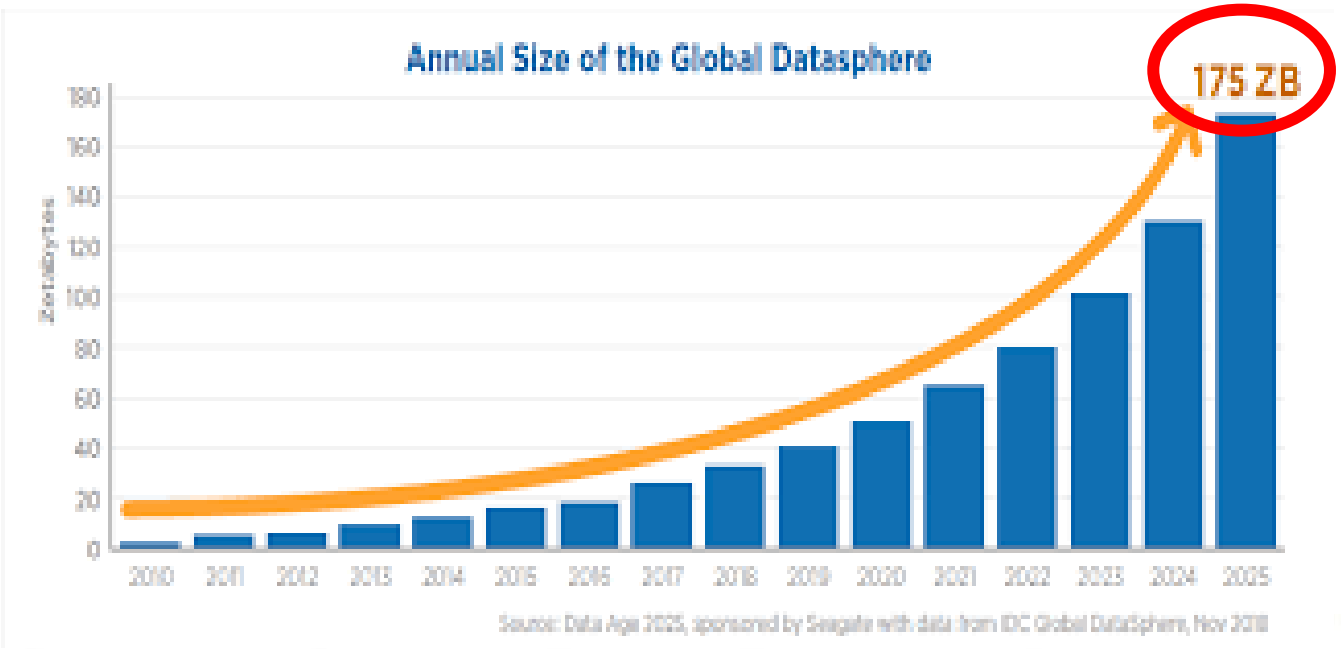
Chryssis Georgiou

PhD Defence

in the Department of Computer Science

University of Cyprus

Growth of the Digital Universe



"Growth of global data , rising from 26 ZB in 2017 to a projected 175 ZB by 2025."
- International Data Corporation (IDC)

"The data in the digital universe doubles every two years."
- EMC Digital Universe

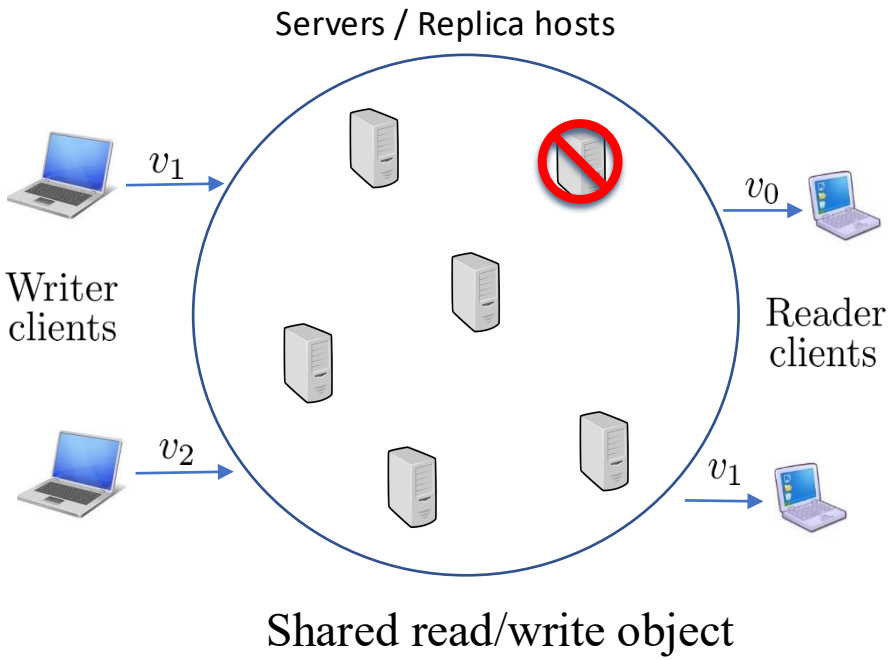
- 1 ZB is 10^{21} bytes = the estimated number of stars in the universe!

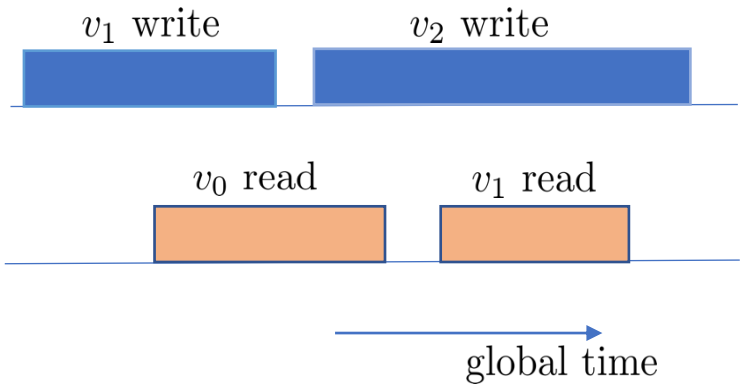
Distributed Storage Systems (DSS)

Commercial Solutions:

- Often provide Weak Consistency guarantees (e.g., Dropbox)
- Limited concurrency (e.g., HDFS - one writer at a time)
- Often rely on centralized solutions to provide strong consistency (e.g., HDFS)
  - Drawback: Performance Bottleneck
- They do not provide rigorously *provable guarantees*

Distributed Shared Memory Emulations (DSMs)





Implementing a fault-tolerant shared object in an asynchronous, message-passing environment:

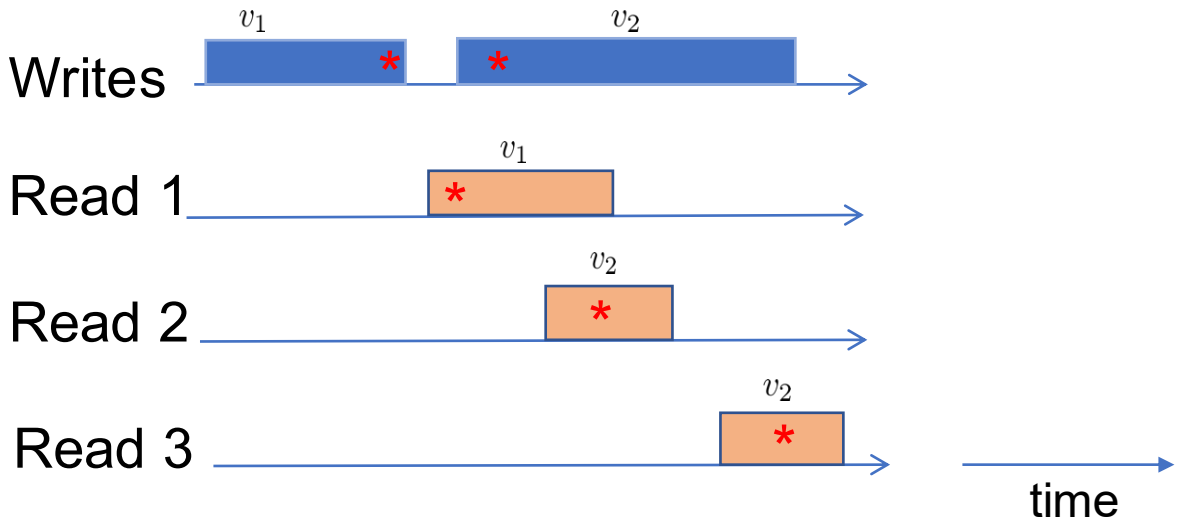
- Availability + Survivability => use redundancy
- Asynchrony + Redundancy => concurrent operations
- Behavior of concurrent operations => consistency semantics
  - Safety, Regularity, **Atomicity** (Atomic DSMs) [Lamport 1986]

L. Lamport,"On Interprocess Communication," Distributed Computing, vol. 1, no. 2, pp. 77–101, 1986.

Atomicity/Linearizability

- Provides the illusion that operations happen in a sequential order
  - a read returns the value of the preceding write
  - a read returns a value at least as recent as that returned by any preceding read

[Herlihy, WIng 1990]



M. Herlihy, J. Wing, "Linearizability: a correctness condition for concurrent objects," ACM TOPLAS 12(3), 463–492 (1990).

Seminal Algorithm - ABD

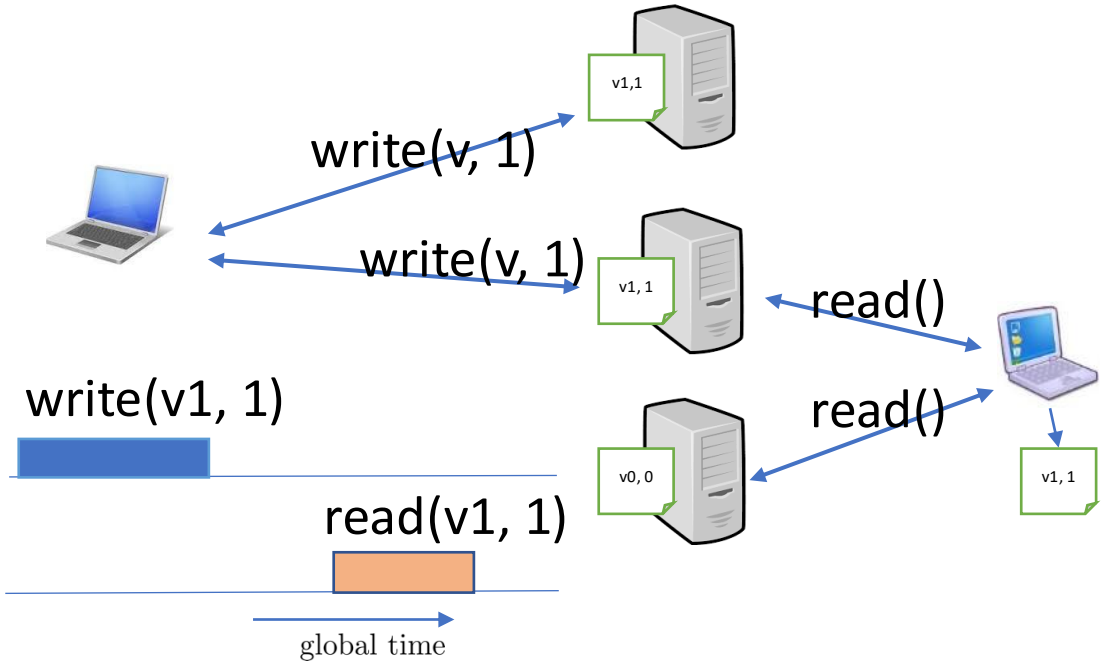
An elegant, intuitive solution that

- uses the power of the majority, and
- assigns logical timestamps to written values for ordering the operations.

[Attiya, Bar-Noy, Dolev 1995]

Dijkstra Prize 2011

- SWMR atomic registers
- S servers, $f < n / 2$
- 1 writer
- R readers



Extended by Lynch and Schwarzmann

in 1997 for MWMR, assigning tags <ts,wid> – **MW-ABD**

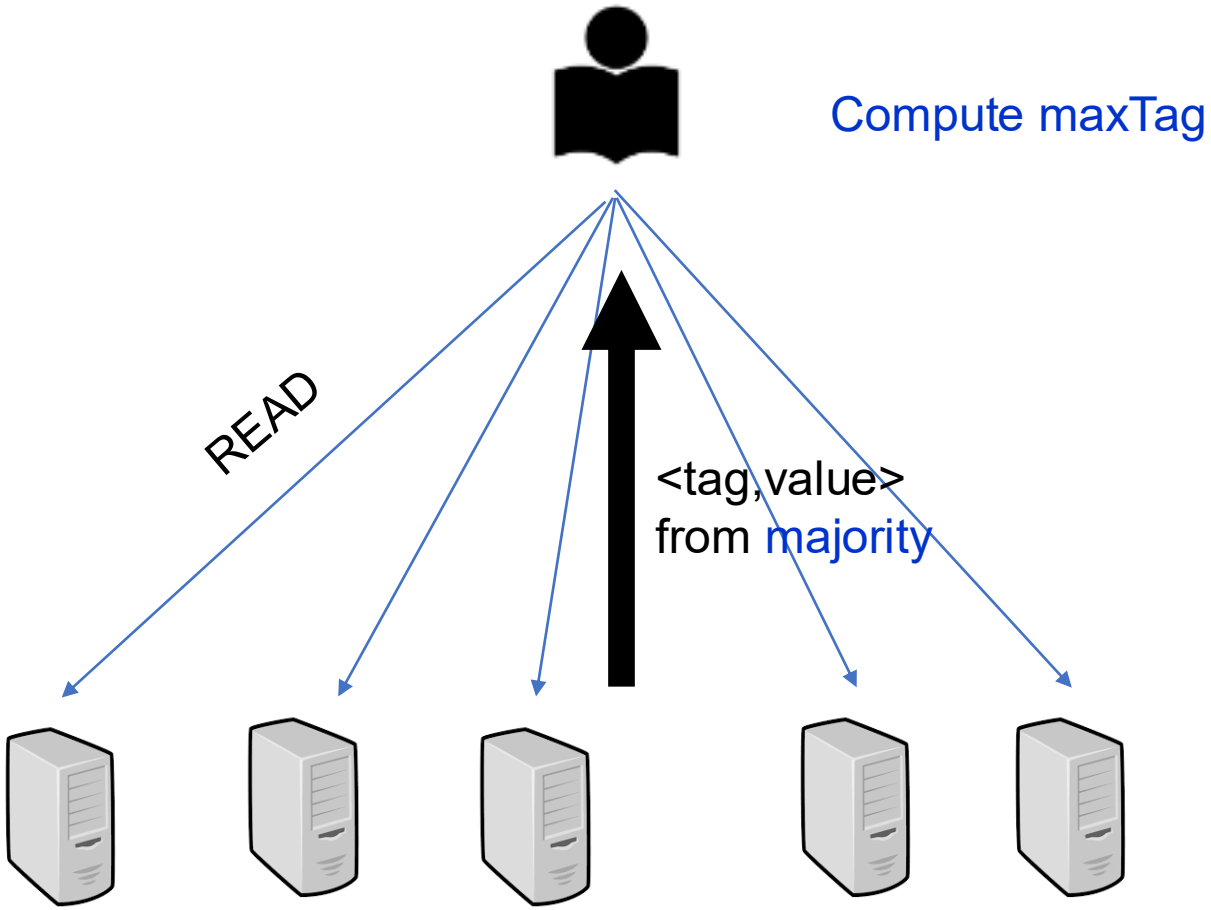
Fault tolerance & *Consistency Guarantees have been* rigorously proved

H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing Memory Robustly in Message-Passing Systems," Journal of the ACM (JACM), vol. 42, no. 1, pp. 124–142, 1995.

N. Lynch, A. Shvartsman.. "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts," In Proc. of FTCS pp. 272–281 (1997).

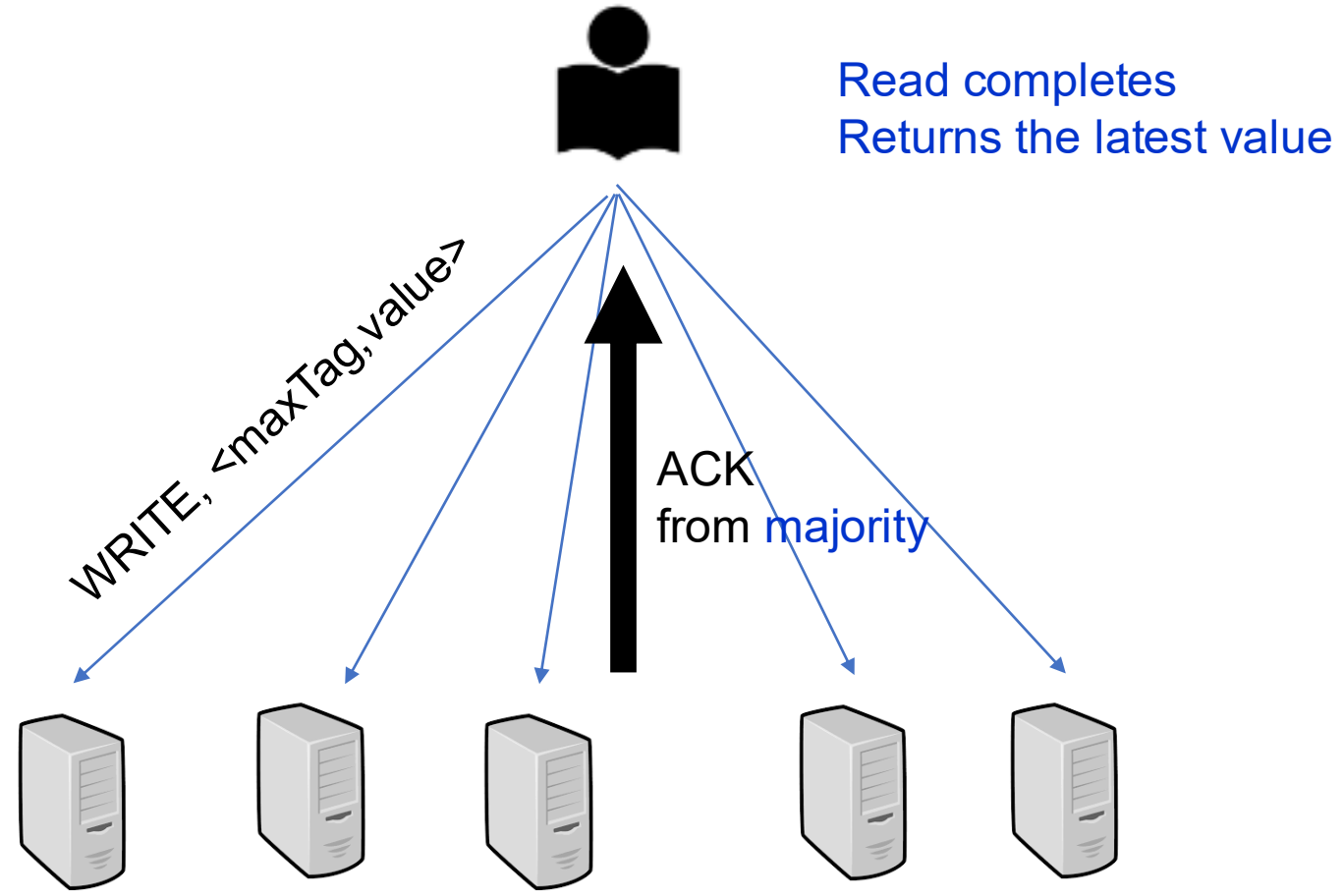
READ Protocol of MW-ABD

Phase1: **Query** - Discover maximum tag and associated value



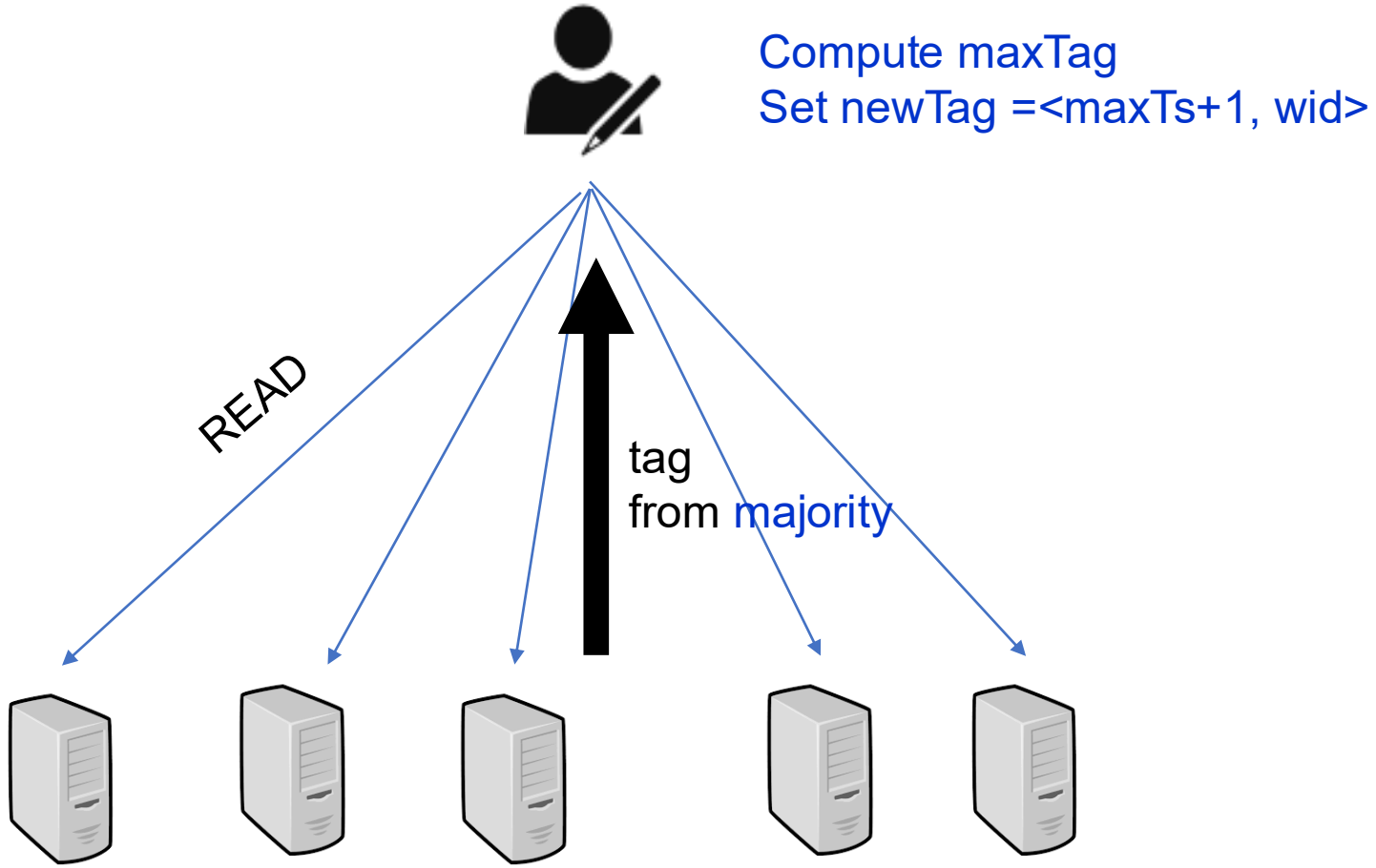
READ Protocol of MW-ABD

Phase 2: **Propagate** <maxTag,value>



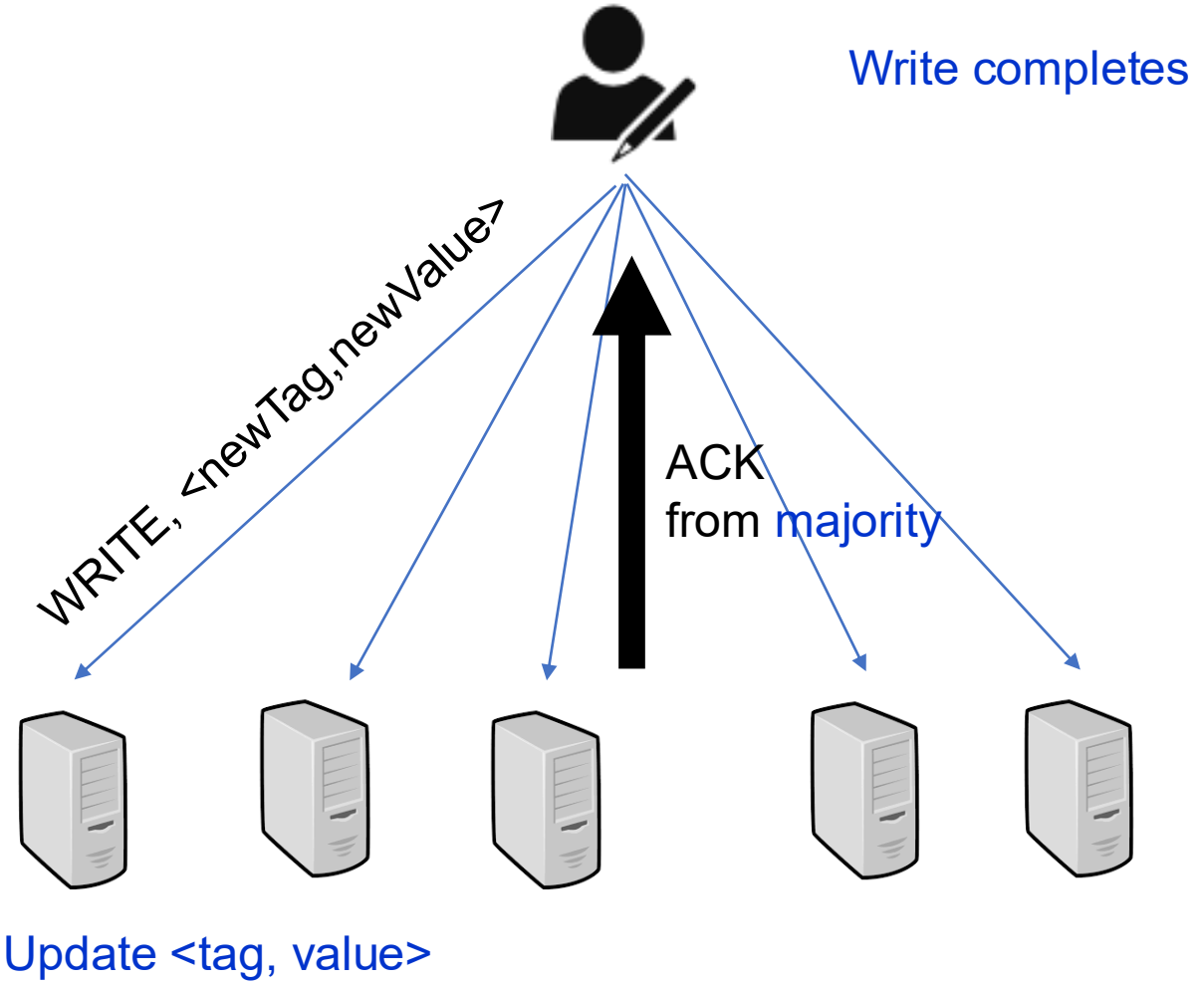
WRITE Protocol of MW-ABD

Phase1: **Query** - Discover maximum tag



WRITE Protocol of MW-ABD

Phase 2: **Write** <newTag, newValue>



Handling Large Objects

- Many DSMs, for both static (fixed servers) and dynamic (remove/add servers) systems, were developed over the years.

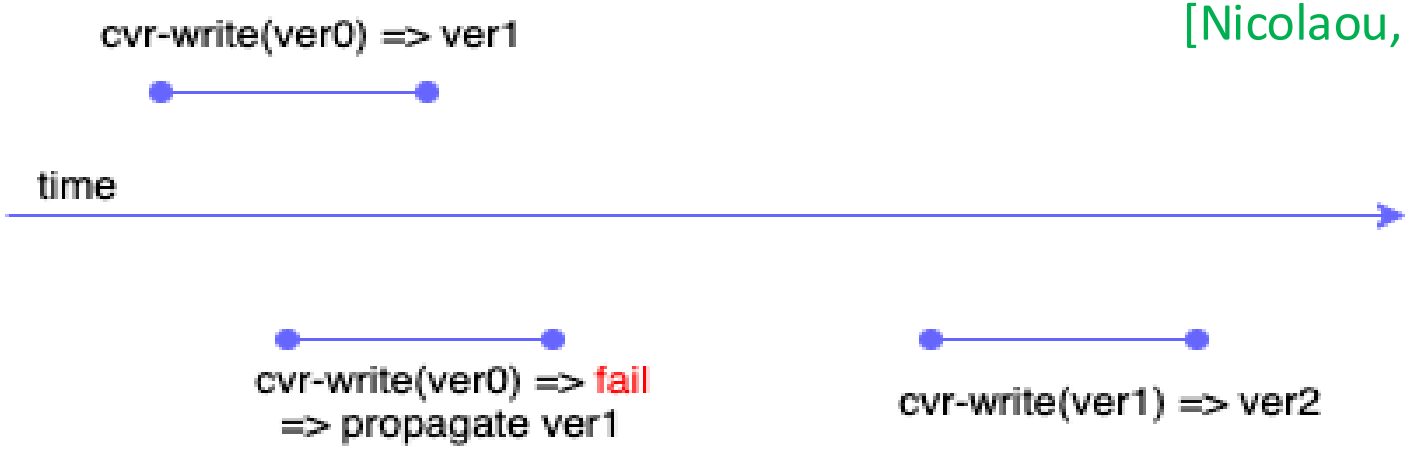
- These protocols are:
  - efficient for small objects
  - expensive solutions that are difficult to implement in an asynchronous, fail prone, message passing environment.

- Limitation 1: No enforced dependence between versions of the object [versioning]
- Limitation 2: Unable to handle write operations working on different parts of a large object [striping]

Versioning and Coverability

- Versioned object: a R/W object where each value written is associated with a version (= tag)
- Coverability is defined over a totally ordered set of versions, and extends linearizability by guaranteeing that:
  - A write on the object succeeds only when the write is associated with the latest version of the object. Otherwise, it becomes a read operation (to obtain the latest version)

[Nicolaou, Anta, Georgiou 2016]

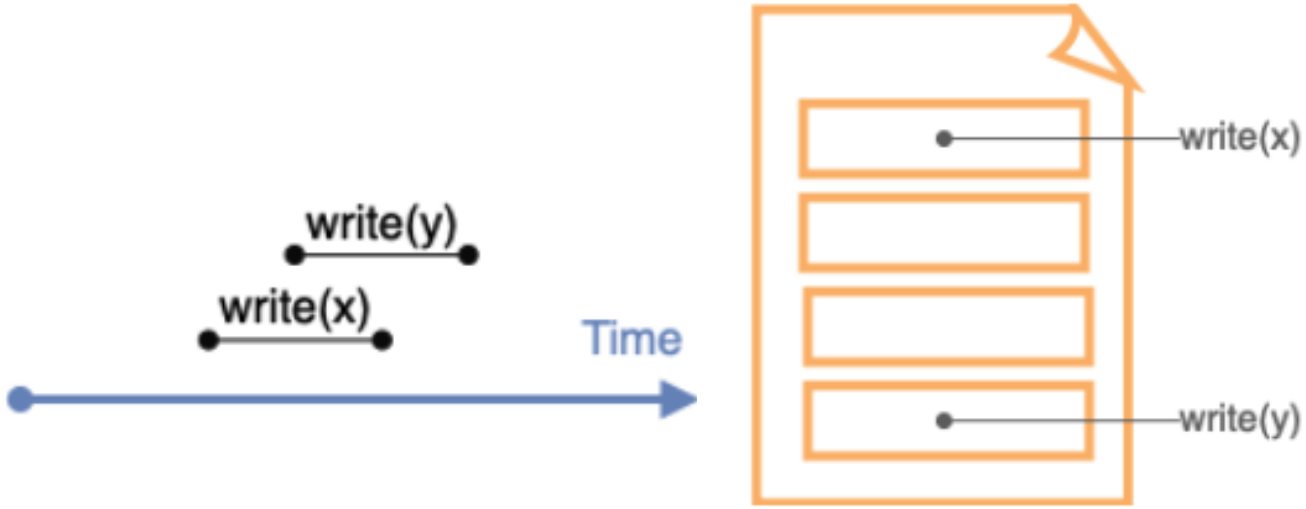


- CoABD: Modified MW-ABD that guarantees coverability

N. Nicolaou, A. Ferna´ndez Anta, and C. Georgiou, "Coverability: Consistent Versioning in Asynchronous, Fail-Prone, Message-Passing Environments," in Proc. of IEEE NCA 2016, IEEE, 2016.

Concurrent Access on Large Objects

- Concurrent write operations may overwrite one another
- In some cases this is unavoidable. But what if two changes take place on different parts of a large file?



Ultimate Objective

The development of an efficient Distributed Shared Memory that provides provable atomic consistency guarantees and high access concurrency at large scale under an asynchronous, crash-prone, distributed and even dynamic environment.

Methodology

1. Implemented the most efficient Atomic Shared Object Algorithm.
2. Specified a Data Fragmentation Strategy.
3. Designed and implemented the framework of our system, and introduce new consistency guarantees.
4. Implemented ARES to introduce a dynamic solution.
5. Combined our Fragmentation Strategy with ARES.
6. Evaluated our implementation against commercial solutions.
7. Deployed and evaluated the system in network testbeds (Emulab, AWS, Fed4FIRE+)
8. Identified any performance bottlenecks using distributed tracing and optimized them.

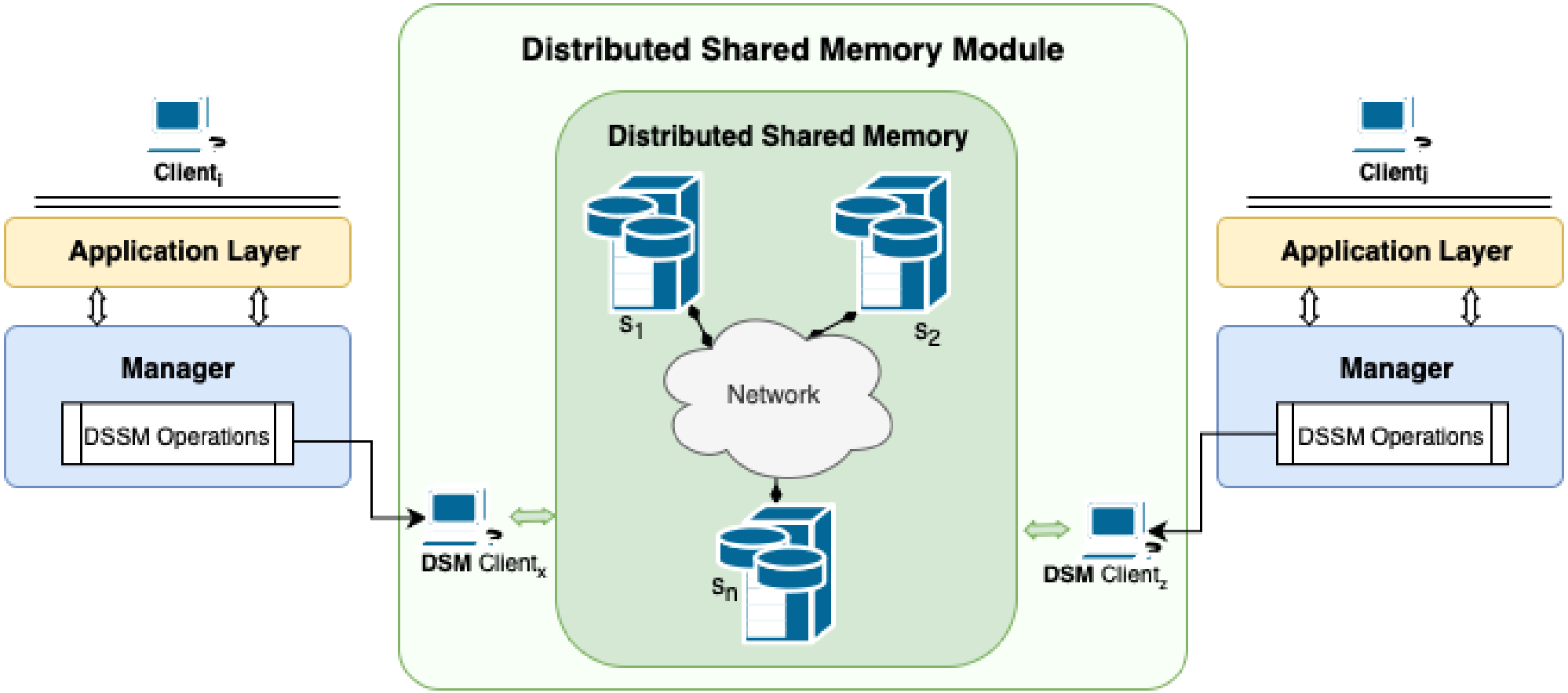
System Settings

To implement an atomic constistent DSM that:

- supports large shared R/W objects
- with two main distinct sets of processes: a set ***C*** of clients and a set ***S*** of servers

  (**Configuration** = the set ***S*** of servers and some additional info)
- Fixed Configuration -> **Static** environment
- Reconfiguration -> **Dynamic** environment
- *C* = a set ***W*** of writers, a set ***R*** of readers, and a set ***G*** of reconfigurers

Basic Arcitectures



Evaluation Setup

Testbeds

Emulab: a network testbed with tunable and controlled environmental parameters.

AMAZON Web Services (AWS) EC2: a web service that provides scalability and performance.

Fed4FIRE+: a federation of testbeds.

Performance Metric

- Average Operation latency of all clients (Communication + Computation Overhead).
- Update/Write Success Ratio

PART Ⅰ - Fragmented Objects: Boosting Concurrency of Shared Large Objects

covering Stages 1-3, 7 in Methodology

Approach and Contribution

- Define concurrent objects: (i) the *block* object, and (ii) the *fragmented* object.
- Define the consistency that the fragmented object provides (*fragmented coverable linearizability*).
- Implement CoBFS, a Framework which implements the fragmented objects.
- Performed an experimental evaluation of CoBFS on Emulab.

Fragmented Objects: Boosting Concurrency of Shared Large Objects

Antonio Fernández Anta [1] Chryssis Georgiou [2]
Theophanis Hadjistasi [3] Nicolas Nicolaou [3] Efstathios Stavrakis [3]
Andria Trigeorgi [2]

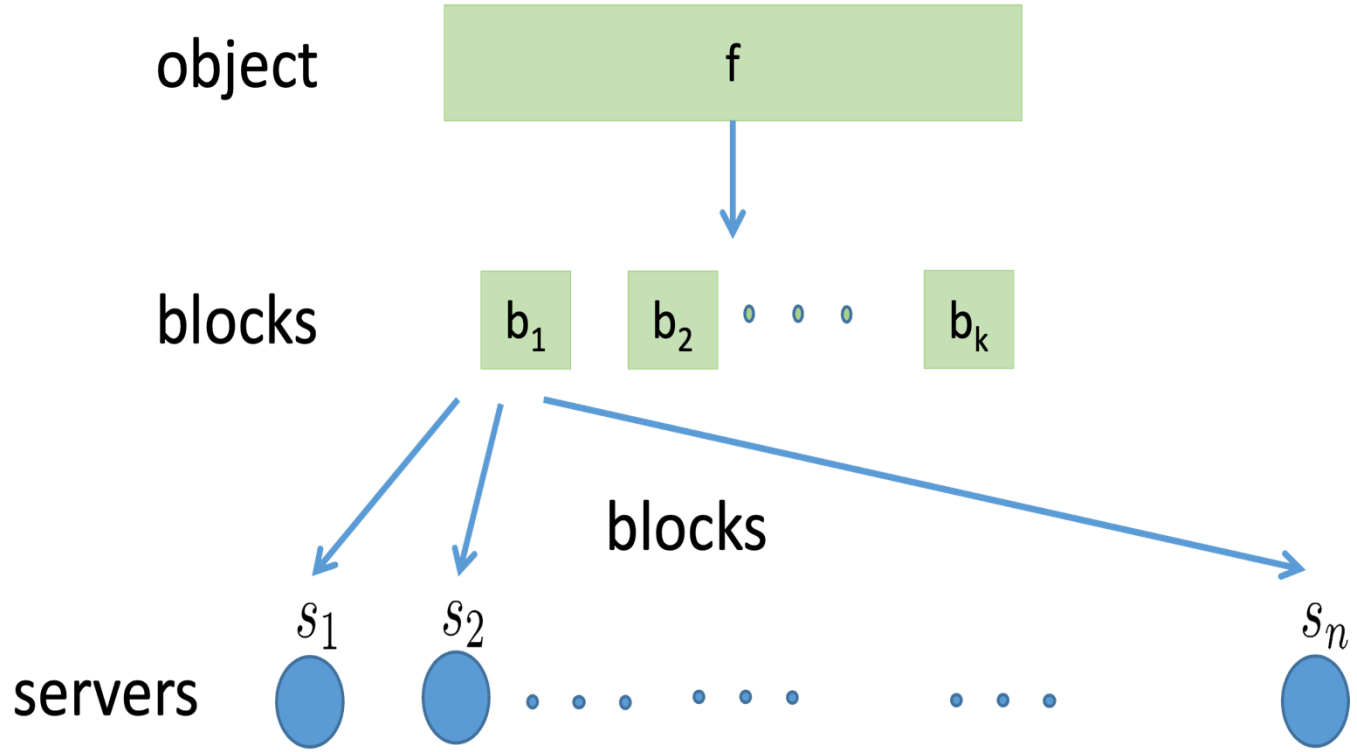
[1] IMDEA Networks Institute, Madrid, Spain
[2] University of Cyprus, Nicosia Cyprus
[3] Algolysis Ltd, Limassol, Cyprus

SIROCCO 2021

Solution: Fragmentation

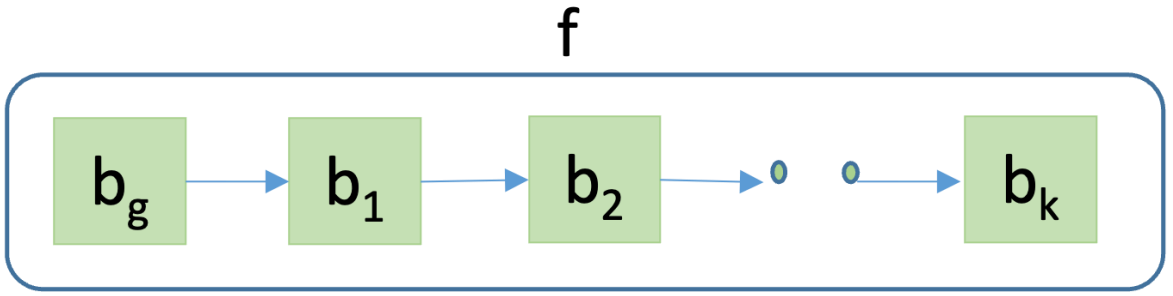
Each object is fragmented into blocks

- Allows big amounts to be distributed all over the servers
- Avoids contention for concurrent accesses to different blocks



Solution: Fragmentation

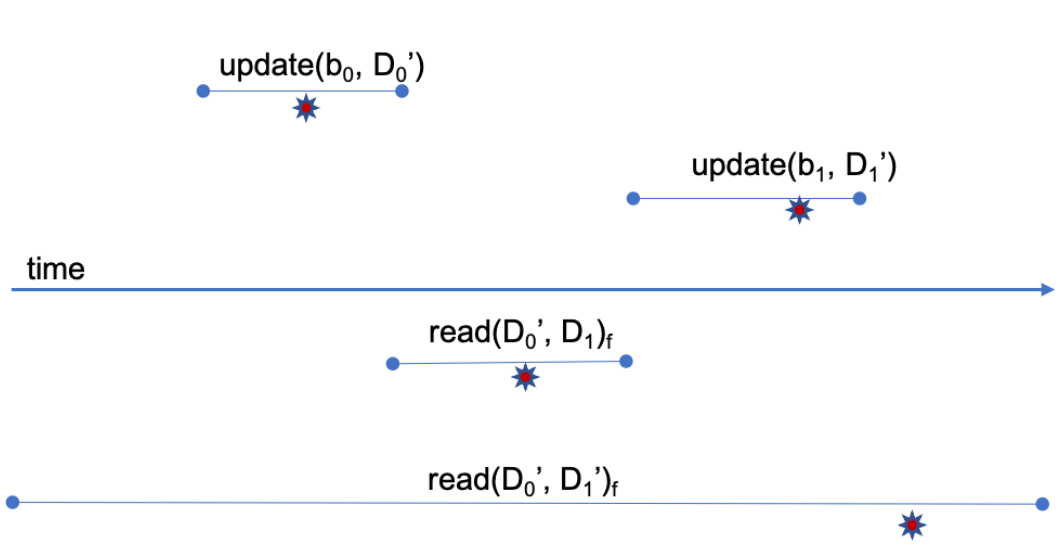
- **Fragmented object**
  - Each *f* is a *list of blocks*
  - Each block has the id of its next block
  - Each block is linearizable and coverable
  - The first block is the genesis block b_{gen}



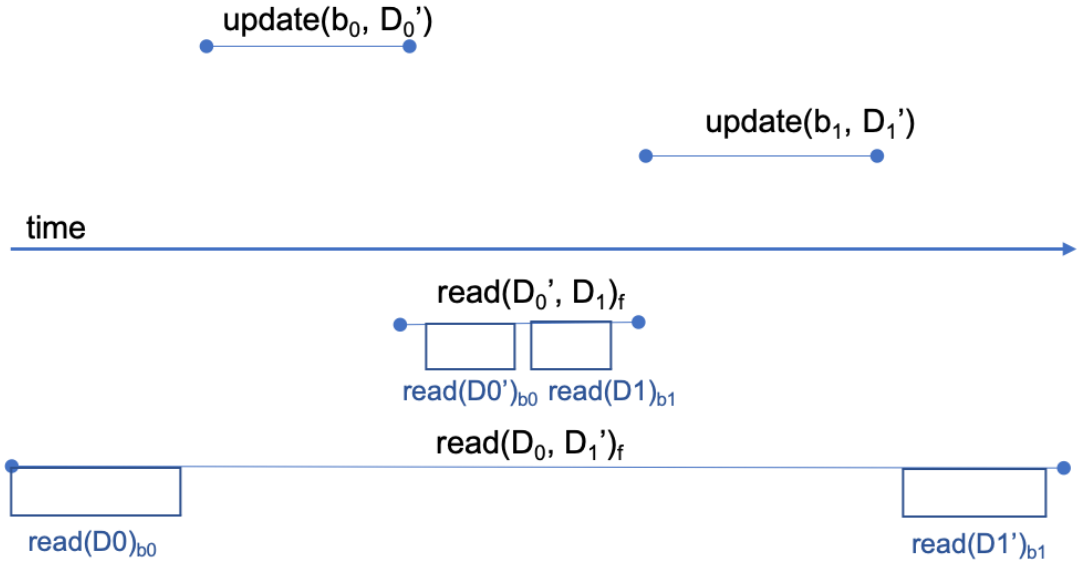
- **Write Operation write(f)**
  - Propagate only modified and new blocks

- **Read Operation read(f)**
  - Start from b_{gen} and read all the blocks

Fragmented Coverable Linearizability

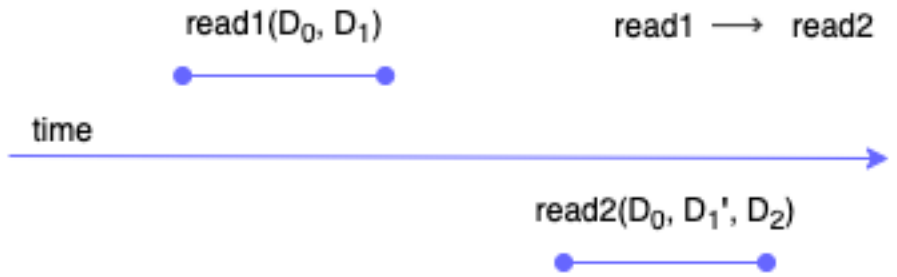


(a) Linearizability on the whole object

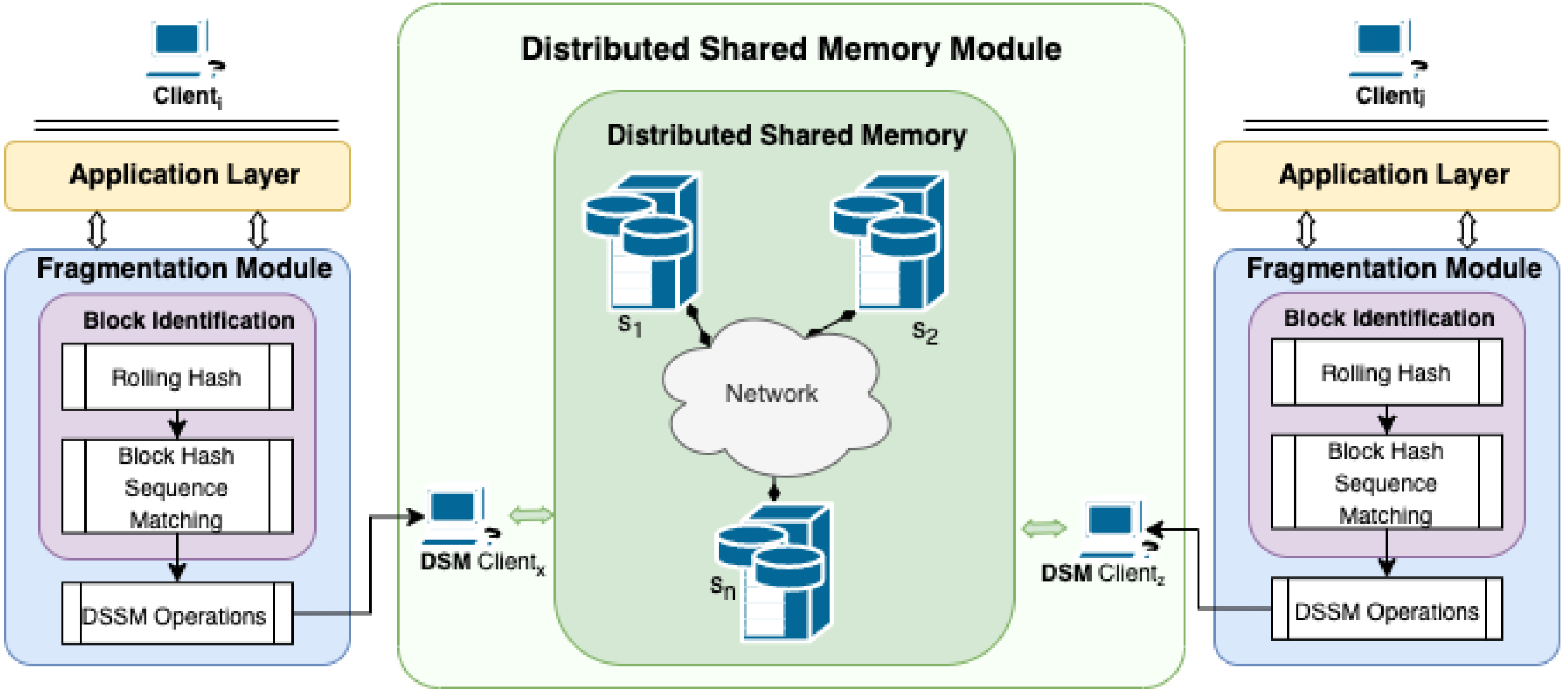


(b) Fragmented Linearizability

Fragmented Coverable Linearizability guarantees that all concurrent operations on different blocks prevail, and only concurrent operations on the same blocks are conflicting.



CoBFS Framework



CoABD-F: Integration of CoABD with CoBFS

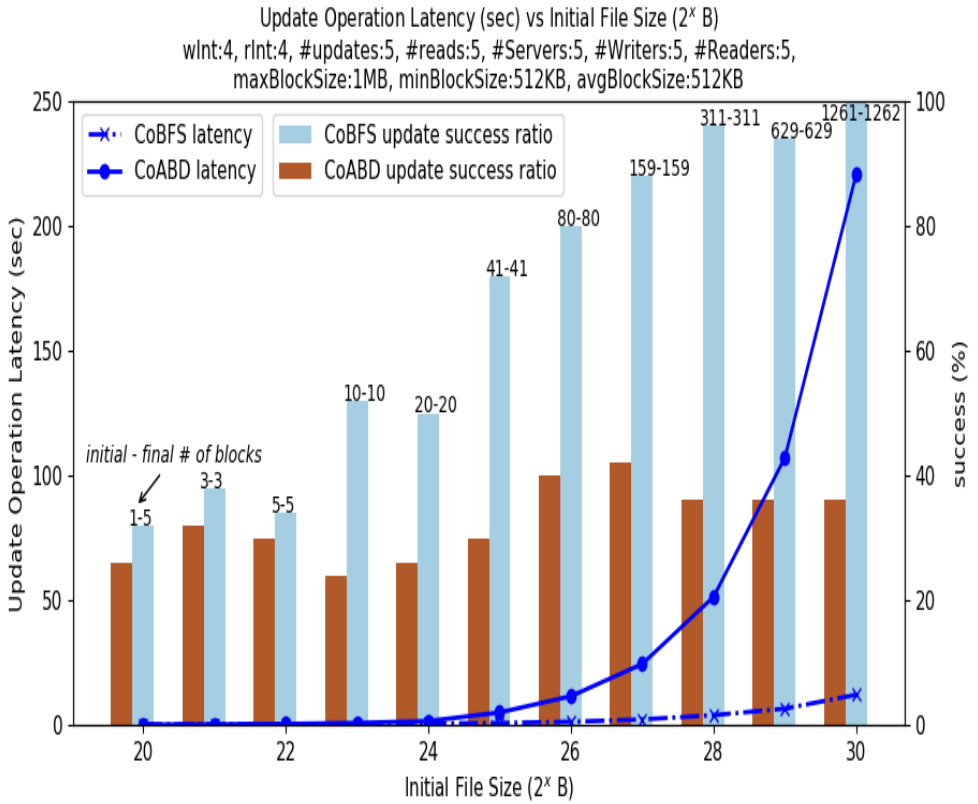
Read Operation

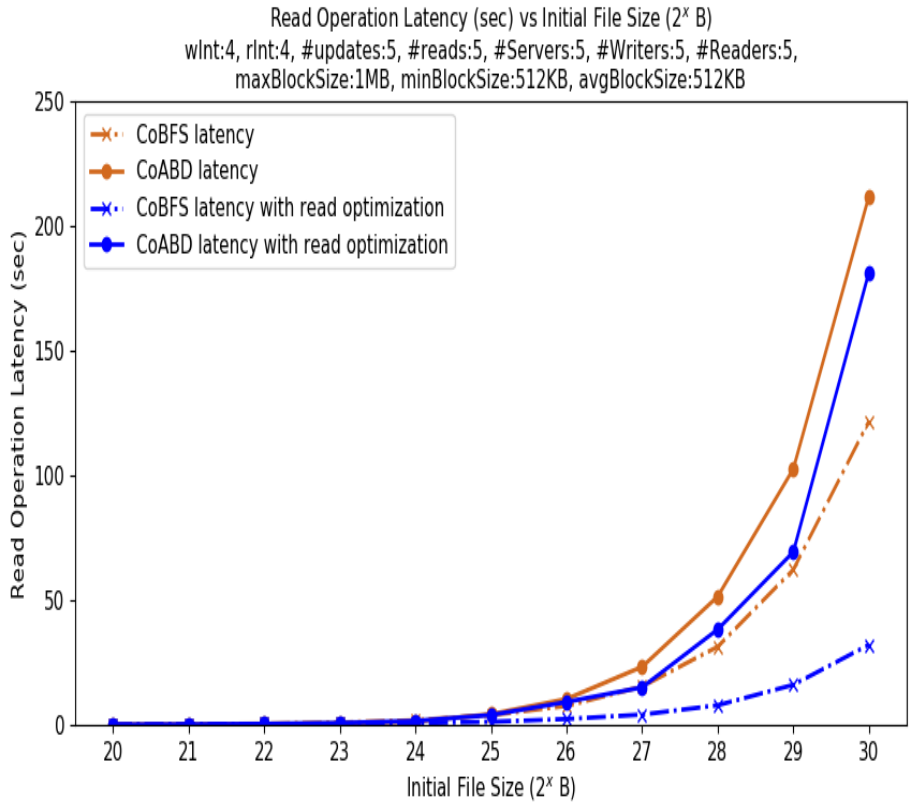
- FM issues dsmm-read for each block
- crv-read executes the read operation on DSMM
- CoABD Optimization: Only blocks that have changed are transferred

Update Operation

- FM divides objects into blocks $<D_0,\ldots,D_k>$ using **Block Identification (BI)**
- FM updates each block with its corresponding data
  - If k=0: dsmm-write is used
  - If k>0: New blocks are created using dsmm-create, modified blocks are written using dsmm-write, and block pointers are updated
  - Block Operations: Sequential creation and write operations for chunks of data
- The block operations are executed using cvr-write on the DSMM

CoABD VS. CoABD-F (Emulab)





File Size:

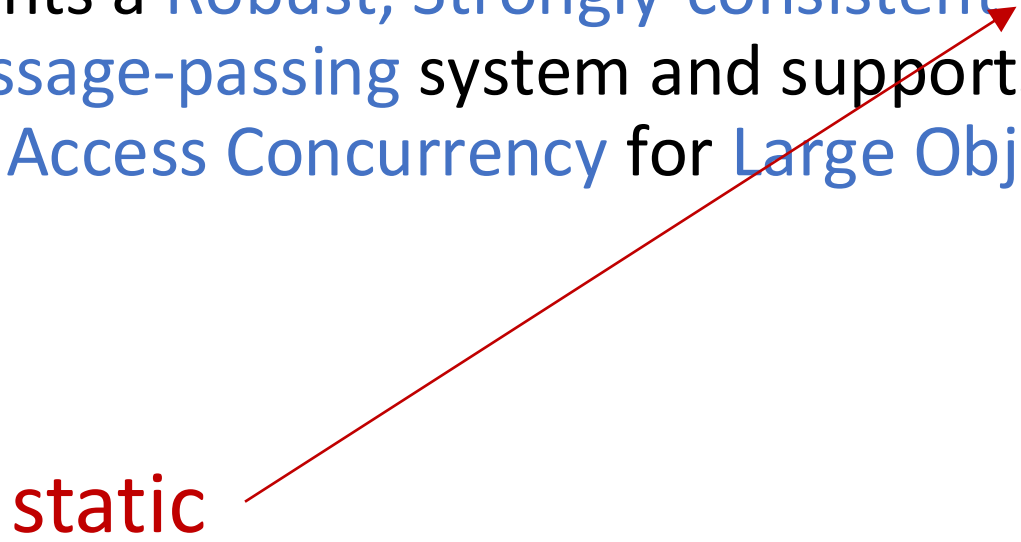
- the update latency of CoABD-F remains at extremely low levels, although the file size increases.
- a read optimization decreases significantly the CoABD-F read latency, since it is more probable for a reader to already have the last version of some blocks.

Overview of Results

- CoABD-F has significantly lower write and read operation latency (especially in larger files)
- For the read operation latency of smaller sizes (1MB), suggest that there is room for improvement
- Trade-off between block size, operation latency and write success rate
- …

Recap and Next Goal

- CoABD-F implements a Robust, Strongly-consistent DSM in an asynchronous message-passing system and supports Versioning, Data Striping, and High Access Concurrency for Large Objects



- Next Goal: Obtain such as DSM for **Dynamic** systems, where servers (replica hosts) change over time, without interrupting the read/write operations or violating data consistency

PART Ⅱ - Implementation and Experimental Evaluation of ARES

covering Stages 4 & 6, 7 in Methodology

Approach and Contribution

- Implement ARES to enable dynamic reconfiguration.
- Performed an experimental evaluation of ARES on Emulab.
- Set up two open-source commercial solutions (Redis and Cassandra) to compare them with ARES.
- Performed experiments in real testbeds (supported by Fed4FIRE+ project), distributed in the European Union (EU) and the USA.

ARES: Adaptive, Reconfigurable, Erasure coded, Atomic Storage

Nicolas Nicolaou [1] Viveck Cadambe [2]
N. Prakash [3] Andria Trigeorgi [4] Kishori M. Konwar [5]
Muriel Medard [5] Nancy Lynch [5]

[1] Algolysis Ltd, Limassol, Cyprus
[2] Pennsylvania State University, US
[3] Intel Corp.
[4] University of Cyprus, Nicosia Cyprus
[5] Massachusetts Institute of Technology, USA

ACM Transactions on Storage

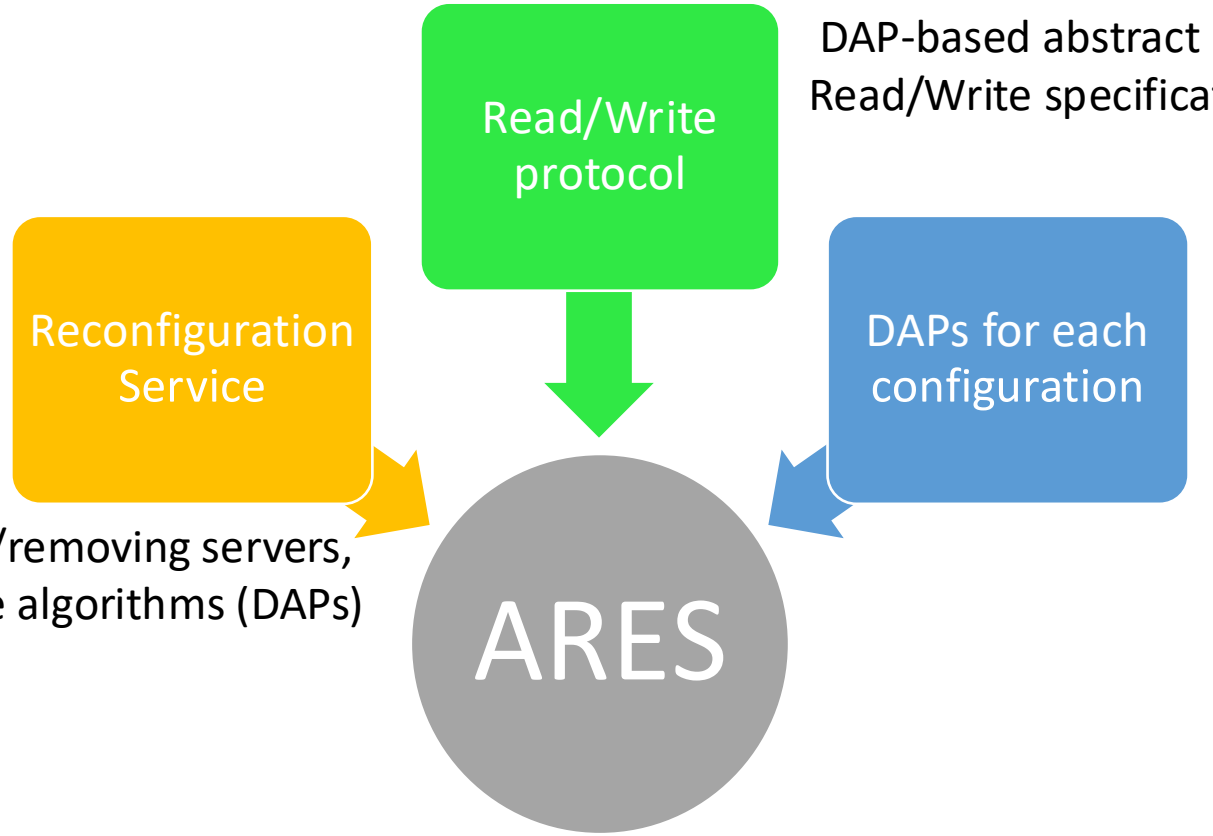
algolysis
algorithmic solutions

University of Cyprus

PennState

ARES (MWMR) - Adaptive, Reconfigurable, Erasure Code, Atomic Storage

[Nicolaou et al 2022]



DAP-based abstract Read/Write specifications

masks host failures by adding/removing servers, and switches between storage algorithms (DAPs)

define the exact methodology to access the object

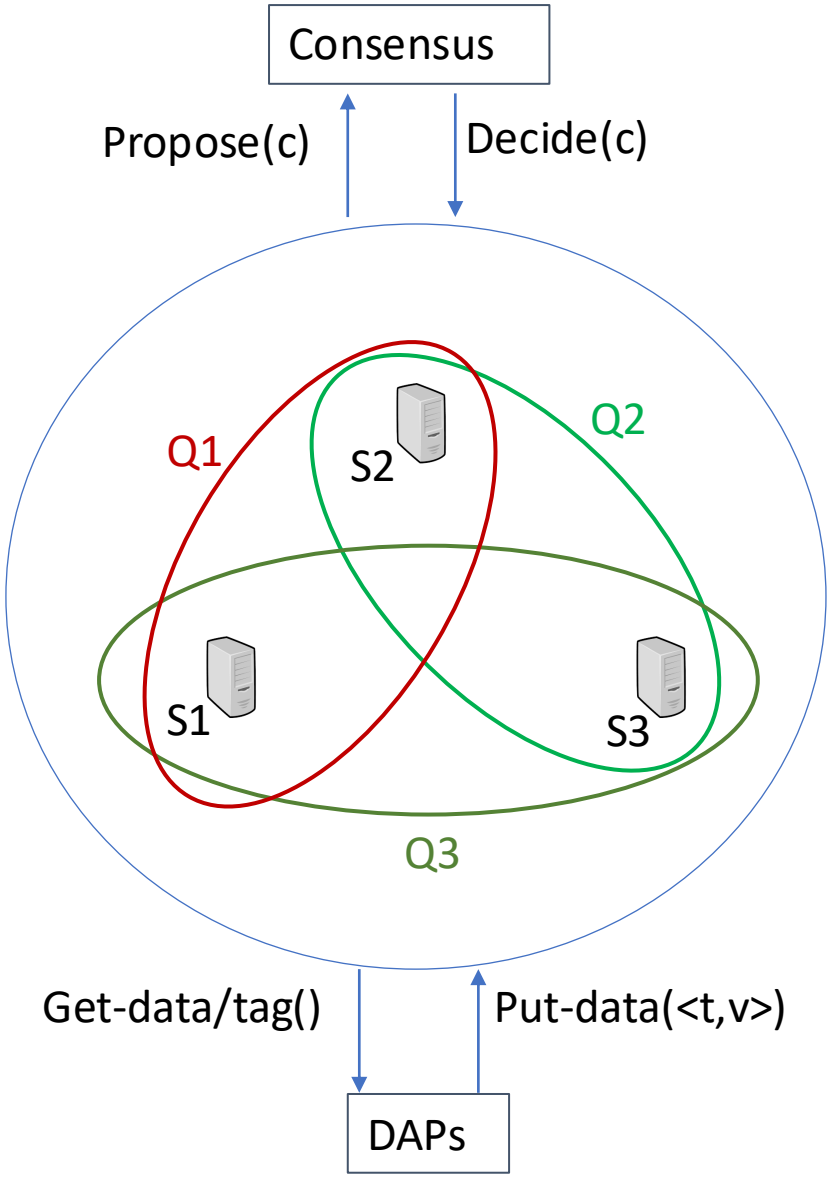
ABD-DAP: uses ABD to access the object replicas

EC-DAP: uses an **erasure coded** based DSM

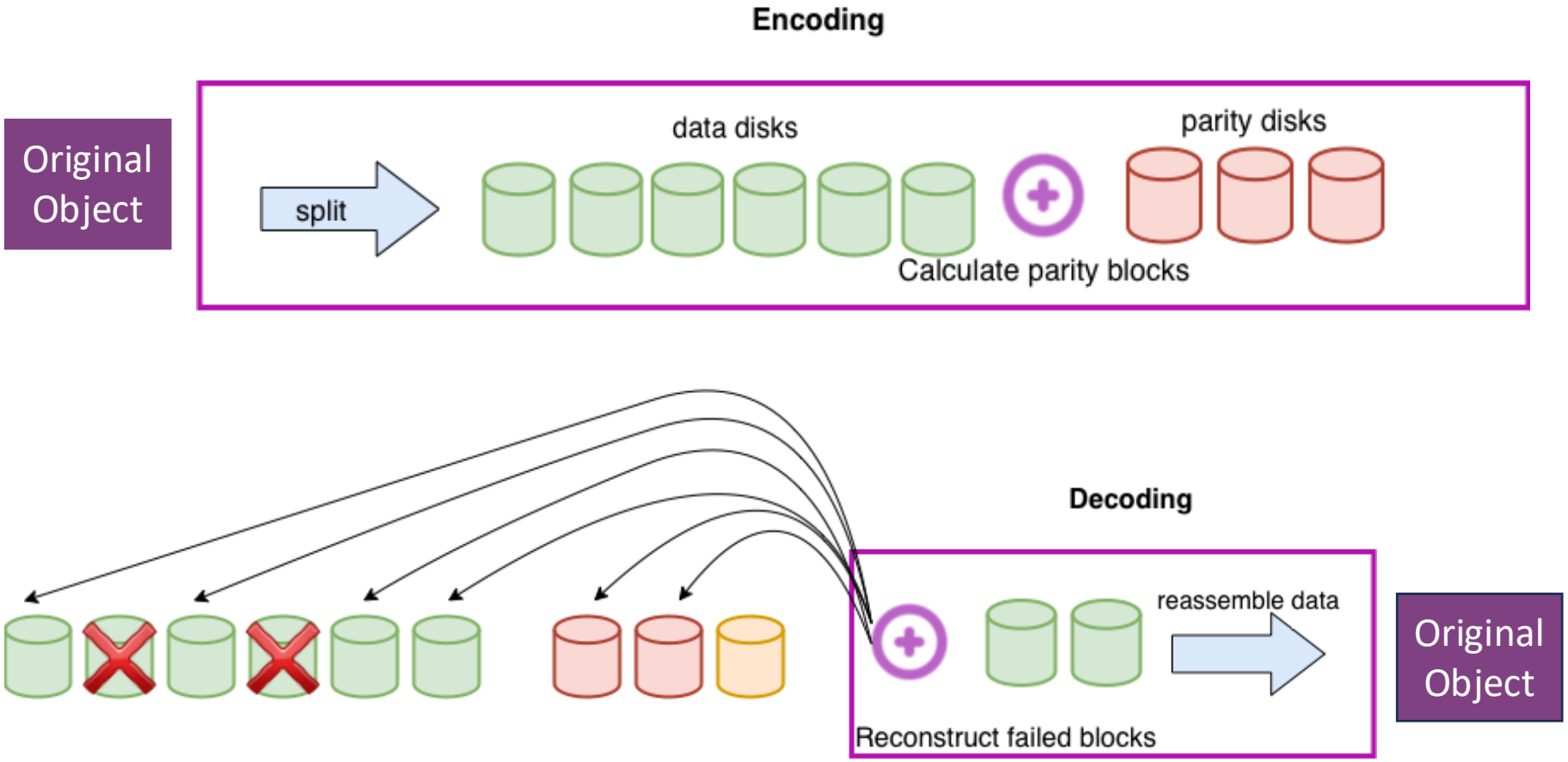
N. Nicolaou, V. Cadambe, N. Prakash, A. Trigeorgi, K. M. Konwar, M. Medard, and N. Lynch, "Ares: Adaptive, reconfigurable, erasure coded, atomic storage," ACM Trans. Storage, jan 2022.

Configurations

- A configuration c is characterized by:
  - A unique identifier
  - A set of servers
  - A quorum set system on servers
  - A consensus instanse
  - A DAP implementation
    - D1. $c.get-tag(): returns\ a\ tag\ \tau \in \mathrm{T}$
    - D2. $c.get-data(): returns\ a\ tag-value\ pair\ (\tau, v) \in \mathrm{T} \times V$
    - D3. $c.put-data(<\tau, v>)$: the tag − value pair$(\tau, v) \in \mathrm{T} \times V$ as argument
    - ABD-DAP & EC-DAP



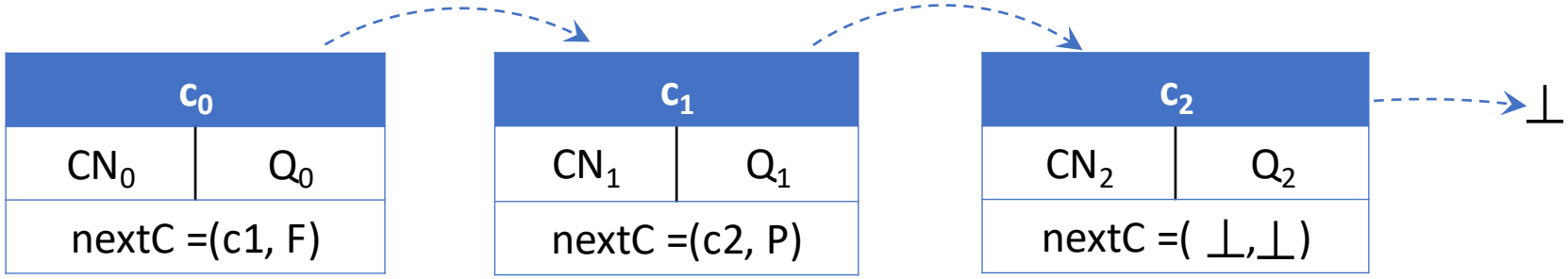
ARES (MWMR) - Adaptive, Reconfigurable, Erasure Code, Atomic Storage



(n, k)-Reed-Solomon code: n=servers, k=data servers, m=parity servers
BUT reads and writes are still applied on the entire object

Configuration Sequence

- Global configuration sequence G_L
- nextC: each server points to the next configuration
  - Same nextC to all servers of a single config c (due to consensus)
- Flags {P, F}: pending, finalized
  - Pending: not yet a quorum of servers received msgs
  - Finalized: new configuration propagated to a quorum of servers



Reconfiguration Service

- A reconfig operation performs 2 major steps:
  1) Configuration *Sequence Traversal*
  2) Configuration *Installation*
     - Transfers the object state from the old to the new configuration

```
6:  operation reconfig(c)
      if c ≠ ⊥ then
8:      cseq ← read-config(cseq)
        cseq ← add-config(cseq, c)
10:     update-config(cseq)
        cseq ← finalize-config(cseq)
12: end operation
```

attempt get to the latest configuration — (1)

introduce the new configuration

move the latest value to the new config

let servers know it is good to be finalized — (2)

This service guarantees that if cseq1 and cseq2 are obtained by two clients resp., then either cseq1 is a prefix of cseq2 or vice versa

Read/Write Operations using DAPs

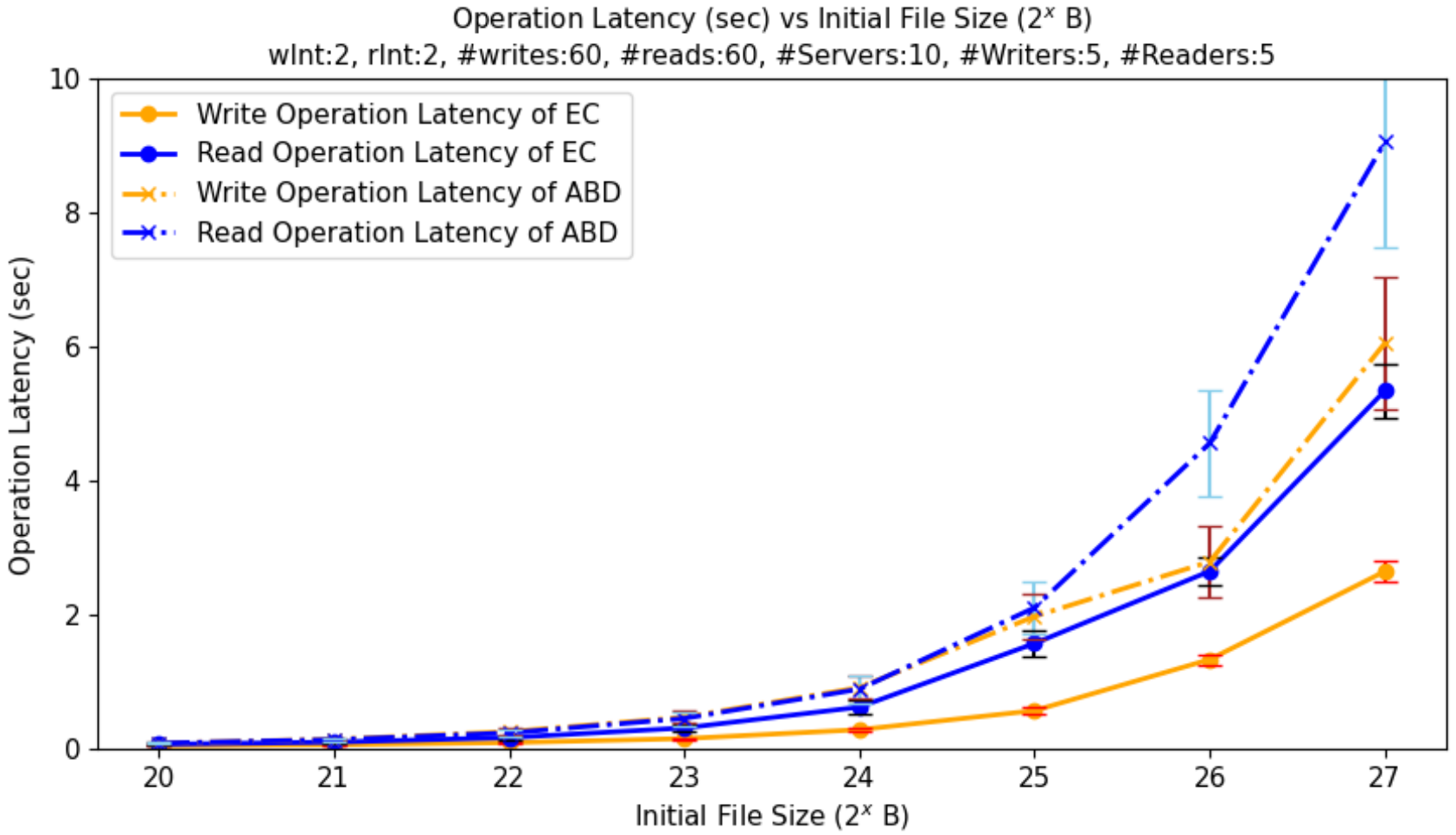
Reader Protocol

- Traverse Config Sequence cseq
- Find μ = max(<c, F>) in cseq
- Set v = last(<c,*>) in cseq
- Discover for μ ≤ i ≤ v
  (t,v)=max(cseq[i].**get-data**())
- Do
  - cseq[v].**put-data**(t,v)
  - Traverse Sequence cseq
- while(|cseq| > v)

Writer Protocol(val) (at w_i)

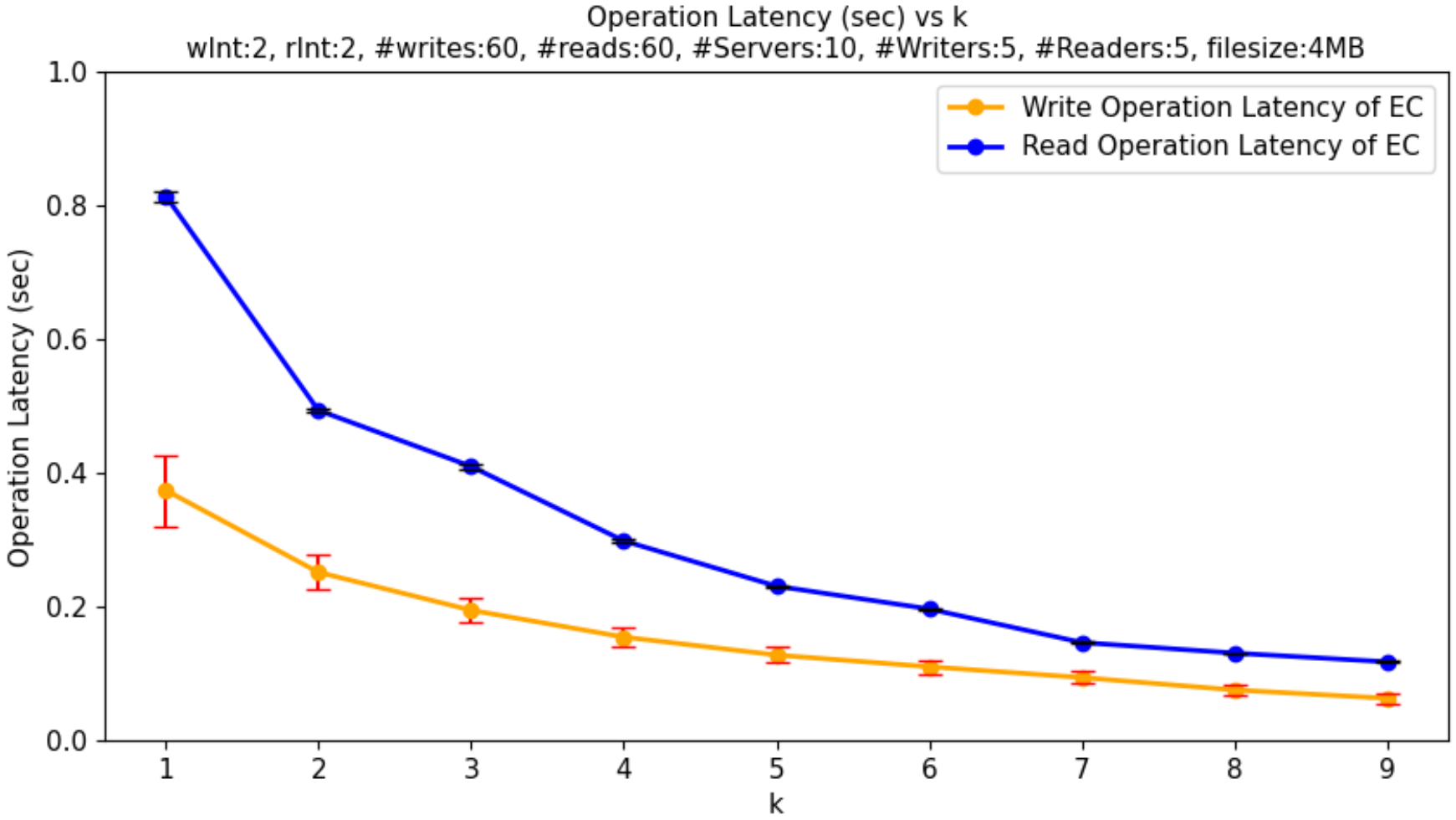
- Traverse Config Sequence cseq
- Find μ = max(<c, F>) in cseq
- Set v = last(<c,*>) in cseq
- Discover for μ ≤ i ≤ v
  t_{max}=max(cseq[i].**get-tag**())
- (t,v)= (<t_{max}+1,w_i>, val)
- Do
  - cseq[v].**put-data**(t,v)
  - Traverse Sequence cseq
- while(|cseq| > v)

File Size (Emulab)



- the read and write latencies of both storage algorithms remain in low levels until 16 MB
- the write operation of EC algorithm is the faster
- the larger messages sent by ABD result in slower read operations

K Scalability (Emulab)



- *small k (=smaller number of data fragments) → bigger sizes of the fragments and higher redundancy.*
- *The write latency seems to be less affected by the number of k since the first phase of write asks only for the tag*

Overview of Results

- The write operation of ARES_EC algorithm is the faster (especially in larger sizes)
- The larger messages sent by ARES_ABD result in slower read operations
- Reconfiguration (server failures & changing DAPs).
- The reconfiguration operation is the slower
- Trade-off between operation latency and parity of EC
- Trade-off between operation latency and the number of writers

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

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

[1]University of Cyprus, Nicosia, Cyprus

[2]Algolysis, Limassol, Cyprus

[3]Pennsylvania State University, PA, USA

SSS 2022

Funded by: *EU's NGIAtlantic.eu cascading grant agreement no. OC4-347*

Main Objective

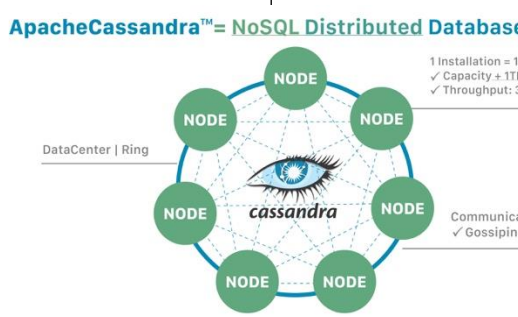
- Primary Limitation: Expensive DSMs that are difficult to implement in an asynchronous, fail prone, message passing environment.

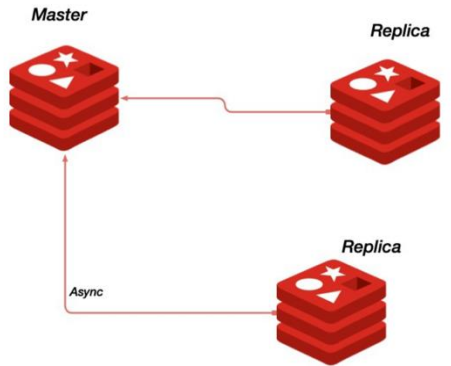
- Recent works: Reduce high communication, storage, and computation overheads (CoBFS, ARES)

How may such algorithms compare to commercially used solutions?

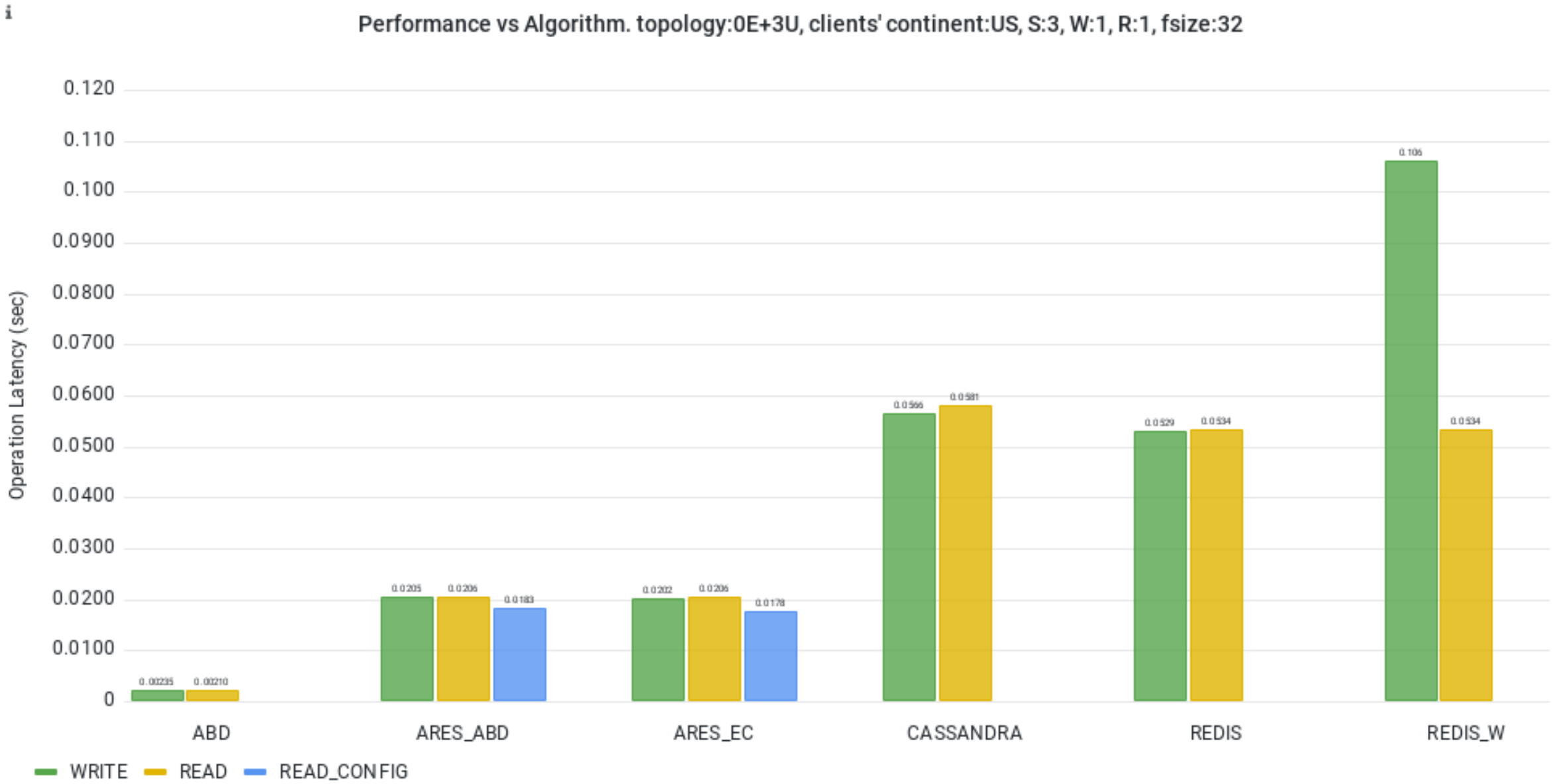
Comparative Table

Algorithm	Consistency guarantees	Data Striping	Non-Blocking Reconfiguration
MWMR ABD	Atomic	NO	NO
ARES-ABD	Atomic	NO	YES
ARES-EC	Atomic	YES	YES
CASSANDRA	Tunable (eventual/atomic)	NO	NO (A single server at a time)
REDIS/REDIS_W	Eventual	NO	NO

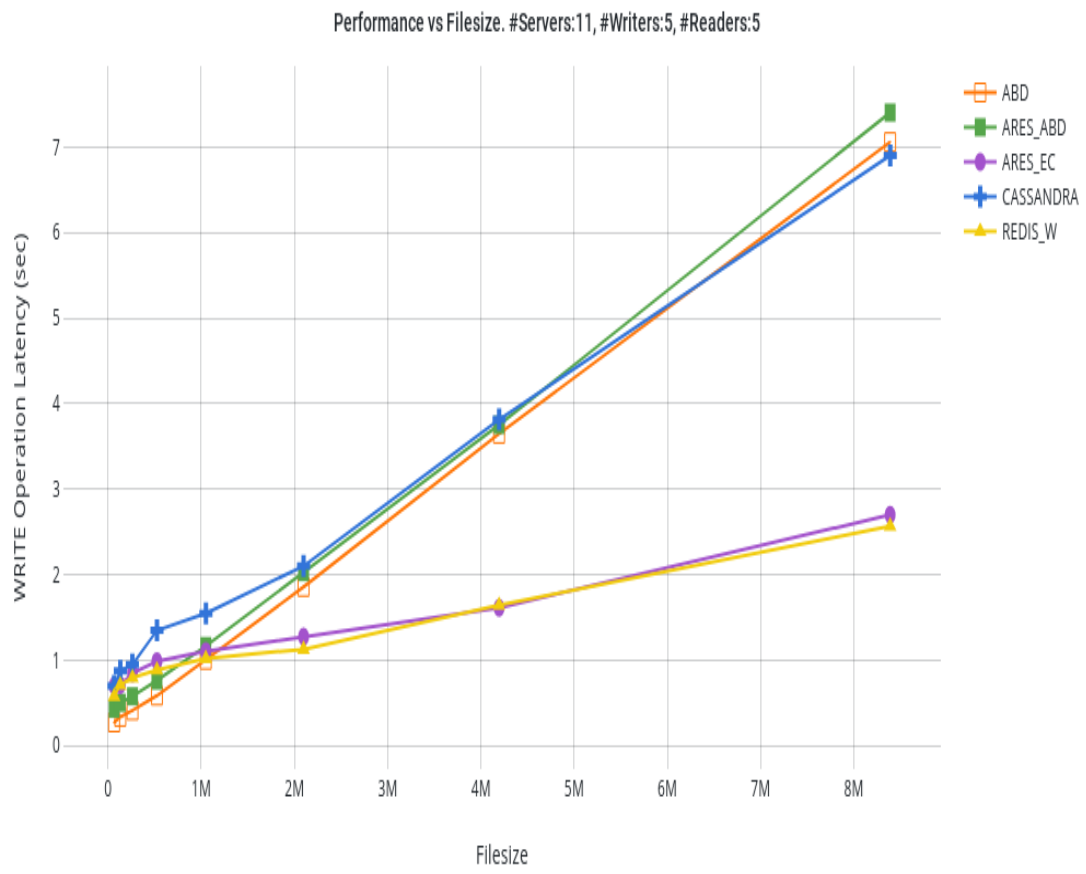


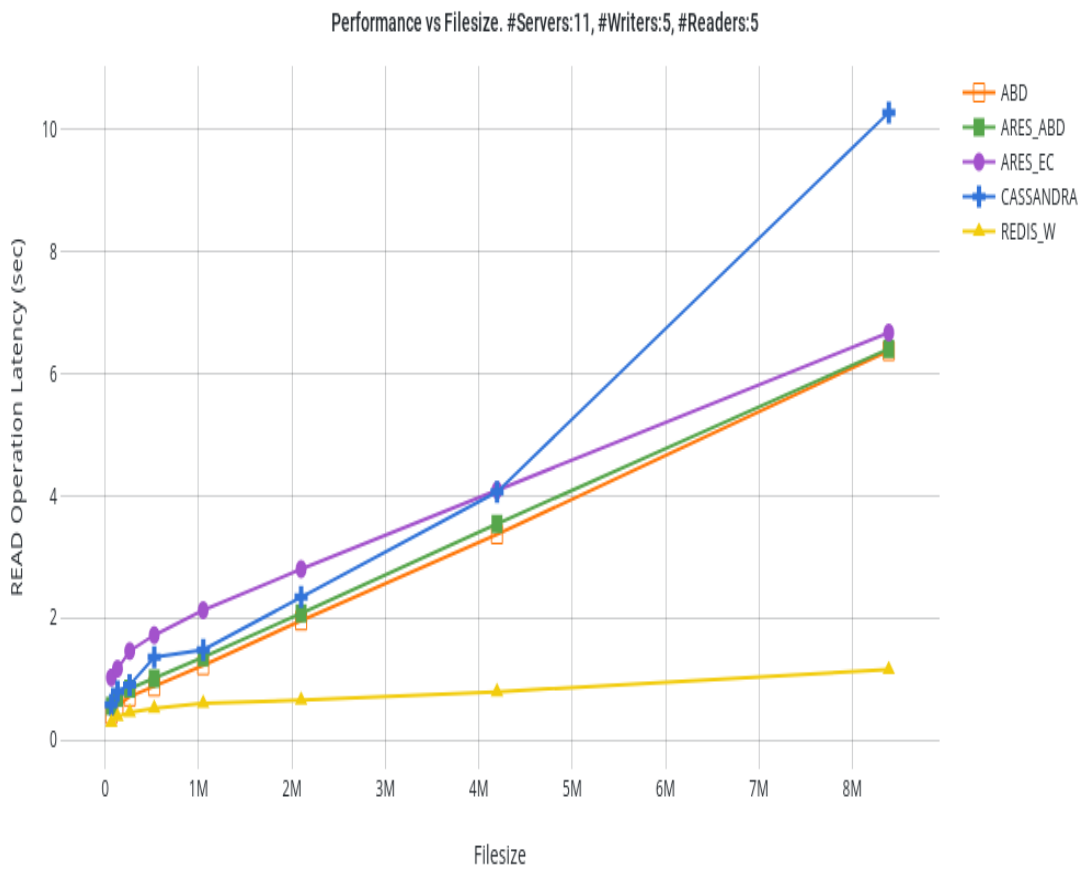


Stress Test – Topology (Fed4FIRE+)



Stress Test – Object Size (Fed4FIRE+)





Overview of Results

- ABD & ARES scale better than Cassandra.
- ARES_EC outperforms Cassandra in larger objects.
- ARES has non-blocking reconfiguration mechanism (server failures & changing DAPs).
- The topology played a major role on the performance (throughput), of all the algorithms we studied.
- Verify the fault-tolerance and the responsiveness of ARES.
- ARES trades performance over consistency with respect to Redis.

Recap and Next Goal

- ARES
  - provable guarantees
  - Operations compete closely/outperform existing DSS solutions (even when offering weaker consistency guarantees).

- Results
  - Data repository: https://github.com/nicolaoun/ngiatlantic-public-data
  - Graphs: https://projects.algolysis.com/ares-ngi/results/

- Next Goal
  - Integrate the dynamic (reconfigurable) DSM algorithm ARES with the DSM module in CoBFS → CoARESF

PART Ⅲ - Fragmented ARES: Dynamic Storage for Large Objects

covering Stages 5, 7 in Methodology

Part Ⅲ - Approach and Contribution

- Integrate the dynamic (reconfigurable) DSM algorithm ARES with the DSM module in CoBFS ⇨ CoARESF

- In order to integrate ARES in CoBFS, we first needed to obtain a coverable version of ARES ⇨ CoARES

- Performed an in-depth experimental evaluation over Emulab and AWS comparing the different versions (with/out fragmentation, with/out EC, with/out reconfiguration).

Fragmented ARES: Dynamic Storage for Large Objects

Authors: Chryssis Georgiou[1], Nicolas Nicolaou[2], Andria Trigeorgi[1]

[1]University of Cyprus, Nicosia, Cyprus

[2]Algolysis, Limassol, Cyprus

DISC 2022

Augusta, GA, USA

CoARES

version: the tag of the coverable object

flag: chg when the write is successful

Query phase: find the max tag-value pair

Main difference: the condition "the writer has the latest version?"

It updates the state of the object!

Algorithm 1 Write and Read protocols for CoAres.

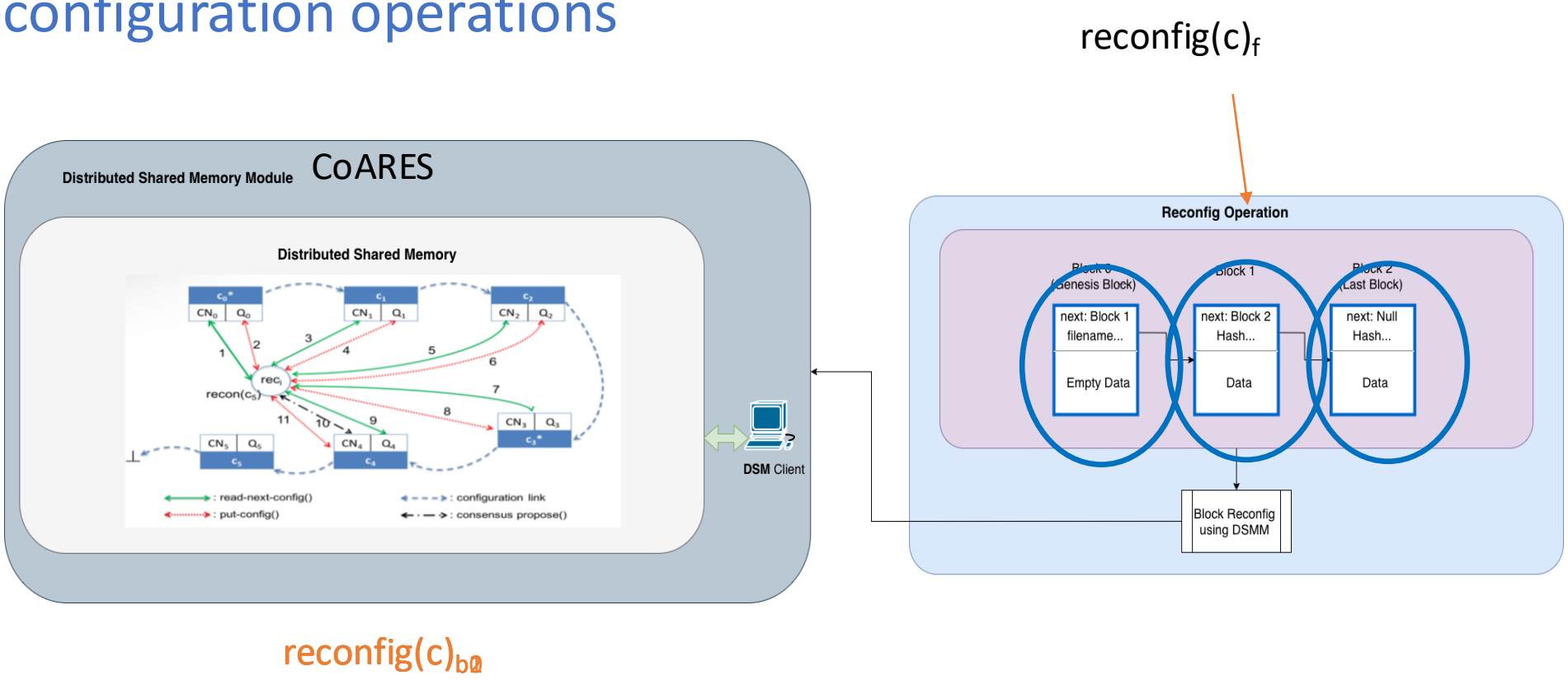
```
1:  CVR-Write Operation:
2:    at each writer w_i
3:    State Variables:
4:      cseq[] s.t. cseq[j] ∈ C × {F, P}
5:      version ∈ N+ × W ∪ {⊥} initially ⟨0, ⊥⟩
6:    Local Variables:
7:      μ ∈ N+ initially 0, ν ∈ N+ initially 0
8:      τ ∈ N+ × W initially ⟨0, w_i⟩
9:      v ∈ V initially ⊥
10:     flag ∈ {chg, unchg} initially unchg
11:   Initialization:
12:     cseq[0] = ⟨c_0, F⟩

13:   operation cvr-write(val), val ∈ V
14:     cseq ← read-config(cseq)
15:     μ ← max({i : cseq[i].status = F})
16:     ν ← |cseq|
17:     for i = μ : ν do
18:       ⟨τ, v⟩ ← max(cseq[i].cfg.get-data(), ⟨τ, v⟩)
19:     if version = τ then
20:       flag ← chg
21:       ⟨τ, v⟩ ← ⟨⟨τ.ts + 1, ω_i⟩, val⟩
22:     else
23:       flag ← unchg
24:     version ← τ
25:     done ← false
26:     while not done do
27:       cseq[ν].cfg.put-data(⟨τ, v⟩)
28:       cseq ← read-config(cseq)
29:       if |cseq| = ν then
30:         done ← true
31:       else
32:         ν ← |cseq|
33:     end while
34:     return ⟨τ, v⟩, flag
35:   end operation
36: CVR-Read Operation:
37:   at each reader r_i
38:   State Variables:
39:     cseq[] s.t. cseq[j] ∈ C × {F, P}
40:   Initialization:
41:     cseq[0] = ⟨c_0, F⟩

42:   operation cvr-read( )
43:     cseq ← read-config(cseq)
44:     μ ← max({j : cseq[j].status = F})
45:     ν ← |cseq|
46:     for i = μ : ν do
47:       ⟨τ, v⟩ ← max(cseq[i].cfg.get-data(), ⟨τ, v⟩)
48:     done ← false
49:     while not done do
50:       cseq[ν].cfg.put-data(⟨τ, v⟩)
51:       cseq ← read-config(cseq)
52:       if |cseq| = ν then
53:         done ← true
54:       else
55:         ν ← |cseq|
56:     end while
57:     return ⟨τ, v⟩
58:   end operation
```

It keeps the max tag!

Difference: it returns both the value and the version

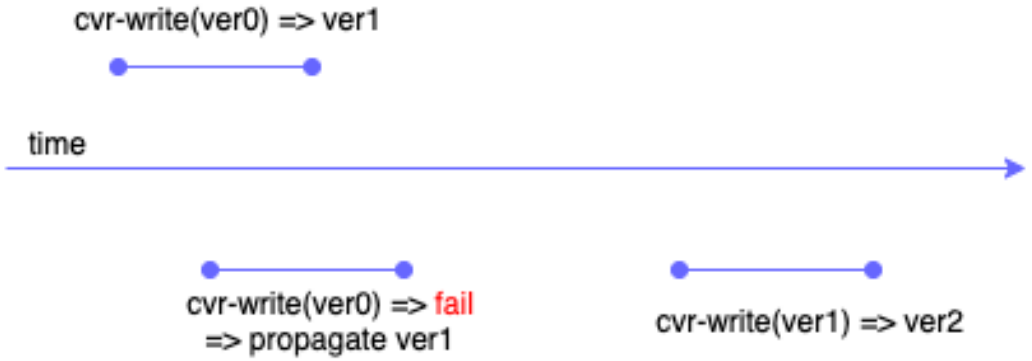
CoARESF

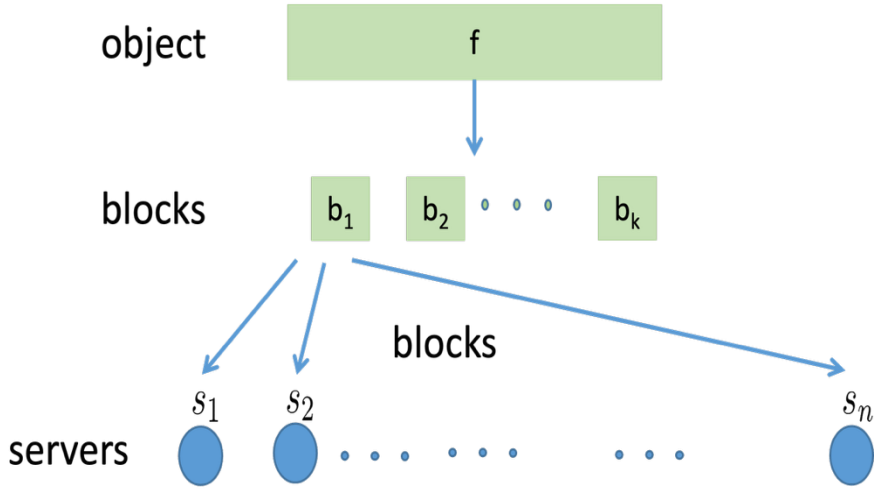
- Integrate CoARES with CoBFS
- Main challenge: Enable the fragmentation approach to invoke reconfiguration operations



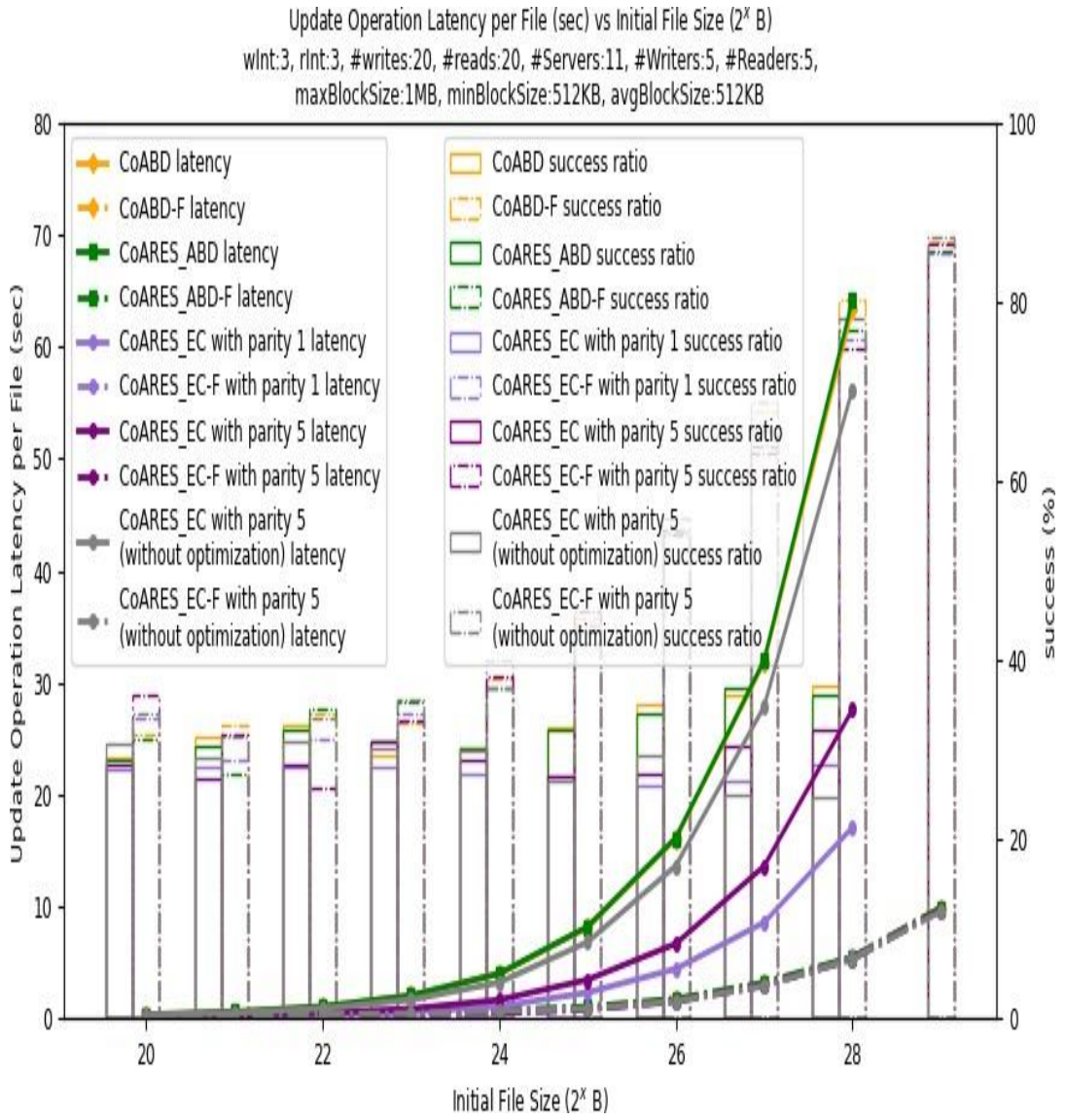
ARES VERSIONS

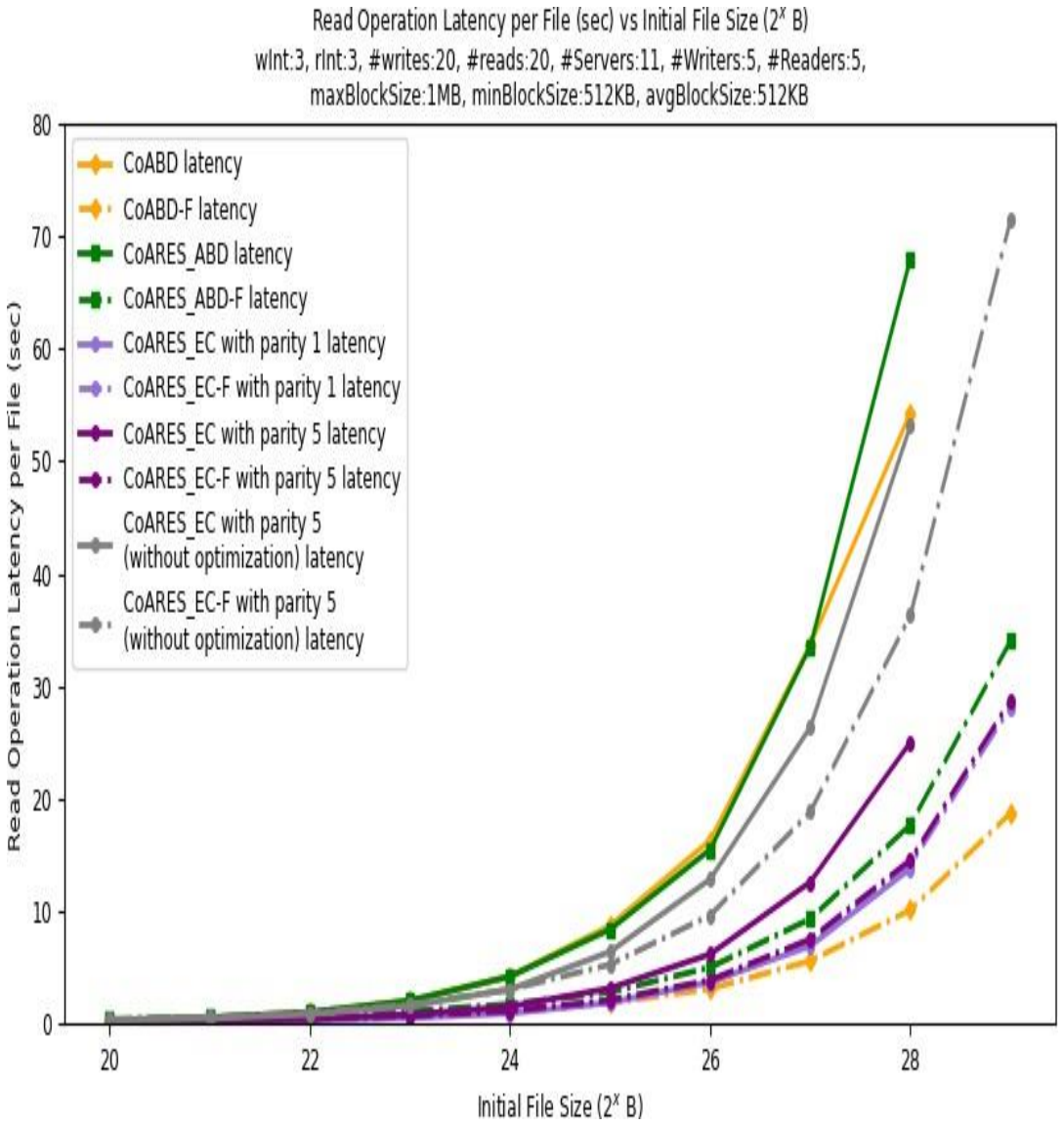
ARESABD	This is Ares that uses the ABD-DAP implementation.
CoARESABD	The coverable version of *ARESABD.*
CoARESABDF	The fragmented version of *CoARESABD*.
ARESEC	This is *ARES* that uses the EC-DAP implementation.
CoARESEC	The coverable version of *ARESEC.*
CoARESECF	This is the two-level data striping algorithm obtained when *CoARESF* is used with the EC-DAP implementation; i.e., it is the fragmented version of *CoARESEC*.



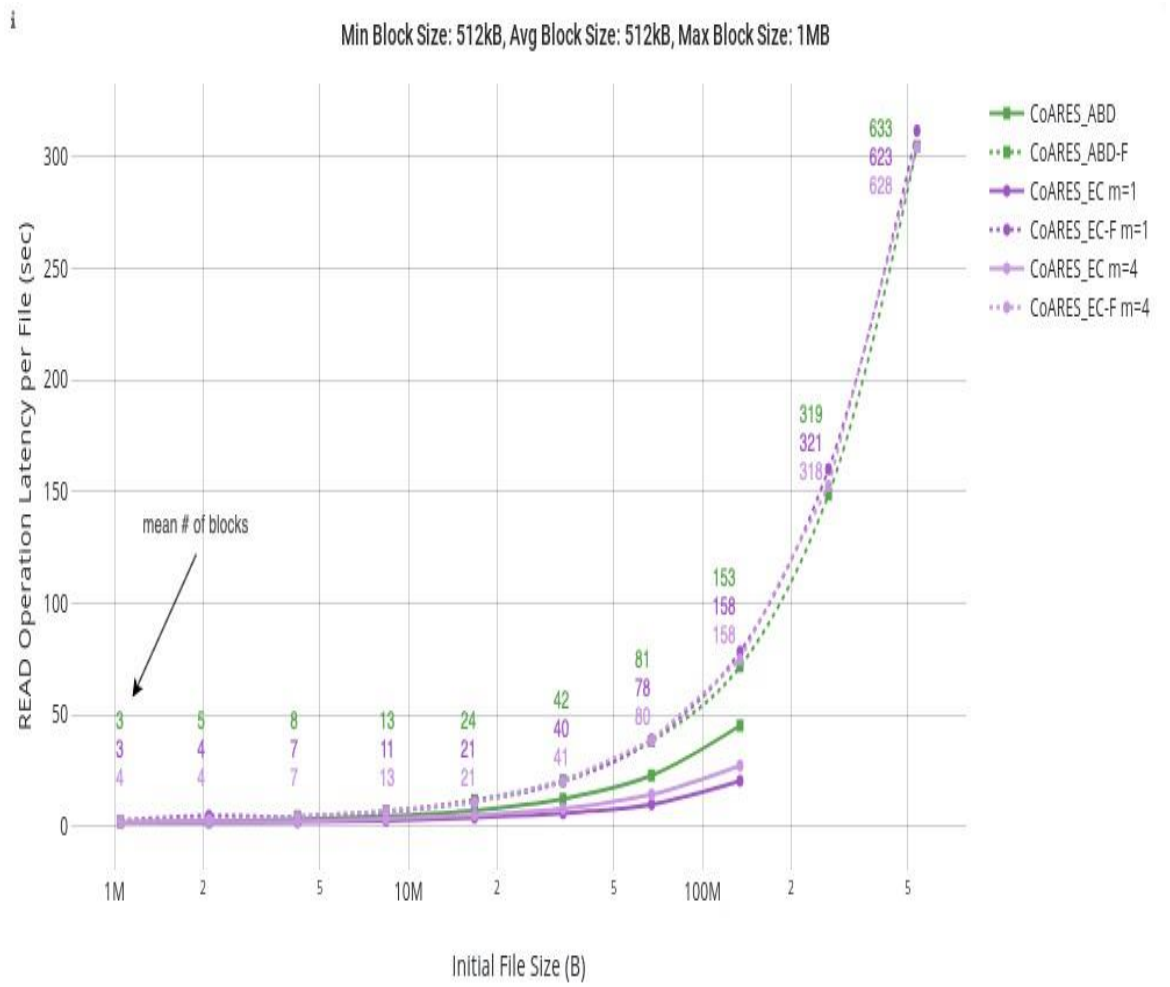


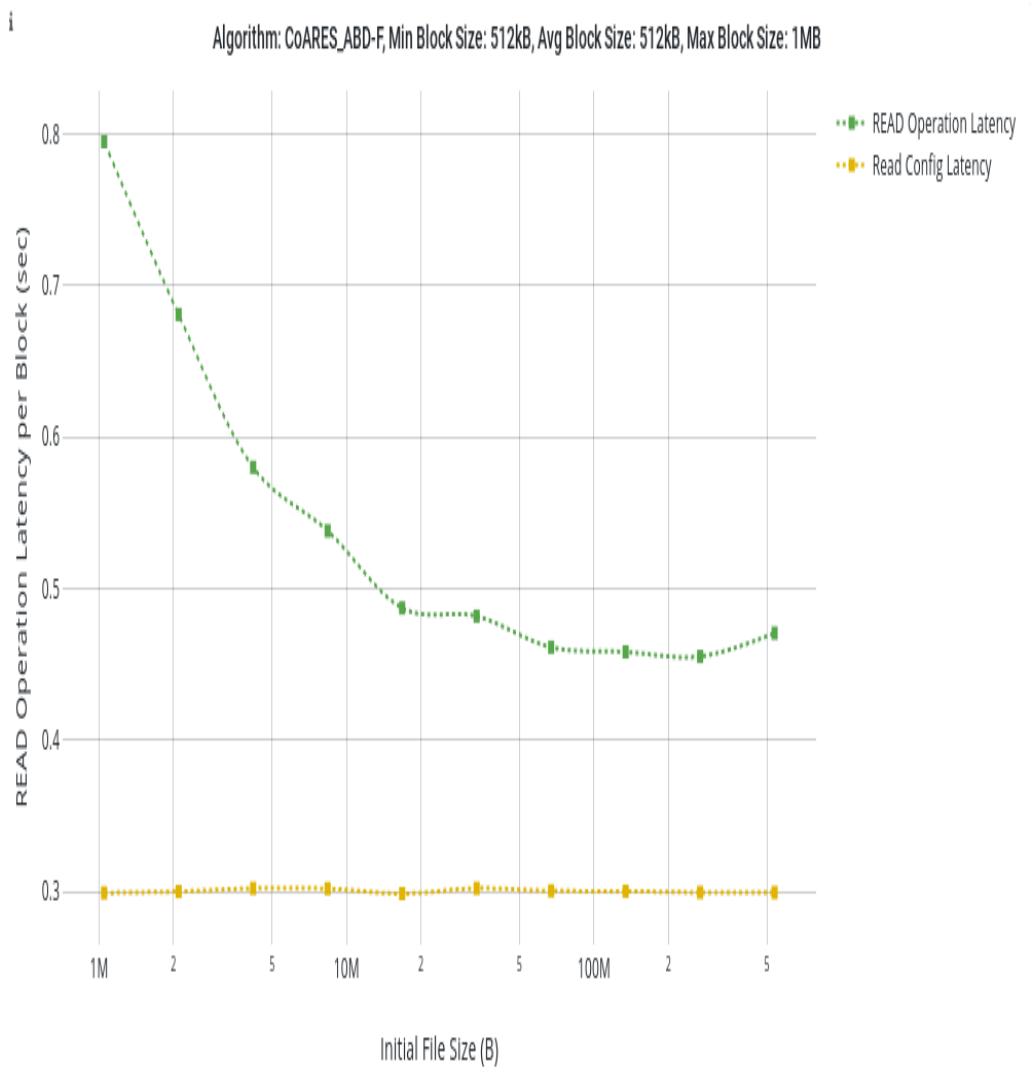
File Size (Emulab)





File Size (AWS)





Overview of Results

- Fragmented algorithms have significantly lower write operation latency both in Emulab and AWS
- For the read operation latency, AWS results of CoARESF suggest that there is room for improvement
- EC-based algorithms are the most scalable as the servers increase
- Trade-off between block size, operation latency and write success rate
- Trade-off between operation latency and the parity of EC
- Trade-off between operation latency and the number of writers

Recap

- Presented and proved correct CoARESF, the first Dynamic, Robust, Strongly-consistent DSM that supports Versioning, (2-level) Data Striping, and High Access Concurrency for Large Objects

- Data available at https://github.com/atrigeorgi/fragmentedARES-data.git

PART $\mathbf{IV}$ – Enhance the Performance of ARES

covering Stages 7-8 in Methodology

Part Ⅳ - Approach and Contribution

- **Identify flaws** in **DSMs** using **Distributed Tracing**.
  We demonstrate this through the ***ARES*** DSM.

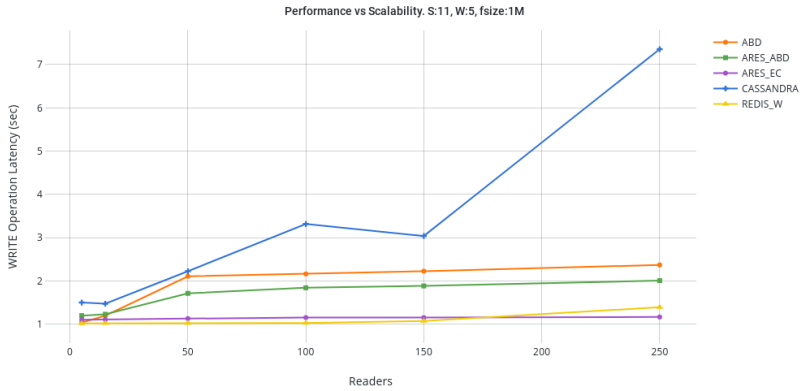
- Develop optimizations based on the found bottlenecks.

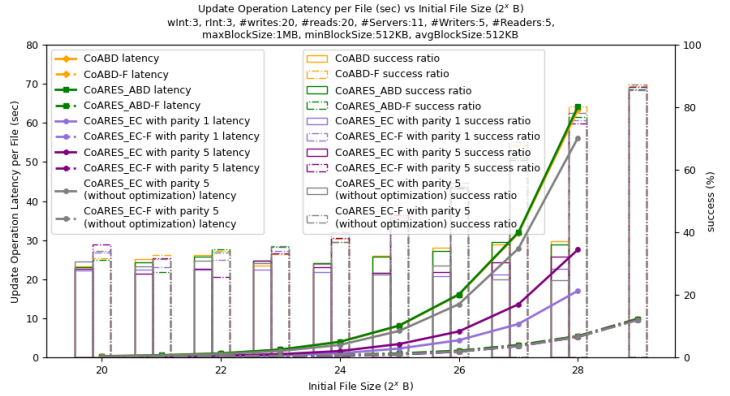
- The correctness of optimized ARES is rigorously proven.

Performance Analysis Challenges in DSMs

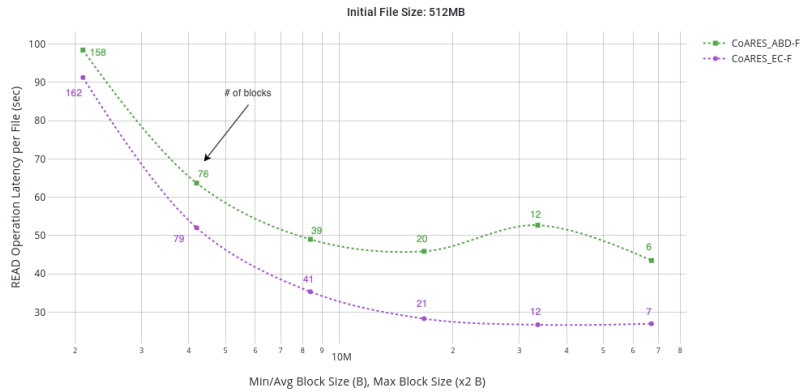
- Identifying performance bottlenecks in complex DSMs can be challenging
- Traditional logging techniques may not provide sufficient insight

```python
import os
import logging
from sys import platform
from pythonjsonlogger import jsonlogger

atrigeorgi
class SetupLogger:

    atrigeorgi
    def setup_logger(self, logfile, level=logging.DEBUG):
        # Due to race conditions, we sometimes get error.
        # OSError: [Errno 17] File exists: 'log'
```






```python
self.logger.debug('READ-COMPLETE-DSMM',
                  extra={"clientID": self.uid, "objectID": file_id, "tag": maxTag, "value": value})
```

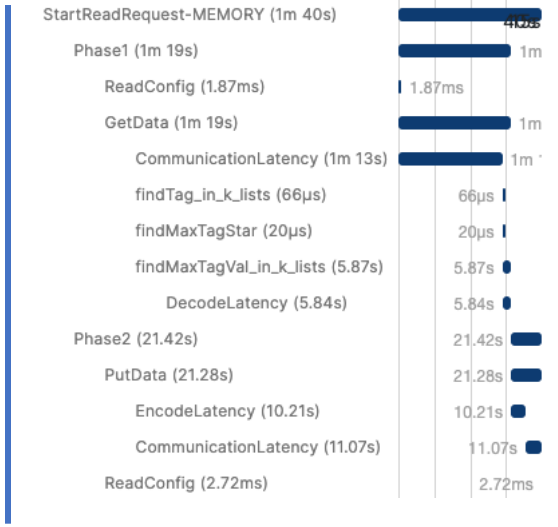


“Distributing Tracing is a monitoring technique used to track individual requests as they move across multiple components within a distributed system. It helps to pinpoint where failures occur and what causes poor performance.”

Distributed Tracing – Terminology

- A **trace** represents the entire journey of a request.
- A **span** represents a unit of work within a trace (e.g., procedures, sections of code).
- Tracings tools: Opentemetry, Zipkin, Jaeger.

```
with self.tracer.start_as_current_span("StartWriteRequest-MEMORY"+self.phase) as parent_span:
    with self.tracer.start_as_current_span("Phase1"):
        with self.tracer.start_as_current_span("GetTag"):

            # increase msg counter for next op
            self.msg_id += 1

            # Creating 'READ' message with the following fields
            # [type, msgID, clientID, objectID]
            message = {'type': 'READ-TAG', 'msgID': self.msg_id, 'clientID': self.uid, 'objectID': file_id}

            with self.tracer.start_as_current_span("CommunicationLatency"):
                # Broadcast it
                self.broadcastMessage(message)

                # Wait for READ-TAG-ACK messages to come from a majority
                msgs_list = self.waitReply(self.majority, "READ-TAG-ACK")
```



Tracing the Latencies of Ares: A DSM Case Study

Authors: Chryssis Georgiou[1], Nicolas Nicolaou[2], **Andria Trigeorgi**[1,2]

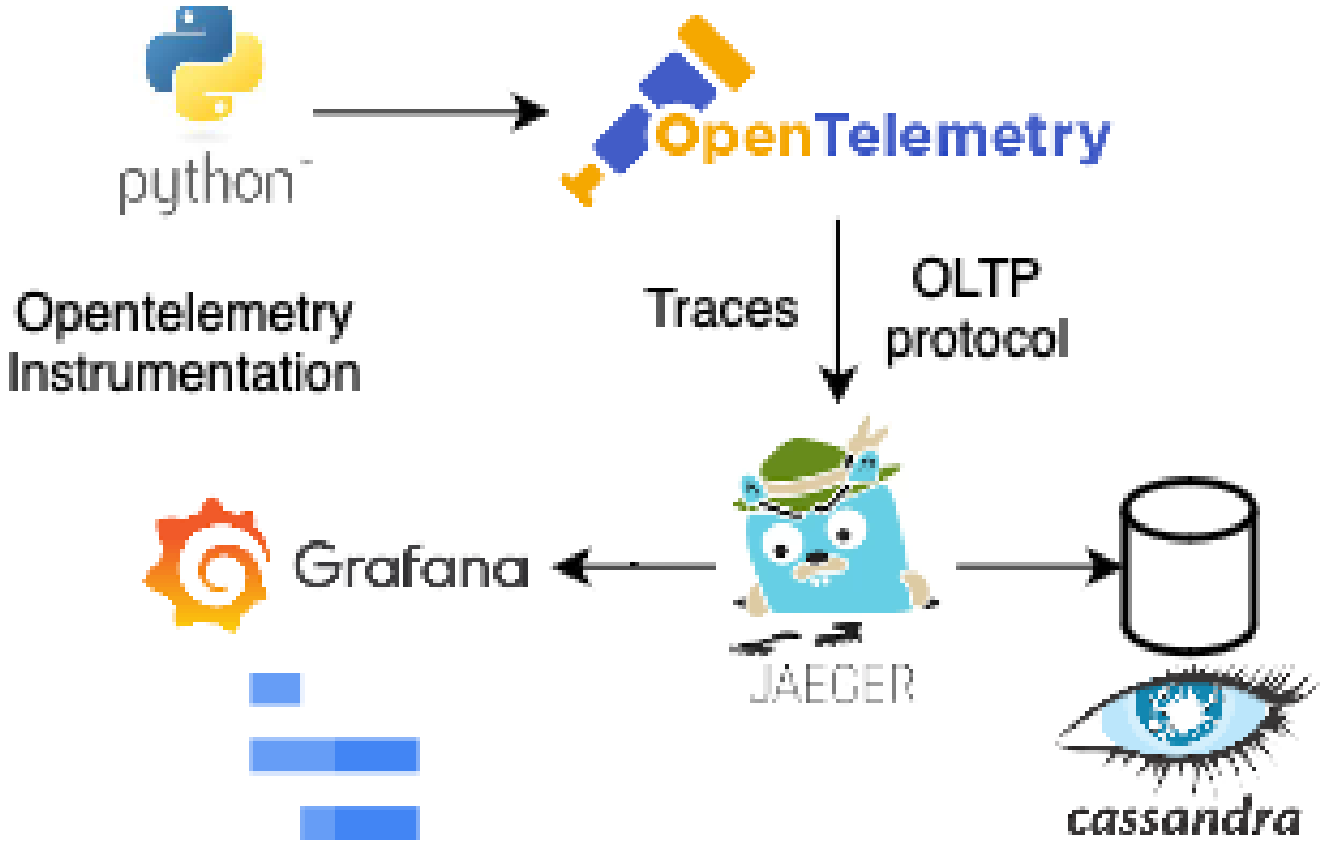
[1]University of Cyprus, Nicosia, Cyprus

[2]Algolysis, Limassol, Cyprus

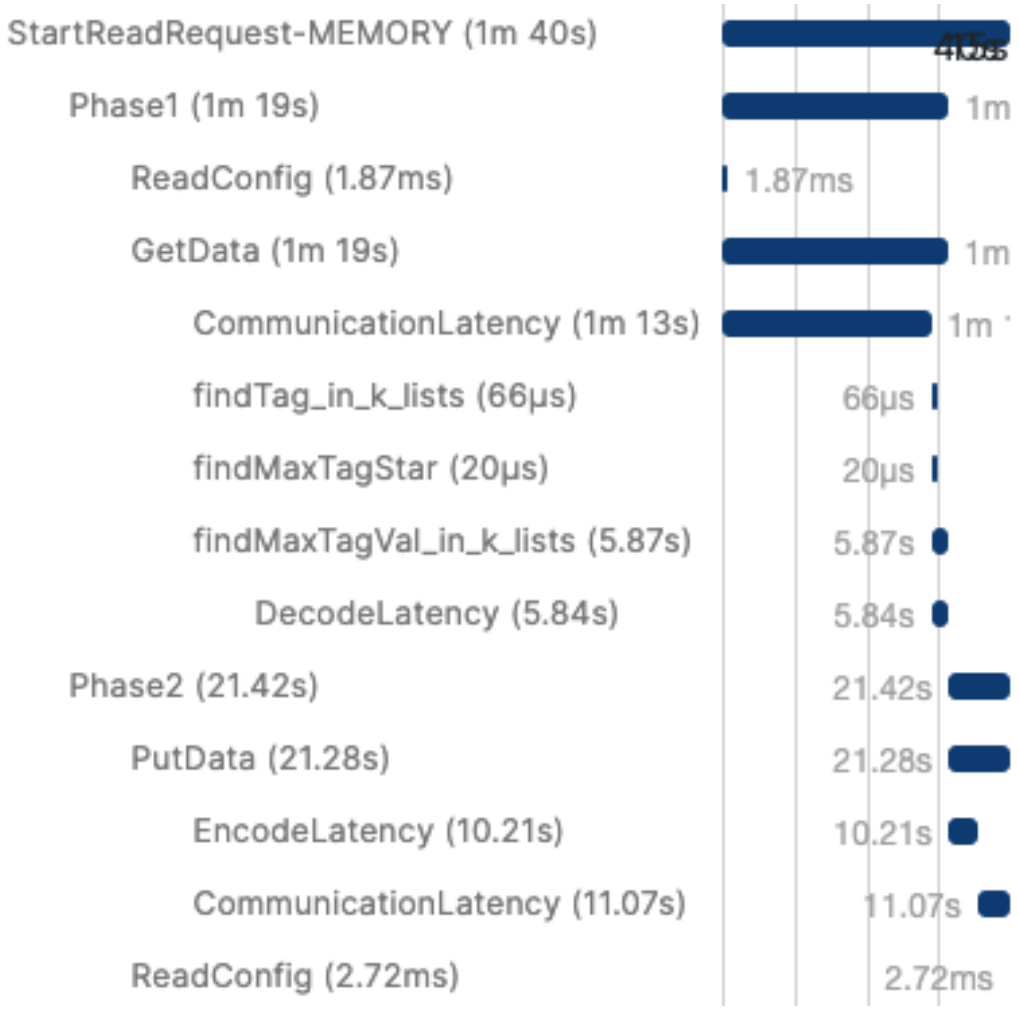
ApPLIED 2024, Nantes, France

Funded by: PHD IN INDUSTRY/1222/0121 and DUAL USE/0922/0048

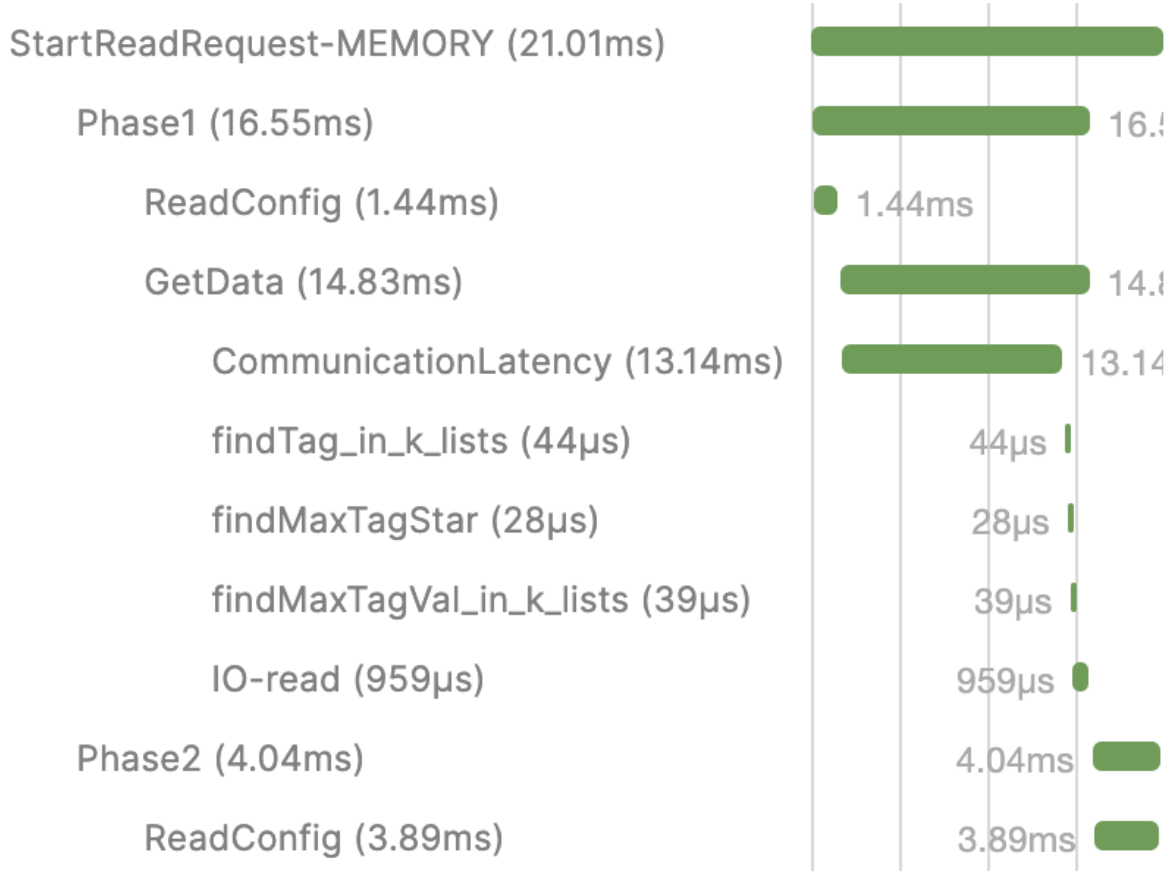
Methodology: ARES Distributed Tracing



File Size

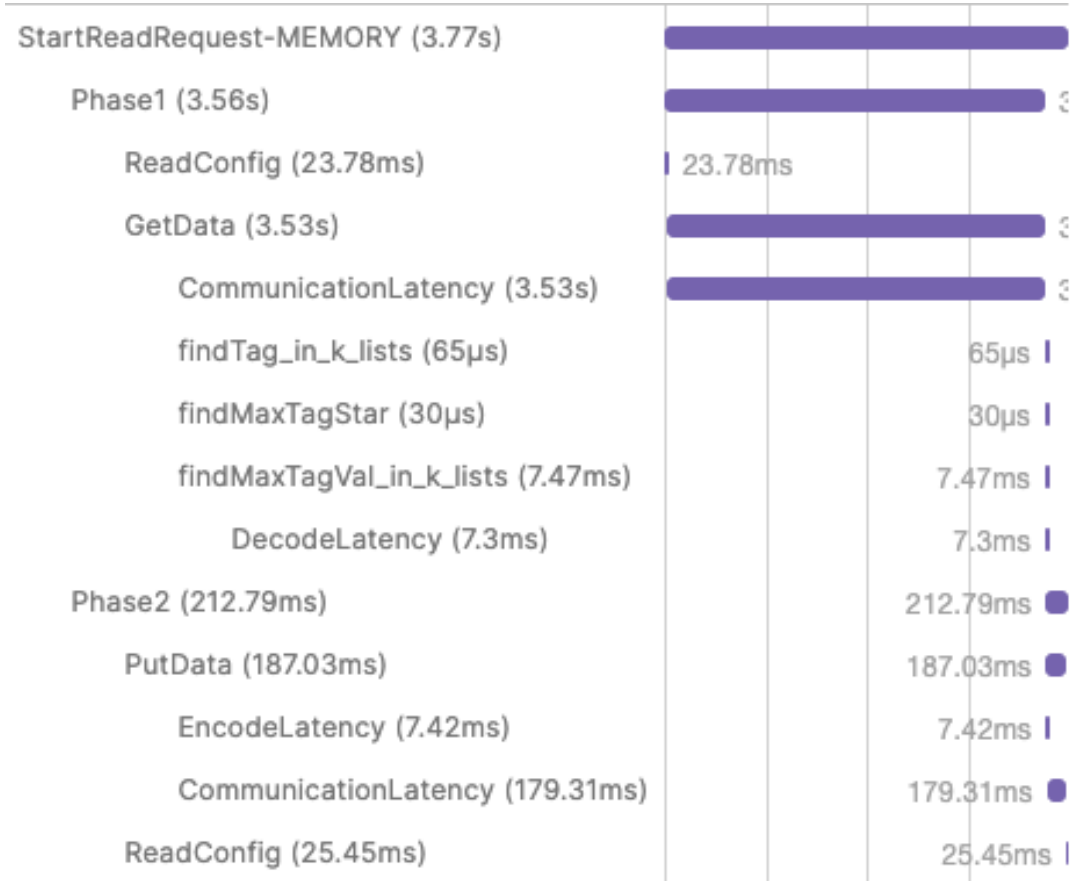


ARESEC, S:11, W:5, R:5, fsize:512MB, Debug Level:DSMM

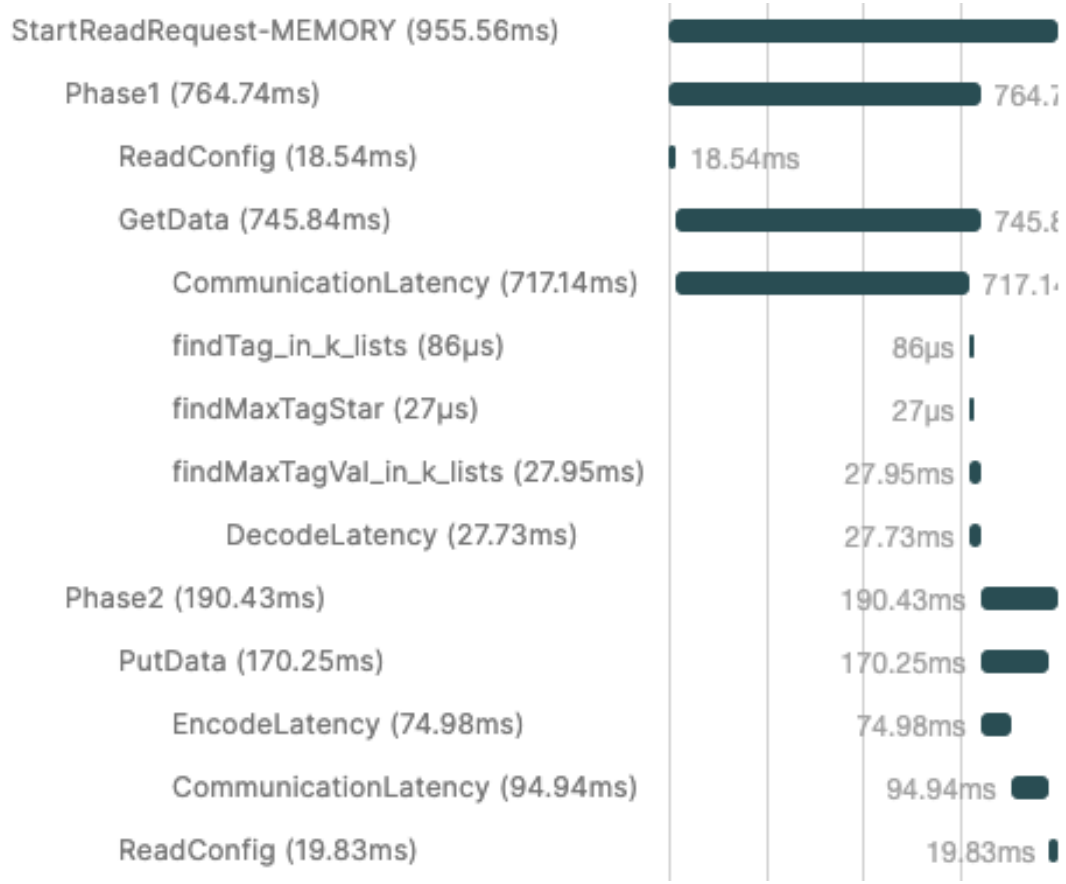


CoARESECF, S:11, W:5, R:5, init fsize:512MB, Debug Level:DSMM

Participation Scalability



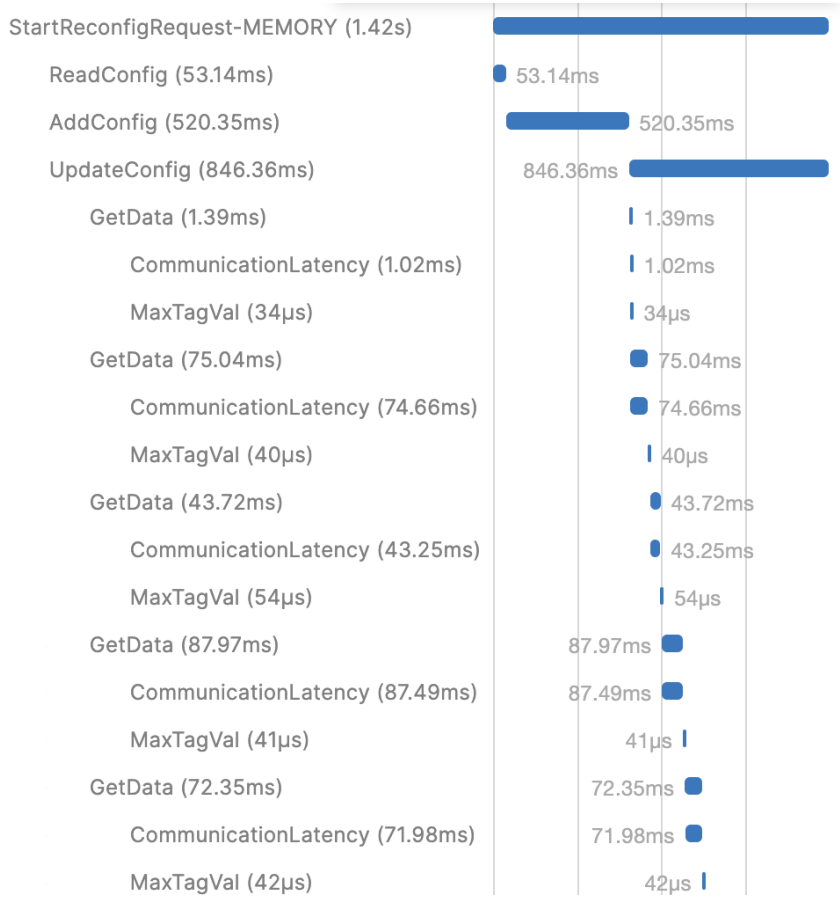
ARESEC, S:3, W:5, R:50, fsize:4MB, Debug Level:DSMM



ARESEC, S:11, W:5, R:50, fsize:4MB, Debug Level:DSMM

Longevity

CoAresF, S:11, W:5, R:15, G=5, fsize:4MB, Debug Level:DSMM



CoAresF, S:11, W:5, R:15, G=5, fsize:4MB, Debug Level:DSMM

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

Authors: Chryssis Georgiou[1], Nicolas Nicolaou[2], **Andria Trigeorgi**[1,2]

[1]University of Cyprus, Nicosia, Cyprus

[2]Algolysis, Limassol, Cyprus

SRDS 2024, Charlotte, USA

Funded by: PHD IN INDUSTRY/1222/0121 and DUAL USE/0922/0048

Optimisation 1: Piggybacking

Main difference:
skip read-config
for latest config

Extra function:
discover the
next config

Query phase:
embed latest
configs with data

```
operation write(val), val ∈ V
8:    cseq ← read-config(cseq)
      μ ← max({i : cseq[i].status = F})
10:   cs ← cseq[μ]
      while cs ≠ ⊥ do
12:     τc, Cs ← cs.cfg.get-tag()
        τmax ← max(τc, τmax)
14:     cs, cseq ← find-next-config(cseq, Cs)
      end while
16:   ⟨τ, v⟩ ← ⟨⟨τmax.ts + 1, ωi⟩, val⟩

      λ ← max({i : cseq[i] ≠ ⊥})
18:   cs ← cseq[λ]
      while cs ≠ ⊥ do
20:     Cs ← cs.cfg.put-data(⟨τ, v⟩)
        cseq ← read-config(cseq)
22:     cs, cseq ← find-next-config(cseq, Cs)
      end while
24: end operation
```

Optimisation 1: Piggybacking – EC-DAP

requests max τ and nextC in one go

returns max τ and servers' nextC

requests data list and nextC

Algorithm 3 EC-DAP II implementation

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

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

procedure c.get-data()
10: **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 $c.Servers$
12: $Cs \leftarrow \{nextC_s :$ received $nextC_s$ from $s\}$
  $Tags_{dec}^{\geq k} =$ set of tags that appears in k Lists
14: $t_{max}^{dec} \leftarrow \max(Tags_{dec}^{\geq k})$
  if $Tags_{dec}^{\geq k} \neq \emptyset$ **then**
16: $fragments \leftarrow \{e : \langle \tau, e \rangle \in Lists \ \& \ \tau = t_{max}^{dec}\}$
  if $\nexists \perp \in fragments$ **then**
18: $v \leftarrow$ decode value for t_{max}^{dec}
  else
20: $v \leftarrow \perp$
  return $\langle t_{max}^{dec}, v \rangle, Cs$
22: **end procedure**

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

If nextC is finalized, servers sends only the tag and their nextC

requests data update and nextC

returns all servers' nextC

Optimisation 2: Garbage Collection

```
procedure gc-config(cseq)
38:   μ ← max({i : cseq[i].status = F})
      CID ← {i : cseq[i] ≠ ⊥ ∧ i < μ}
40:   for id in CID do
        send (GC-CONFIG, next) to each s ∈ cseq[id].Servers
42:     until ∃Q, Q ∈ cseq[id].Quorums s.t. rec_i receives ACK from ∀s ∈ Q
        /* remove the {id, cseq[id]} */
44:     cseq ← cseq \ {id, cseq[id]}
      return cseq
46: end procedure
```

Last finalized config

older configs from last finalized config

send GC request to servers

Remove the GC configs

EC-DAP

server updates nextC to point to the last finalized config

the server sets the data of config to ⊥

```
Upon receive (GC-CONFIG, cfgT_in) s_i, c_k from q
  if cfgT_in.cfg.ID > nextC.cfg.ID then
    nextC ← cfgT_in
  for τ, e in List do
    List ← List \ {⟨τ, e⟩}
    List ← List ∪ {⟨τ, ⊥⟩}
  send ACK to q
end receive
```

Server' Response

Optimisation 3: Batching

```
6:  operation reconfig(c, D)
      if c ≠ ⊥ then
8:      cseq ← read-config(cseq)
        cseq ← add-config(cseq, c)
10:     update-config(cseq, D)
        cseq ← finalize-config(cseq)
12:     cseq ← gc-config(cseq)
    end operation
```

executes on a domain of objects

transfer latest tag-value pairs for each object from configs

```
    procedure update-config(cseq , D)
24:   μ ← max({j : cseq[j].status = F})
      λ ← max({j : cseq[j] ≠ ⊥})
26:   M = []
      for o in D do
28:     M[o] ← ∅
        for i = μ : λ do
30:       for o in D do
            ⟨t, v⟩, _ ← cseq[i].cfg.get-data() for object o
32:         M[o] ← M[o] ∪ {⟨τ, v⟩}
        for o in D do
34:       ⟨τ, v⟩ ← max_t{⟨t, v⟩ : ⟨t, v⟩ ∈ M[o]}
          cseq[λ].cfg.put-data(⟨τ, v⟩) for object o
36:   end procedure
```

gathers the tag-value pairs for each object o in D

transfers the maximum pair to the new config

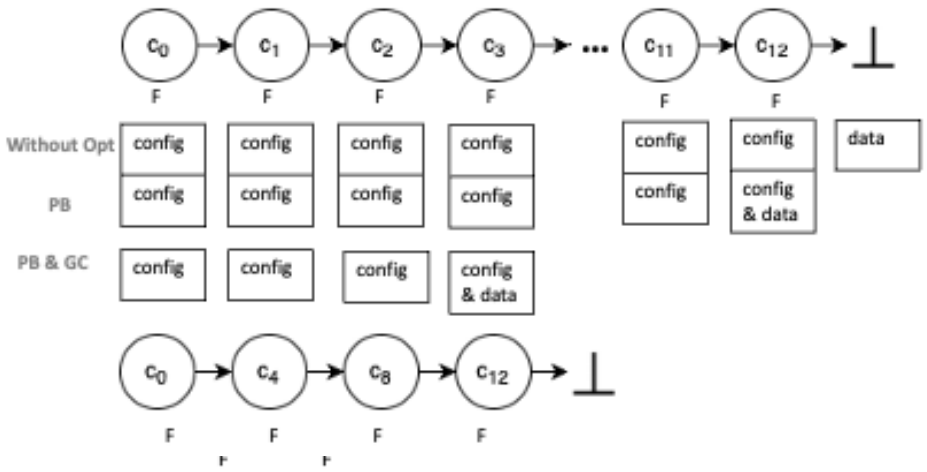
Optimization Results – Piggyback

alg./f_{size}	CoAresABDF	CoAresABDF PB	CoAresECF	CoAresECF PB
1MB	284ms	278ms	149ms	142ms
256MB	9s	5s (44%)	9.65s	3.82s (60%)
512MB	21.8s	15.2s (30%)	23.2s	10.9s (53%)

TABLE I: READ Operation - File Size - S:11, W:5, R:5

- **CoARESF (256 MB & 512 MB):** Significant performance drops without optimization.
  - Non-optimized: **4 rounds per block** with double read-config.
  - With PB Optimization: Reduced to **2 rounds**, lower read latency.

Optimization Results – Garbage Collection



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 II: READ Operation - Reconfigurations - S:11, W:1, R:10:, G:4

Scenario 1:

- PB version has the worst latency, since transfers data and config in 11 round trips.
- PB with GC is fastest, since it updates pointers reducing actions.
- CoARESF & Larger Objects (64MB): No differences between versions since the first block finds the latest config and the next block starts from that config.

Scenario 2:

- Original vs. PB has similar performance with one extra round trip.
- PB with GC is faster, skipping every 4 configurations, fewer rounds needed.

Recap

- Used tracing to pintpoint inefficiencies by monitoring individual procedures.
- Develop optimizations, leading to ARES II.
- Show the **correctness** of ARES II and conduct performance **evaluations** to showcase its improvements over ARES.
- Distributed tracing is crucial for diagnosing and resolving performance issues in DSM algorithms.

Optimization Strategies

- **Piggy-backing**: Integrating configurations with read/write messages to expedite configuration discovery.
- **Garbage Collection**: Eliminating obsolete configurations for quicker access to the latest data.
- **Data Batching**: A single reconfiguration across multiple objects to enhance efficiency.

Conclusion

- CoBFS has the following advantages:
  - High Concurrent accesses
  - Strong Consistency
  - Large file size (tested up to 1GB file)

- CoARESF has the following advantages:
  - the first dynamic DSM with coverable fragmented objects & 2-Level of Striping
  - Optimized for High Concurrent accesses

- Theoretical principles illustrated by extensive experimental evaluation
  - atomic consistency, data striping, erasure coding, access to the same files under heavy concurrency, fault-tolerance, reconfiguration

Ongoing and Future Work

- Design Reconfiguration Orchestration Strategies (ROS) for dynamic DSM 

  - **When** to invoke reconfigurations
  - **How** to reconfigure
  - Ensure that the system remains operational despite server failures.
  - Improve performance by replacing older servers with more powerful ones.

- Server Failure Prediction 

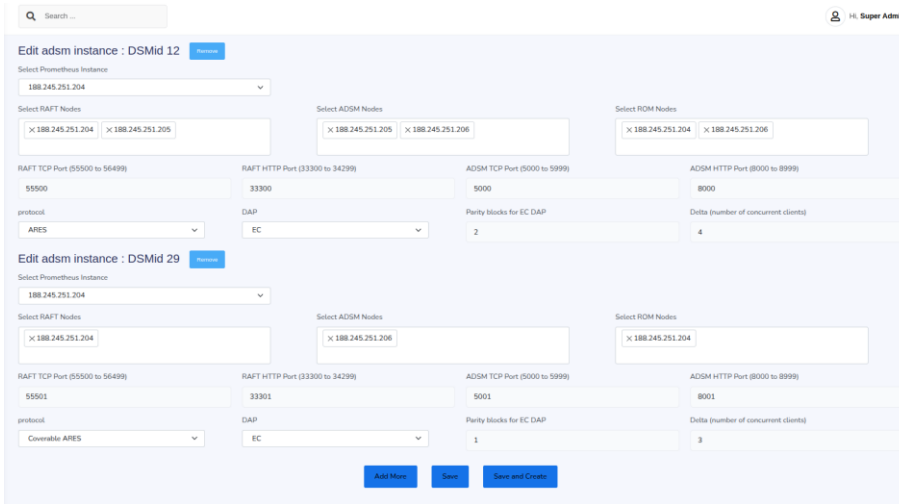
  - Monitor environmental parameters of servers (performance, capacity, availability, and health).
  - Threshold-based approaches for determine when to reconfigure.
  - Predict failures using Machine Learning.

- Develop, Deploy, and Evaluate Web Platform 

  - Deploy and get access to configurations of DSM with ROS support by specifying servers.
  - Manage existing configurations.
  - Get access to existing configuration for reading and writing data objects either through the platform directly or through a third-party application using appropriate security tokens.

- Fully-functional Distributed Storage System with Security Guarantees

- Extensive Experimental Evaluation
  - Compare CoARESF against commercial solutions that employ a striping method.
  - Integrate any commercial solution into CoBFS.



Publications

Journal Articles

1. Ares: Adaptive, reconfigurable, erasure coded, atomic storage. N. Nicolaou, V. Cadambe, N. Prakash, **A. Trigeorgi**, K. M. Konwar, M. Medard, and N. Lynch. *Journal of ACM Trans. Storage* 2022.
2. Boosting Concurrency and Fault-Tolerance for Reconfigurable Shared Large Object. A. F. Anta, C. Georgiou, T. Hadjistasi, N. Nicolaou, E. Stavrakis, and **A. Trigeorgi**. *Under Submission* 2024.

Articles in Conferences

3. Robust and strongly consistent distributed storage systems. **A. Trigeorgi**. In *Proc. of RCIS* 2021.
4. Fragmented Object : Boosting Concurrency of Shared Large Objects. A. Anta, C. Georgiou, N. Nicolaou, T. Hadjistasi, E. Stavrakis, and **A. Trigeorgi**. In *Proc. of SIROCCO* 2021.
5. Fragmented ARES: Dynamic Storage for Large Objects. C. Georgiou, N. Nicolaou, **A. Trigeorgi**. In Proc. of *DISC* 2022. (arXiv:2201.13292)
6. Invited paper: Towards Practical Atomic Distributed Shared Memory: An Experimental Evaluation. **A. Trigeorgi**, N. Nicolaou, C. Georgiou, T. Hadjistasi, E. Stavrakis, V. Cadambe and B. Urgaonkar. In Proc. of SSS 2022.
7. Robust and Consistent Distributed Storage as a Service. **A. Trigeorgi**. In *Proc. of GEC* 2024.
8. Tracing the Latencies of Ares: A DSM Case Study. C. Georgiou, N. Nicolaou, and **A. Trigeorgi**. In *Proc. Of ApPLIED* 2024.
9. Ares II: Tracing the Flaws of a (Storage) God. C. Georgiou, N. Nicolaou, and **A. Trigeorgi**. In *Proc. Of SRDS* 2024.

Acknowledgments

Collaborators:

- Prof. Chryssis Georgiou, University of Cyprus (Supervisor)
- Dr. Nicolas Nicolaou, Algolysis Ltd
- Dr. Efstathios Stavrakis, Algolysis Ltd
- Prof. Antonio Fern´andez Anta, IMDEA Networks
- Dr. Theophanis Hadjistasis, Algolysis Ltd

Examination Committee:

- Prof. Anna Philippou, University of Cyprus
- Assoc. Prof. George Pallis, University of Cyprus
- Prof. Gregory Chockler, University of Surrey, UK
- Assoc. Prof. Herodotos Herodotou, Cyprus University of Technology

Thank you!

For more information you can see the websites of our related projects:

Related Projects

- **PHD IN INDUSTRY**/1222/0121
- **DUAL USE**/0922/0048
- **POST-DOC**/0916/0090
- **EU's NGIAtlantic.eu** cascading grant agreement no. OC4-347

CORRECTNESS

Data Access Primitives (DAPs)

- Operation Ordering: logical tags $\tau =< ts, wid >$
- For a configuration c, any client p may invoke any of the following DAPs:
  - D1. $c.get-tag( ): returns\ a\ tag\ \tau\ \in\ \mathrm{T}$
  - D2. $c.get-data( ): returns\ a\ tag-value\ pair\ (\tau, v) \in \mathrm{T}\ \times\ V$
  - D3. $c.put-data(< \tau, v >)$: the tag $-$ value pair$(\tau, v)\ \in \mathrm{T}\ \times\ V$ as argument
- Property 1: DAP consistency conditions
  - C1: if a put-data(<τ,v>) precedes a get-data/get-tag operation that returns τ', then $\tau' \geq \tau$
  - C2: if a get-data returns <τ',v'>, then there exists put-data(<τ',v'>) that precedes or is concurrent to the get-data operation

DAP-ABD and DAP-EC satify Property 1

Correctness of CoARES

Theorem

CoARES implements a linearizable coverable object, given that the DAPs implemented in any configuration c satisfy DAP Consistency Conditions.

Proof challenges: CoARES satisfies coverability despite any reconfiguration in the system.

- New values are not overwritten (by writes associated with older versions)
- Versions are unique
- Eventually a single version path prevails

Correctness of CoARESF

Theorem

CoARESF implements a linearizable coverable fragmented object.

Proof challenges

- f remains connected and composed of the most recent blocks, despite concurrent read/write and reconfig operations
- Each block may exist in different configurations and be accessed by different DAPs
- Show that fragmented linearizable coverability cannot be violated

From ARES to ARES II

- **Piggy-backing:** Integrating configurations with read/write messages to speed up configuration discovery.
- **Garbage Collection:** Eliminating obsolete configurations for quicker access to the latest data.
- **Data Batching:** A single reconfiguration across multiple objects to enhance efficiency.

Correctness of ARES II

- The correctness of ARES II depends on the correct implementation of both **EC-DAP II** and the **Reconfiguration protocol**.

Correctness of EC-DAP II

Theorem

Revised Property 1 to accommodate the fact that get-data can return a tag associated with either a value from V or ⊥.

Proof Challenges

- **Tag Guarantee:** get-data() always returns a tag ≥ any previous put-data().
- **Value Origin:** Values from **get-data()** come from **put-data()**, the initial value, or ⊥ (due to garbage collection).

Reconfiguration Protocol Properties

Theorem

For completed actions $\pi 1$ and $\pi 2$, where $\pi 1 \rightarrow \pi 2$:

a. **Configuration Uniqueness**
b. **Subsequence**
c. **Sequence Progress**

Proof Challenges

- All processes must have the same config at index k.
- Next configs must point to a higher index than the current one.
- Future actions must start from a config at or above the last finalized config.

Atomicity

Theorem

ARES II guarantees atomicity, given that the DAPs implemented in any configuration c satisfy Property 1 and satisfy the reconfiguration properties.

Proof Challenges

- Ensure get-data returning ⊥ does not prevent the read operation from retrieving a valid non-⊥ value.
- The read operation must continue until it finds a finalized (not GC) config with the highest tag with a non-⊥ value.
- Batching must behave like multiple reconfigurations, maintaining the correctness of read/write operations.

Correctness of EC-DAP II

Theorem

Revised Property 1: DAP consistency conditions

C1: if a put-data(<τ,v>) precedes a get-tag (or get-data) operation that returns τ' (or <t',v'>/ <t',⊥>), then $\tau' \geq \tau$

C2: if a get-data returns <τ',v'> or <t',⊥>, then there exists put-data(<τ',v'>) that precedes or is concurrent to the get-data operation

Proof Challenges

- The tag returned by get-data() is guaranteed to be ≥ the tag from any prior put-data().
- Values returned by get-data() are either from a put-data() operation, the initial object value, or ⊥ (due to garbage collection).

Reconfiguration Protocol Properties

Theorem

For completed actions $\pi 1$ and $\pi 2$, where $\pi 1 \rightarrow \pi 2$:

a. **Configuration Uniqueness**: Configuration sequences in any two processes are identical at any common index i.
b. **Subsequence**: The configuration sequence x observed by $\pi 1$ is a subsequence of the sequence y observed by $\pi 2$, i.e. $\lambda(x) \leq \lambda(y)$.
c. **Sequence Progress**: Finalized configurations progress in order.

Proof Challenges

- Must have the same configuration at index k, regardless of how they received it (propose operation or DAP/read-config)
- If the last configuration has index k, subsequent configurations with index j must have a nextC pointing to a configuration with index $i \geq j+1$.
- If the last finalized configuration has index k, future actions must start from a configuration with index j such that $j \geq k$.

Atomicity

Theorem

ARES II guarantees atomicity, given that the DAPs implemented in any configuration c satisfy Property 1 and satisfy the reconfiguration properties.

Proof Challenges

- Ensure that get-data returning ⊥ does not prevent the read operation from retrieving a valid non-⊥ value.
- The read operation must continue until it finds a finalized configuration that is not garbage collected, yielding the highest tag with a non-⊥ value.
- Batching is indistinguishable from multiple reconfigurations, ensuring the correctness of read/write operations.

EXTRA SLIDES

Algorithm/ System	Data scalability	Data access Concurrency	Consistency guarantees	Versioning	Data Striping	Non-blocking Reconfic
GFS	YES	concurrent appends	relaxed	YES	YES	YES (short downtime)
HDFS	YES	files restrict one writer at a time	atomic (centralized)	NO	YES	YES
Cassandra	YES	YES	tunable (default= eventual)	YES	NO	NO
Dropbox	YES	creates conflicting copies	eventual	YES	YES	N/A
Colossus	YES	concurrent appends	relaxed	YES	YES	YES
Blobseer	YES	YES	atomic (centralized)	YES	YES	YES
Tectonic	YES	files restrict one writer at a time	read-after-write	YES	YES	YES
CoABD	NO	NO	atomic	YES	NO	NO
CoBFS (using ABD)	YES	YES	atomic	YES	YES	NO
LDR	YES	NO	atomic	NO	NO	NO
RAMBO/ DynaStore SMStore/SpSnStore	NO	NO	atomic	NO	NO	YES
AresABD	NO	NO	atomic	NO	NO	YES
AresEC	NO	NO	atomic	NO	YES	YES
CoAresABD	NO	NO	atomic	YES	NO	YES
CoAresEC	NO	NO	atomic	YES	YES	YES
CoAresABDF	**YES**	**YES**	**atomic**	**YES**	**YES**	**YES**
CoAresECF	**YES**	**YES**	**atomic**	**YES**	**YES (2 lvl striping)**	**YES**

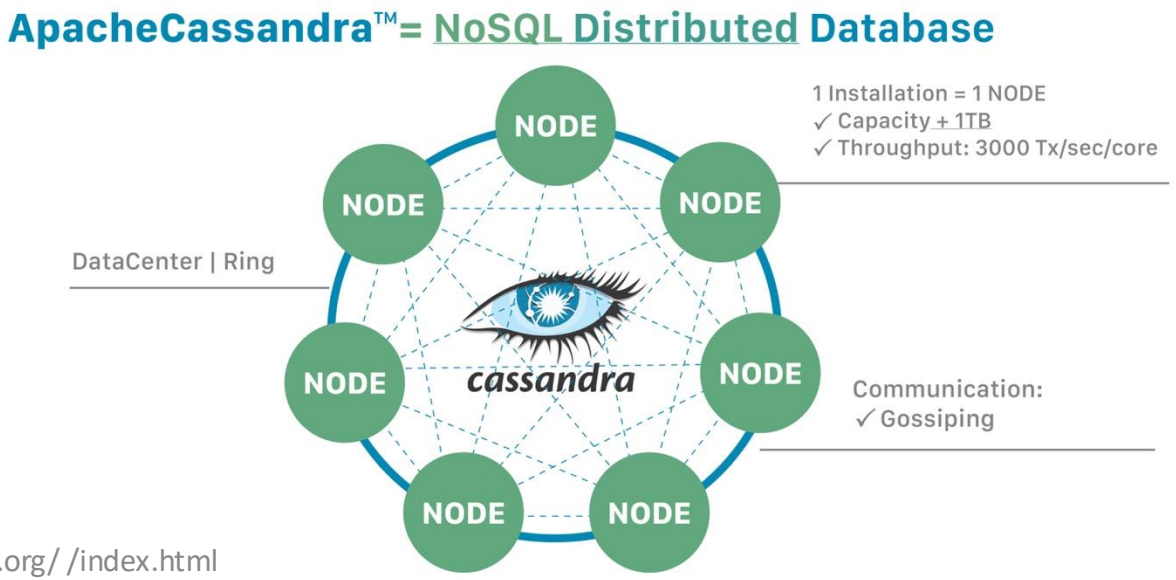
Distributed Storage Systems

System	Data scalability	Data access Concurr.	Consist. guarantees	Versioning	Data Striping	Non-blocking Reconfig.
GFS	YES	concurrent appends	relaxed	YES	YES	YES (short downtime)
Colossus	YES	concurrent appends	relaxed	YES	YES	YES
HDFS	YES	files restrict one writer at a time	atomic centralized	NO	YES	YES
CASSANDRA	YES	YES	tunable (default= eventual)	YES	NO	NO
DROPBOX	YES	creates conflicting copies	eventual	YES	YES	N/A
REDIS	YES	YES	eventual	YES	NO	NO
BLOBSEER	YES	YES	atomic centralized	YES	YES	YES
TECTONIC	YES	files restrict one writer at a time	Read-your- writes	YES	YES	YES

CASSANDRA

- What is Cassandra? a key-value Distributed Database.
- Why Cassandra? availability, high performance, horizontal scalability.
- How it works? Gossip protocol, Peer-to-peer communication in a Ring topology.
- Tunable Consistency = number of nodes to acknowledge an operation (default=Eventually). Can support strong consistency.
- Tunable Replication Factor (# of copies).

We set RF=n, CL=majority (n/2+1) - atomic

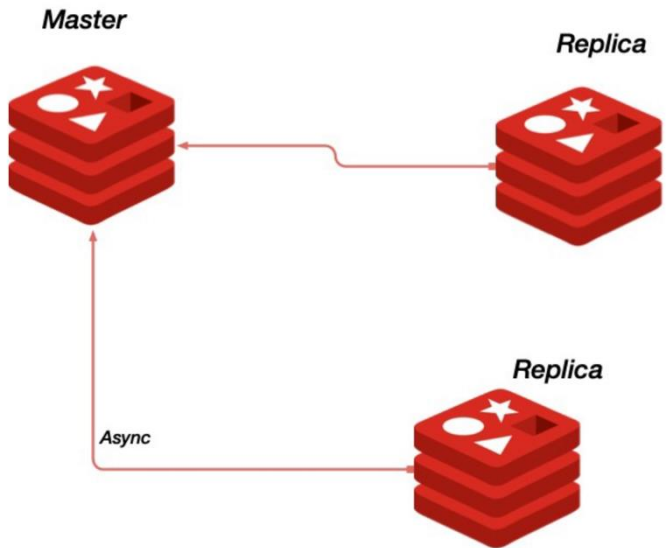


Cassandra, https://cassandra.apache.org/ /index.html

REDIS

- What is Redis? Ultra-fast in-memory key-value store. Used as database, cache, and for simple apps.
- It has memory limitations.
- “WAIT” command for synchronous replication

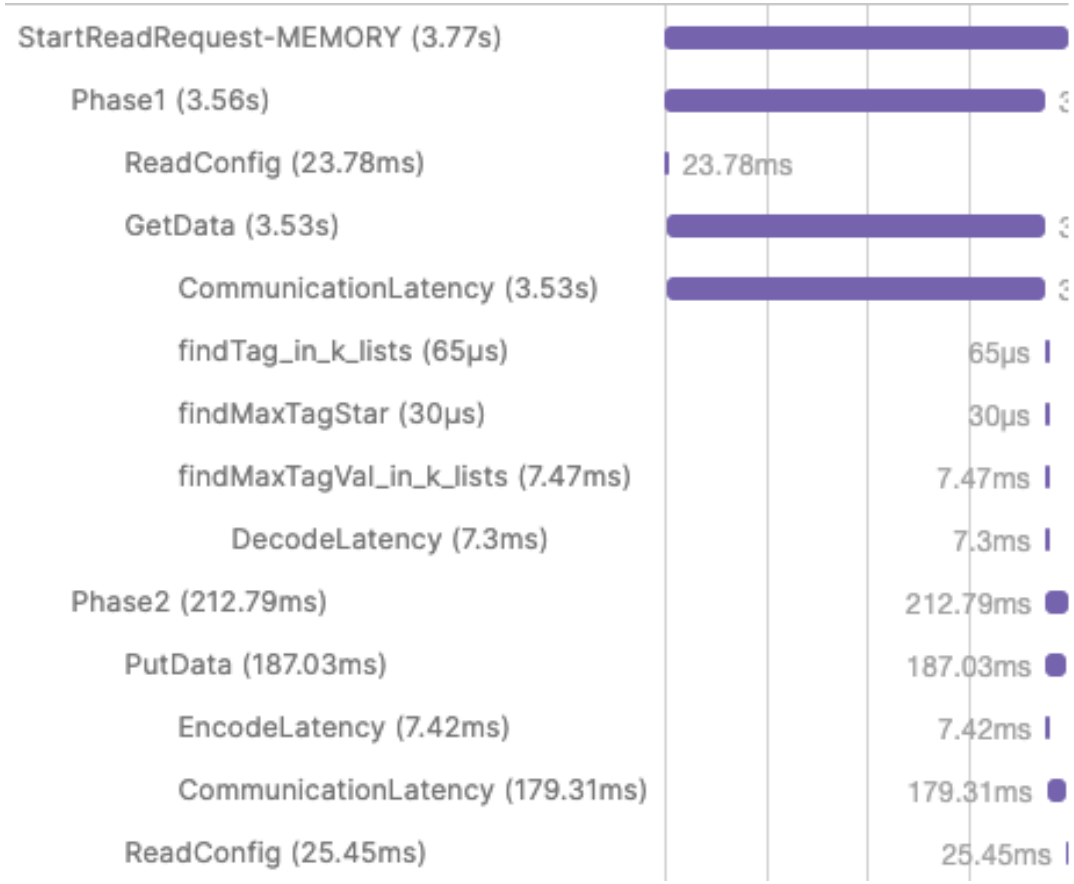
  (We set a majority (n/2 + 1) of waiting write acks).
- It provides Eventual consistency.
- Replication: Master ensures that one or more slaves becomes exact copies of it. Clients can connect to the master or to the slaves. Slaves are read only by default.



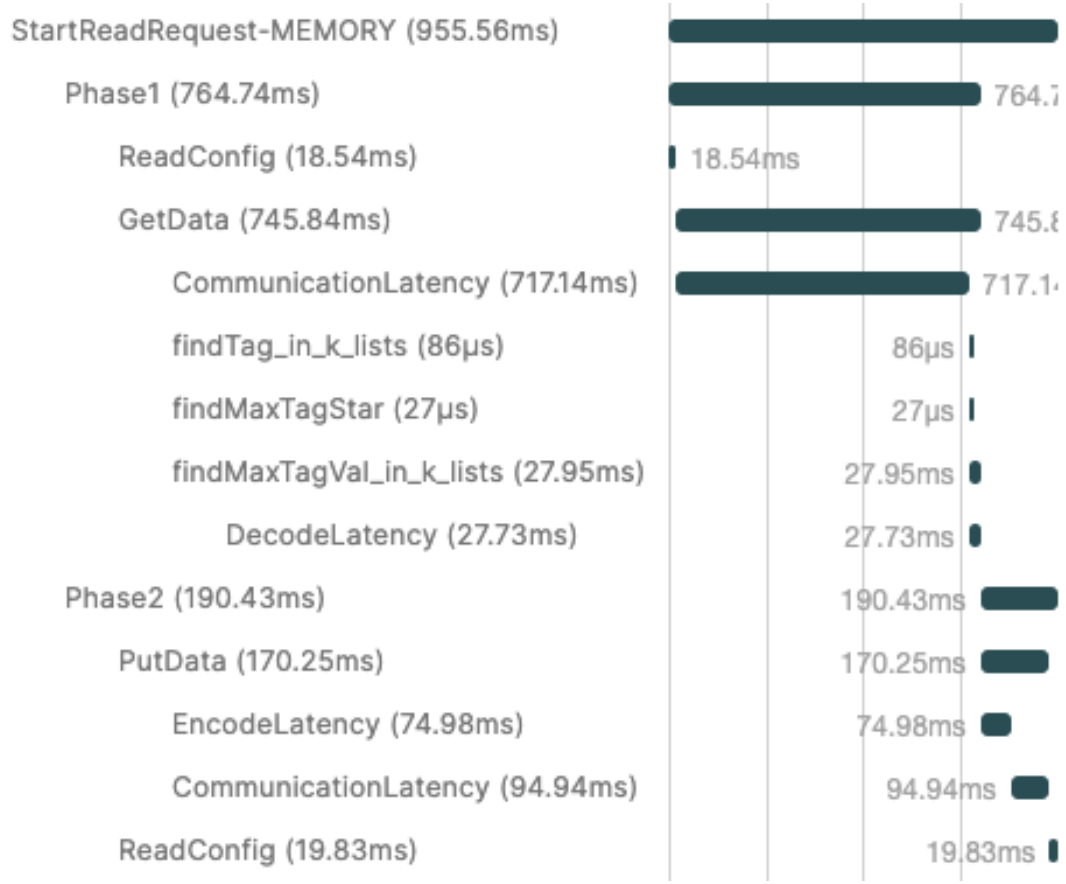
Trade-offs

- **Block size of FM.** data striping performance highly depends on the block size.
- **Parity of EC.** the further increase of the parity (and thus higher fault-tolerance) the larger the latency.
- **Parameter δ of EC.** δ = #writers & each server sends δ + 1 concurrent values in the first phase→ as the #writers increases, the latency also increases

Participation Scalability

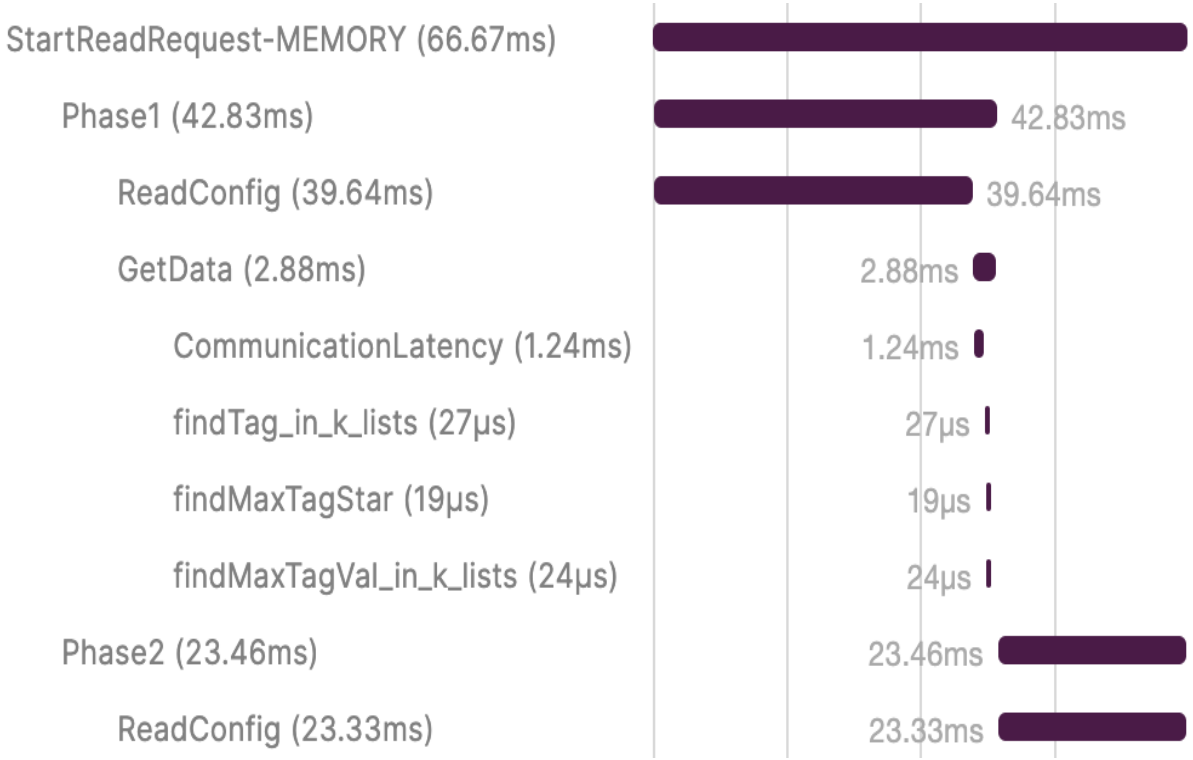


ARESEC, S:3, W:5, R:50, fsize:4MB, Debug Level:DSMM

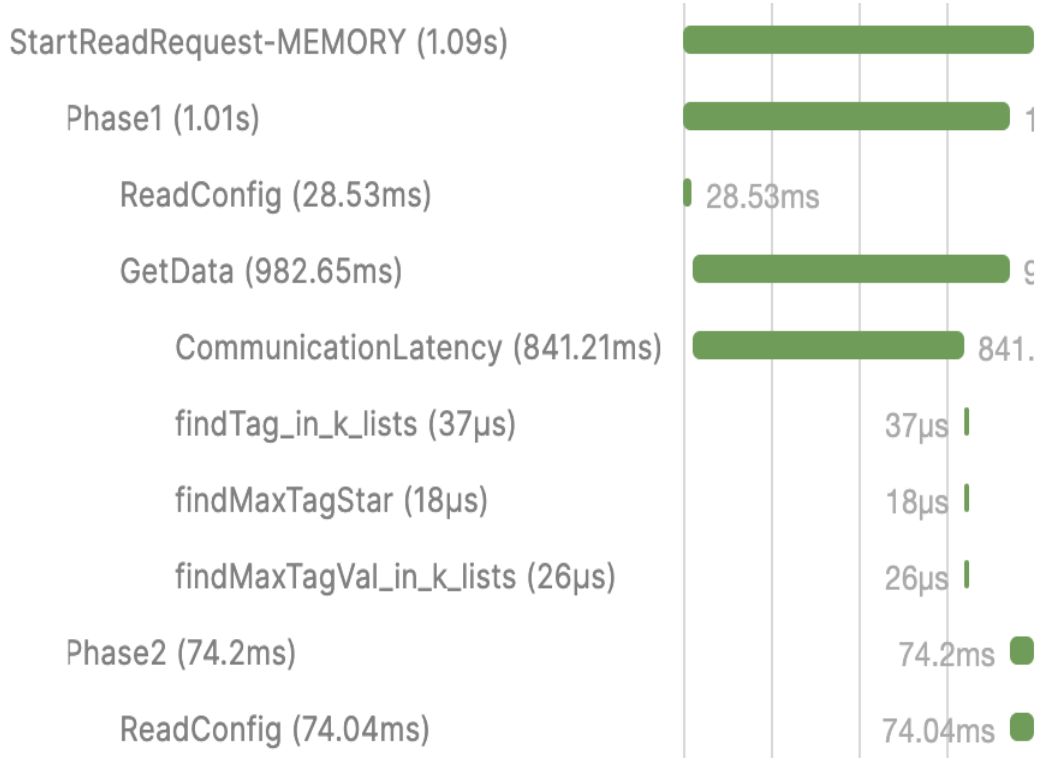


ARESEC, S:11, W:5, R:50, fsize:4MB, Debug Level:DSMM

Block Sizes

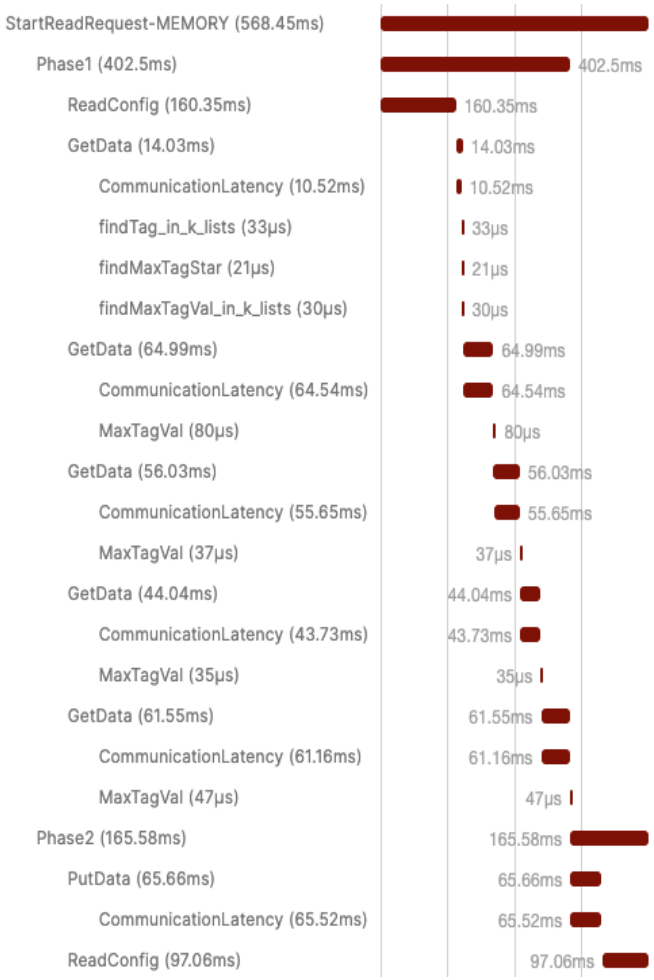


CoAresECF, S:11, W:5, R:5, fsize:512MB, Min/Avg Block Size:2MB, max Block Size:4MB, Debug Level:DSMM

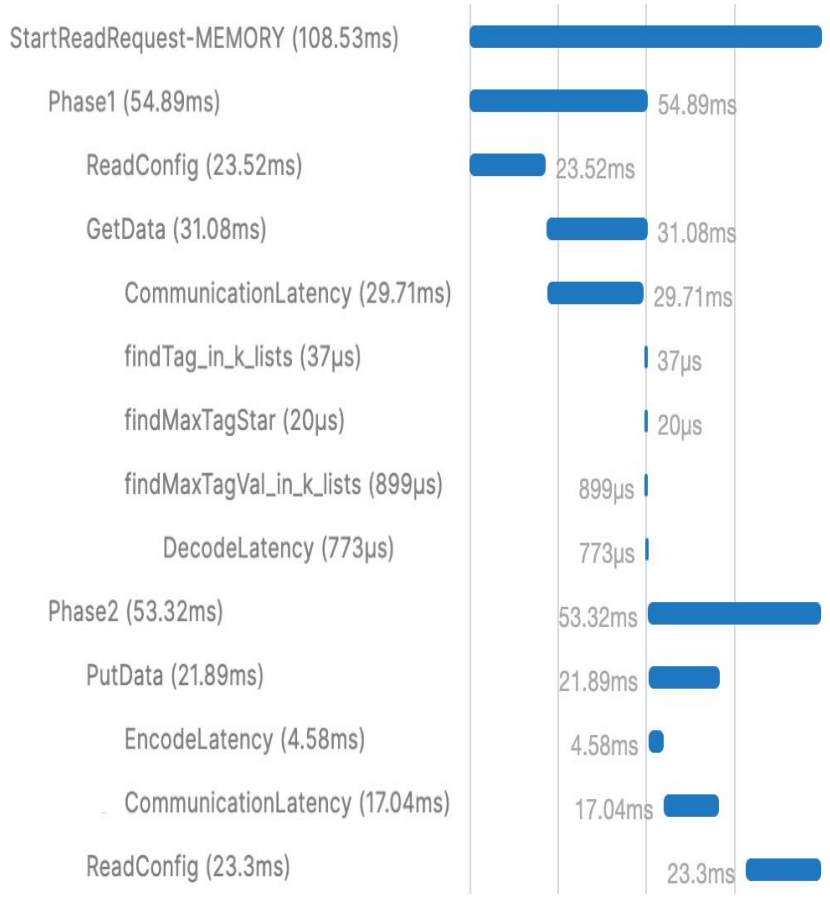


CoAresECF, S:11, W:5, R:5, fsize:512MB, Min/Avg Block Size:64MB, max Block Size:128MB, Debug Level:DSMM

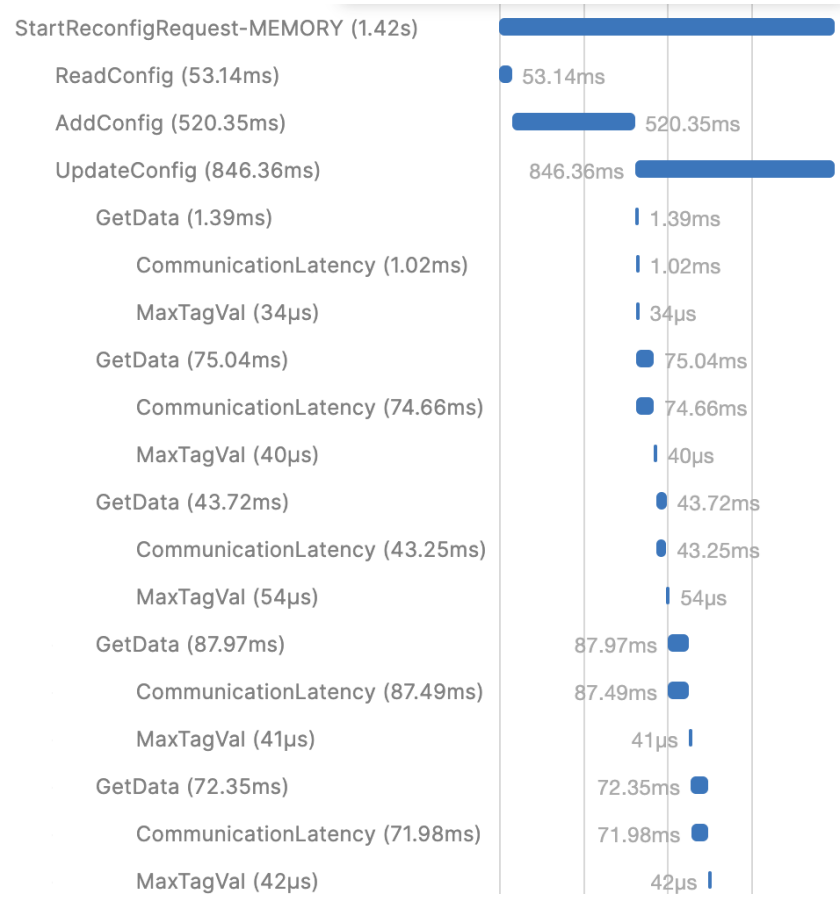
Longevity



CoAresF, S:11, W:5, R:15, G=5, fsize:4MB, Debug Level:DSMM

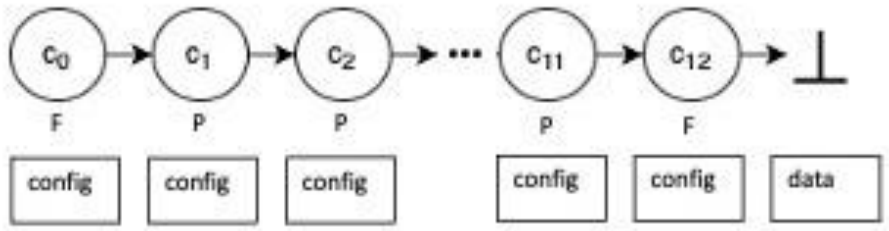


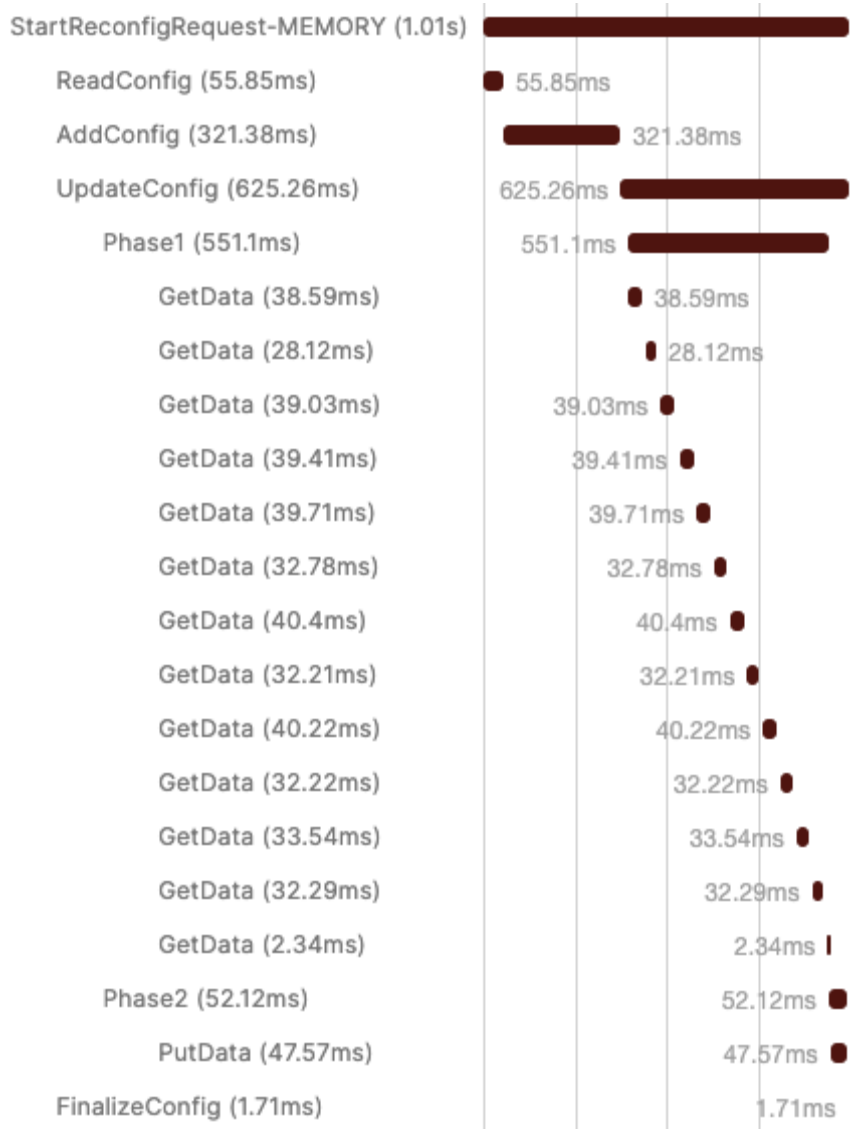
CoAresF, S:11, W:5, R:15, G=1, fsize:4MB, Debug Level:DSMM



CoAresF, S:11, W:5, R:15, G=5, fsize:4MB, Debug Level:DSMM

The Latencies of **read-config** and **get-data**.





The Latencies of **read-config** and **get-data**.