System Implementation and Experimental Findings

Andria Trigeorgi
trigeorgi.andria@ucy.ac.cy

University of Cyprus, Nicosia Cyprus

COLLABORATE Project Final Seminar 2021

Overview

Overview

Comparative table

Algorithm/ System	Data scalability	Data Concurrency	Consistency guarantees	Versioning	Data Stripping
ABD	NO	YES	strong	NO	NO
LDR	YES	YES	strong	NO	NO
CoABD	NO	YES	strong	YES	NO
GFS	YES	concurrent appends	relaxed	YES	YES
HDFS	YES	one writer at a time	strong (centr.)	NO	YES
Dropbox	YES	conflicting copies	eventual	YES	YES
Blobseer	YES	YES	strong (centr.)	YES	YES
CoBFS	YES	YES	strong	YES	YES

Overview

1. Introduction
   - Comparative Table
   - Purpose
2. CoBFS
   - Design
   - Basic Architecture
   - Update and Read operations
   - ARES
3. How we run an experiment
   - How we run an experiment
4. Evaluation
   - CoBFS VS. CoABD - Emulab testbed
   - ARES - Emulab testbed
   - all algorithms - AWS testbed
   - all algorithms - Emulab testbed
5. Conclusions

Our Goal

The development of a Robust and Strongly Consistent DSS while providing highly concurrent access to its users and maintaining strong consistency.

Overview

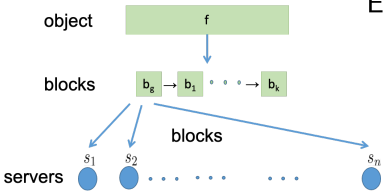
Overview

Design

CoBFS: a Distributed File System with fragmented objects

Design

CoBFS: a Distributed File System with fragmented objects

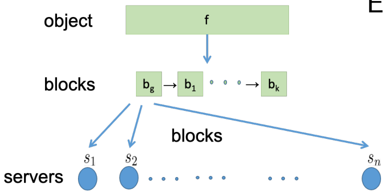


Each object is fragmented into blocks

- Allows big amounts to be distributed all over the servers
- Avoids contention for concurrent accesses to different blocks
- Each block is linearizable and coverable

Design

CoBFS: a Distributed File System with fragmented objects



Each object is fragmented into blocks

- Allows big amounts to be distributed all over the servers
- Avoids contention for concurrent accesses to different blocks
- Each block is linearizable and coverable

- **Fragmented object**: Each f is a *list of blocks*. The first block is the b_{gen}. Each block has the id of its next block.

Overview

Basic Architecture

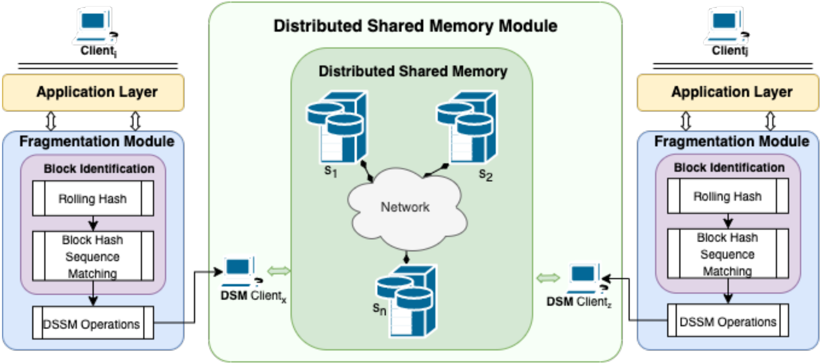
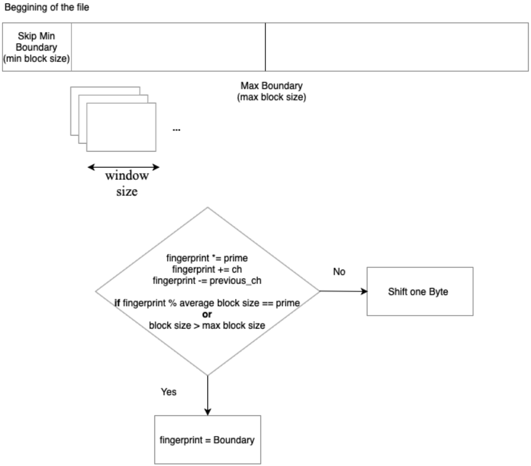


Figure: The basic architecture of CoBFS

Overview

Write/Update operation

- **Block Division:** splits a f into blocks based on its contents, using *rabin fingerprints*.



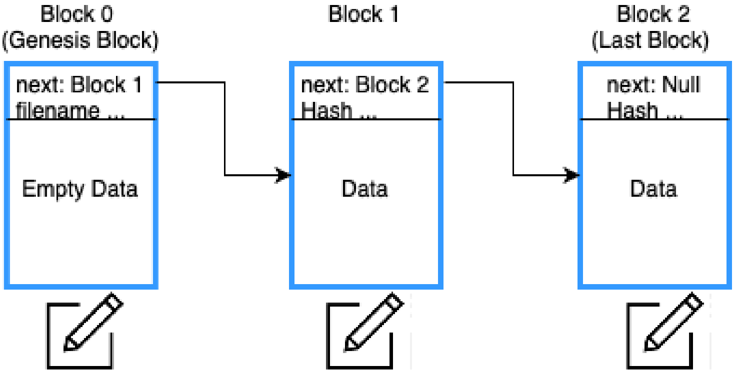
M. O. Rabin, "Fingerprinting by random polynomials," Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., no. TR-15-81. pp. 15–18, 1981.

- **Block Matching:** Use a **string matching** algorithm to find the differences between the new hashes and the old hashes in the form of the statuses: (*i*) equality, (*ii*) modified, (*iii*) inserted, (*iv*) deleted.

- **Block Updates:**
  (*i*) equality, i.e. $hash_i = hash(b_j) \Rightarrow D_i{=}D(b_j)$
  (*ii*) modified $\Rightarrow$ an *update* is performed to modify the $D(b_j)$ to D_i
  (*iii*) inserted $\Rightarrow$ an *update* is performed to create the new blocks
  (*iv*) deleted $\Rightarrow$ is treated as a modification that sets an empty value

Black,P.:Ratcliff pattern recognition.Dictionary of Algorithms and Data Structures(2021)

Read operation



Read Optimization in DSMM: In the first phase, if a server has a smaller tag than the reader, it replies only with its tag. The reader performs the second phase only when it has a smaller tag than the one found in the first phase.

Overview

ARES : Adaptive , Reconfigurable , Erasure coded , Atomic Storage

- ARES is composed of three main components:
  - a reconfiguration protocol
  - a read/write protocol
  - a set of data access primitives (*DAPs*): ABD, EC

1

[1]Nicolaou, N., Cadambe, V., Prakash, N., Trigeorgi, A. et al. (2021). ARES : Adaptive , Reconfigurable , Erasure coded , Atomic Storage, 1(1).

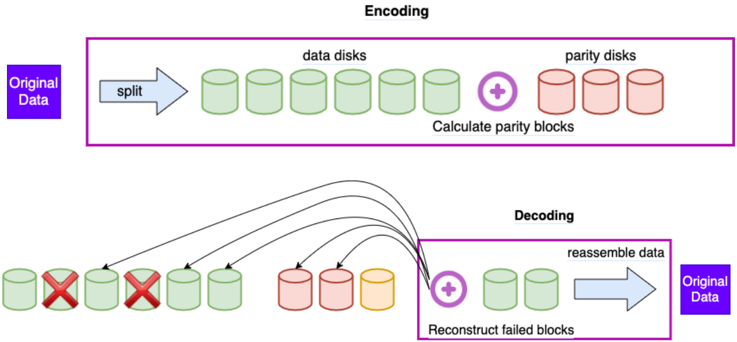
ARES : Adaptive , Reconfigurable , Erasure coded , Atomic Storage

- ARES is composed of three main components:
  - a reconfiguration protocol
  - a read/write protocol
  - a set of data access primitives (*DAPs*): ABD, EC
- Reconfiguration service:
  - mask hosts failures by adding/removing servers
  - switching between storage algorithms (*DAPs*)

[1]

[1] Nicolaou, N., Cadambe, V., Prakash, N., Trigeorgi, A. et al. (2021). ARES : Adaptive , Reconfigurable , Erasure coded , Atomic Storage, 1(1).

Erasure-Coded (EC) approaches



(n, k)-Reed-Solomon code: n=servers, k=data servers, m=parity servers

BUT reads and writes are still applied on the entire object

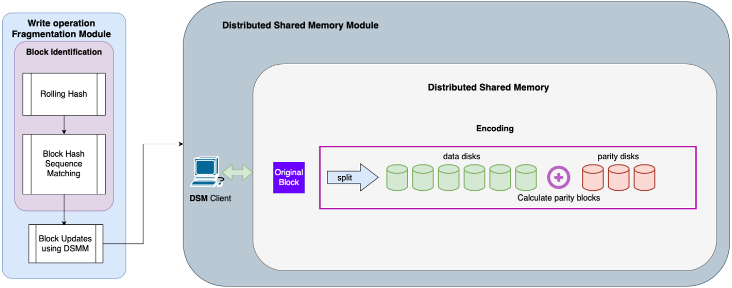


Figure: Update operation

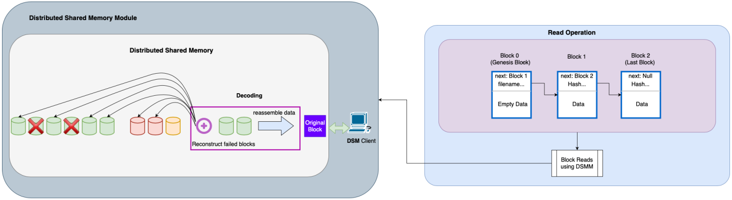


Figure: Read operation

Overview

Overview

How we Run an Experiment

There are two main steps to run an experiment:

Emulab network testbed: https://www.emulab.net/
Ansible: https://www.ansible.com/overview/how-ansible-works/
AWS EC2: https://aws.amazon.com/ec2/

How we Run an Experiment

There are two main steps to run an experiment:

- booting up the *Client* Nodes (either writer or reader) and the *Server* Nodes in an emulation testbed (**Emulab**) or an overlay testbed (**AWS**)
- executing each scenario using **Ansible Playbooks**.

Emulab network testbed: https://www.emulab.net/
Ansible: https://www.ansible.com/overview/how-ansible-works/
AWS EC2: https://aws.amazon.com/ec2/

How we Run an Experiment

There are two main steps to run an experiment:

- booting up the *Client* Nodes (either writer or reader) and the *Server* Nodes in an emulation testbed (**Emulab**) or an overlay testbed (**AWS**)
- executing each scenario using **Ansible Playbooks**.

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

Emulab network testbed: https://www.emulab.net/
Ansible: https://www.ansible.com/overview/how-ansible-works/
AWS EC2: https://aws.amazon.com/ec2/

How we Run an Experiment

There are two main steps to run an experiment:

- booting up the *Client* Nodes (either writer or reader) and the *Server* Nodes in an emulation testbed (**Emulab**) or an overlay testbed (**AWS**)
- executing each scenario using **Ansible Playbooks**.

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

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

Emulab network testbed: https://www.emulab.net/
Ansible: https://www.ansible.com/overview/how-ansible-works/
AWS EC2: https://aws.amazon.com/ec2/

How we Run an Experiment

There are two main steps to run an experiment:

- booting up the *Client* Nodes (either writer or reader) and the *Server* Nodes in an emulation testbed (**Emulab**) or an overlay testbed (**AWS**)
- executing each scenario using **Ansible Playbooks**.

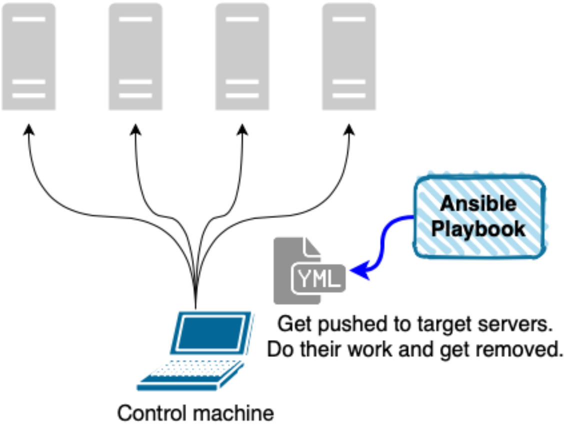
Emulab: a network testbed with tunable and controlled environmental parameters.

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

Ansible: a tool to automate different IT tasks.

Emulab network testbed: https://www.emulab.net/
Ansible: https://www.ansible.com/overview/how-ansible-works/
AWS EC2: https://aws.amazon.com/ec2/

Ansible


Ansible
Playbook
YML
Get pushed to target servers.
Do their work and get removed.
Control machine

Important!

To access a VM node through ssh, it needs a public IP!

Important!

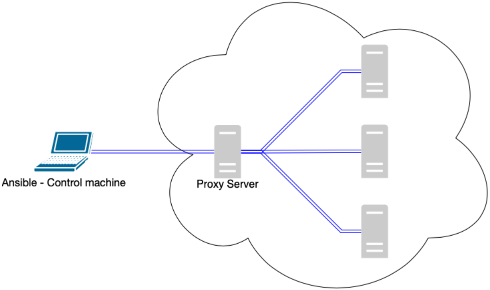
To access a VM node through ssh, it needs a public IP!

Routable IPs are a limited resource!

Important!

To access a VM node through ssh, it needs a public IP!

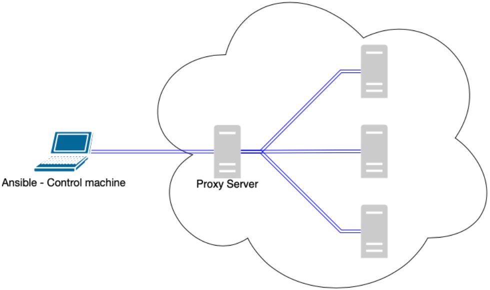
Routable IPs are a limited resource!



Important!

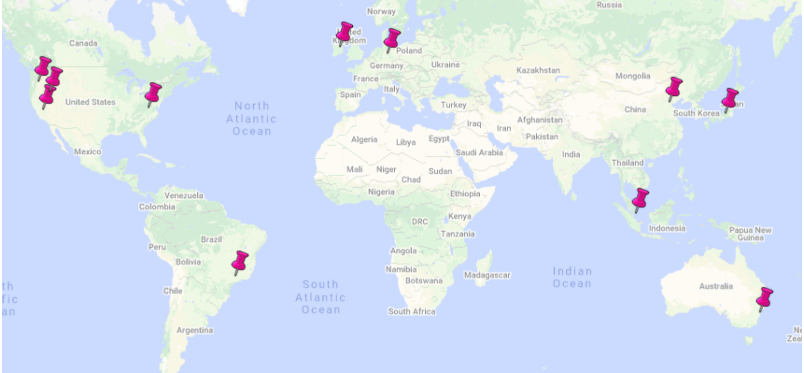
To access a VM node through ssh, it needs a public IP!

Routable IPs are a limited resource!



Increase the limit of the number of ssh connections on the proxy server (update the file "/etc/ssh/sshd_config")

AWS Global Map



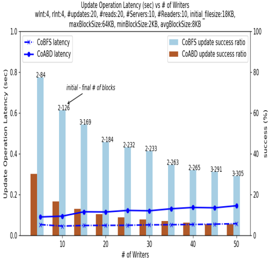
Overview

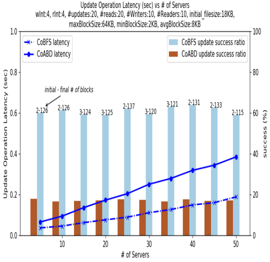
Overview

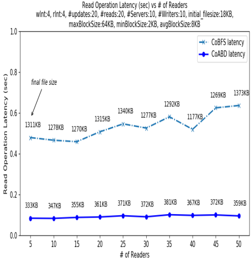
Types of Scenarios:

- **Performance VS. Scalability:** examine performance as the number of service participants increases
- **Performance VS. File Size:** examine performance when using different initial file sizes
- **Performance VS. Block Size**: examine performance under different block sizes (CoBFS only)

Scalability results for algorithms CoABD and CoBFS

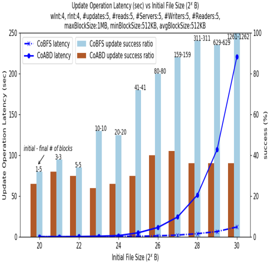


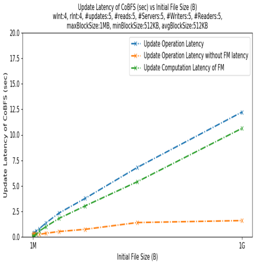


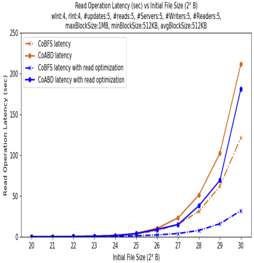


- As each writer has to update only the affected blocks, the update operation latency in CoBFS is always smaller
- Concurrency: As the number of writers increases (hence concurrency), the number of unsuccessful updates in CoABD is greater.
- the higher successful ratio in CoBFS provides more data and hence CoBFS read is slower

File Size results for algorithms CoABD and CoBFS

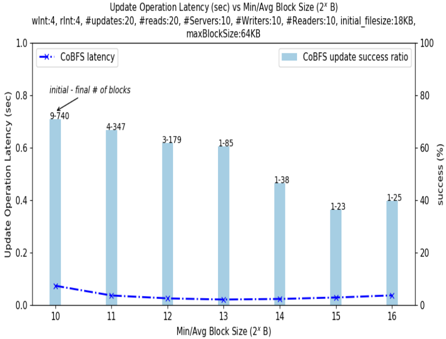


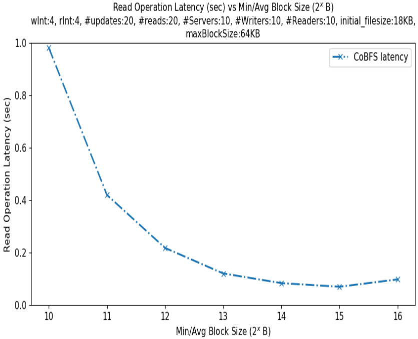




- the update latency of CoBFS remains at extremely low levels, although the file size increases.
- a read optimization decreases significantly the CoBFS read latency, since it is more probable for a reader to already have the last version of some blocks.

Block Size results for CoBFS algorithm





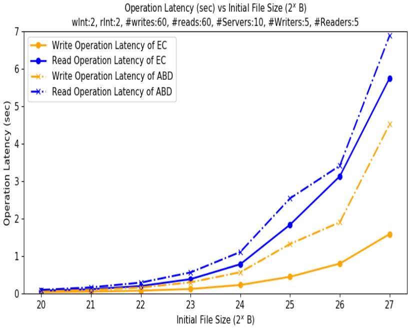
- further increase of b_{size} forces the decrease of the CoBFS latencies
- Concurrency: with a larger number of blocks, the probability of two writes to collide decreases. ⇒ better success rate

Overview

Types of Scenarios:

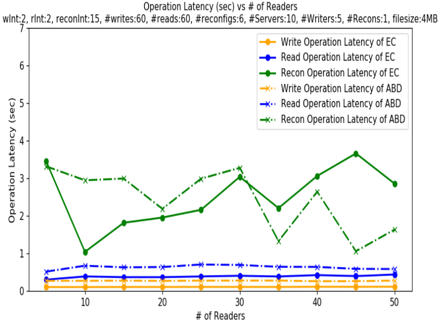
- **Performance VS. File Size:** evaluate how the read and write latencies are affected by the size of the shared object.
- **Performance VS. Scalability of Readers:** compare the read and write latency of the system with two different storage algorithms, while the readers increase.
- **Changing Reconfigurations (Emulab):** In this scenario, we evaluate how the read and write latencies are affected when increasing the number of readers, while also changing the storage algorithm.
- **Performance VS.** k **(EC only):** examine the read and write latencies with different numbers of k (parameter of Reed-Solomon)

File Size results for ARES algorithm



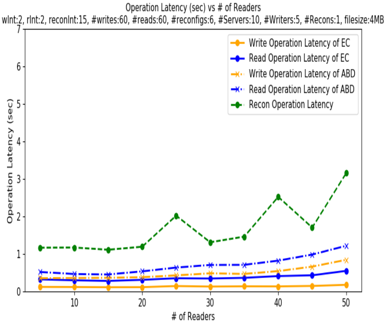
- the read and write latencies of both storage algorithms remain in low levels until 16 MB
- the write operation of EC algorithm is the faster
- the larger messages sent by ABD result in slower read operations

Reader Scalability results for ARES algorithm

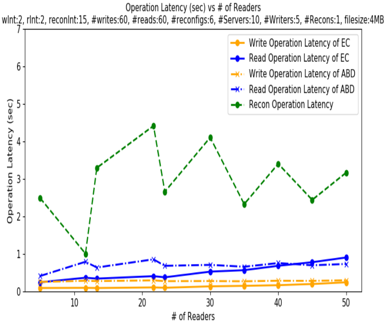


- the reduced message size of read and write operation in EC keep their latencies lower than the coresponding latencies of ABD

Changing Reconfigurations results for ARES algorithm



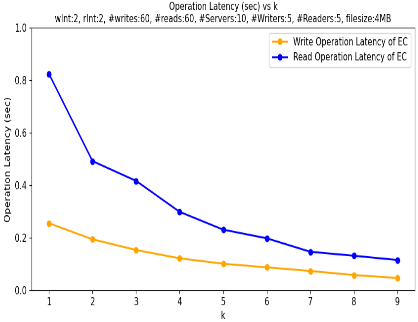
(i)



(ii)

- (i) the reconfigurer chooses randomly between the two storage algorithms
- (ii) the reconfigurer switches between the two storage algorithms
- our choice of k (=parity servers) minimizes the coded fragment size but introduces bigger quorums and thus larger communication overhead. $\Rightarrow$ in smaller file sizes, the ARES may not benefit so much from the coding
- the reconfiguration delays is higher than the delays of all other operations.

k Scalability results for ARES algorithm



- small k (=smaller number of data fragments) $\Rightarrow$ bigger sizes of the fragments and higher redundancy.
- The write latency seems to be less affected by the number of k since the encoding is considerably faster as it requires less computation.

Overview

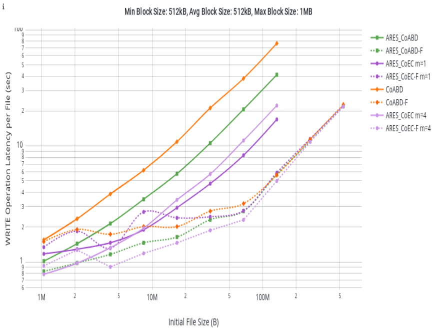
Types of Scenarios:

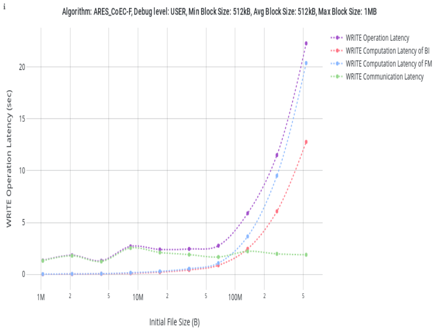
- **Performance VS. Initial File Sizes:** examine performance when using different initial file sizes

- **Performance VS. Scalability of nodes under concurrency:** examine performance as the number of service participants increases
  - $|R|$ and $|W|$: [5, 10,15, 20, 25], $|S|$: [3, 5, 7, 9, 11].
  - parities: [1, 2, 3, 4, 5]
  - the clients and servers are distributed in a round-robin fashion.
  - we calculate all possible combinations of readers, writers and servers where the number of readers or writers is kept to 5.

- **Performance VS. Block Sizes:** examine performance under different block sizes (only for algorithms use the FM module)

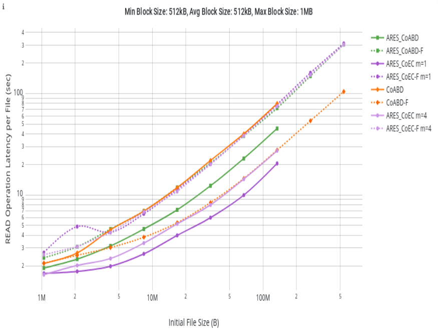
File Size results

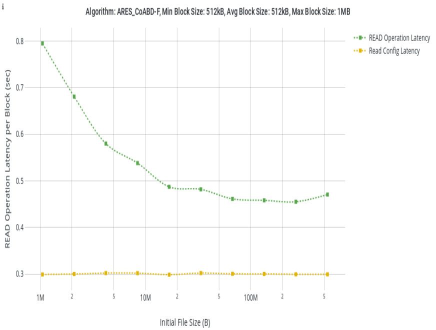




- the update latency of fragmented algorithms achieve significantly smaller write latency, when the file size increases.
- the BI computation latency contributes significantly to the increase of fragmented algorithms' update latency.

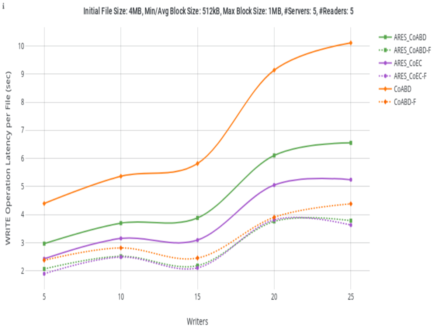
File Size results

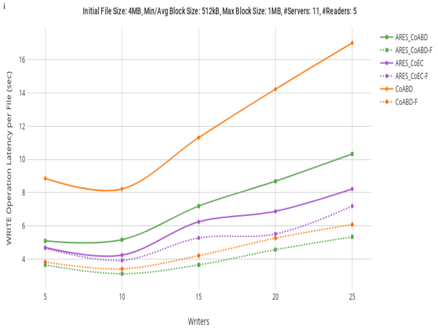




- the read latency of CoABD-F is much smaller than of COABD.
- the ARES-F client has a stable overhead (read-config) for each block request of file update operation.

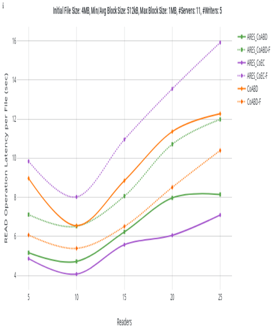
Scalability Results

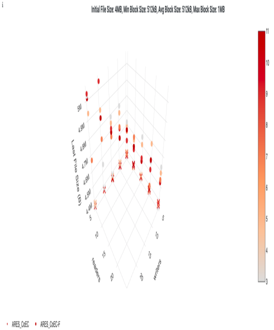


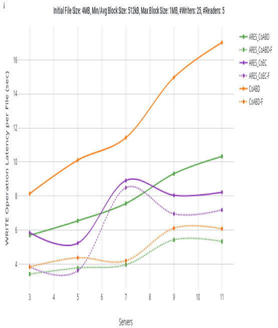


- the write latency of ARESCOEC is the lowest among non-fragmented algorithms because of the striping level.
- the ARES client has a stable overhead (read-config) for each block request.
- the fragmented algorithms perform significantly better write latency.

Scalability Results

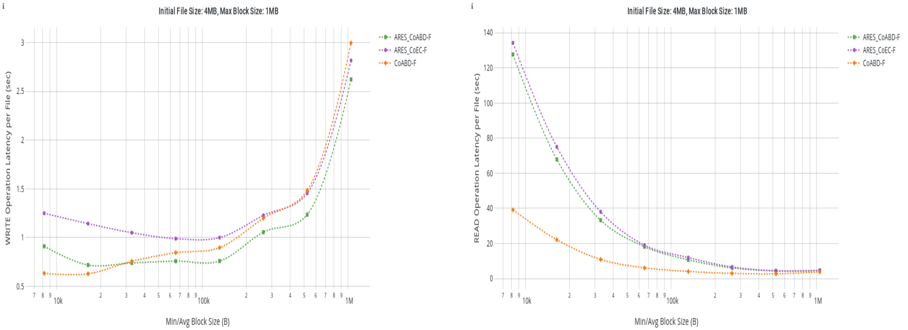






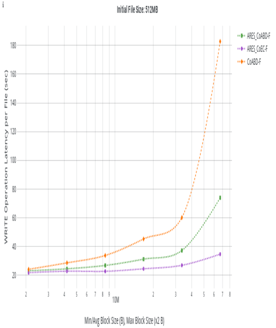
- due to the block allocation strategy in fragment algorithms, more data are successfully written ⇒ slower ARES read operation
- the file size in non-fragmented algorithms stays almost unchanged as the number of servers increases since the cross marks are not widely spread.

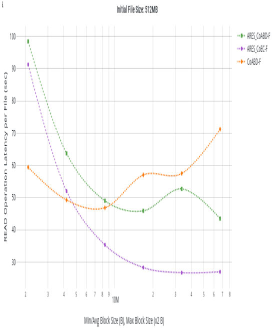
Min/Avg Block Sizes results

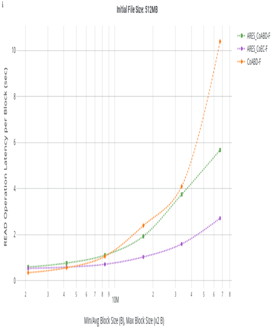


- larger min/avg block sizes are used ⇒ the update latency reaches its highest values since larger blocks need to be transferred.
- too small min/avg block sizes ⇒ more new blocks during update operations ⇒ more update block operations, and hence slightly higher update latency.
- smaller block sizes ⇒ more read block operations to obtain the file's value.

Min/Avg/Max Block Sizes' results







- all the algorithms achieve the maximal update latency as the block size gets larger.
- a larger block needs more time to be updated in the shared memory level.

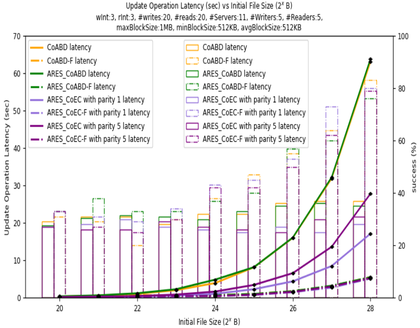
Overview

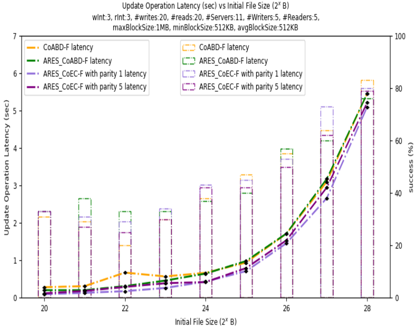
Types of Scenarios:

- **Performance VS. Initial File Sizes:** examine performance when using different initial file sizes
- **Performance VS. Scalability of nodes under concurrency:** examine performance as the number of service participants increases
- **Performance VS. Block Sizes:** examine performance under different block sizes (only for algorithms use the FM module)
- **Changing Reconfigurations:** In this scenario, we evaluate how the read and write latencies are affected when increasing the number of readers/writers, while changing the storage algorithm and the reconfigurer chooses randomly the number of servers between [3, 5, 7, 9, 11].

parities: 3 servers: 1, 5 servers: 2, 7 servers: 3, 9 servers: 4, 11 servers: 5

File Size results

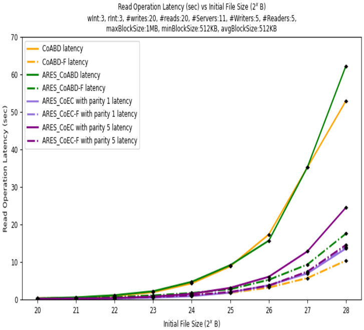


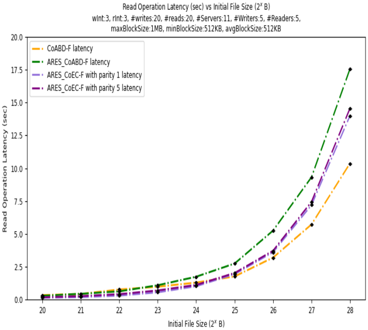


only Fragmented Algorithms

- the update latency of fragmented algorithms remains at extremely low levels, although the file size increases.
- successful file updates achieved by fragmented algorithms are significantly higher (the probability of two writes to collide on a single block decreases as the file size increases)

File Size results

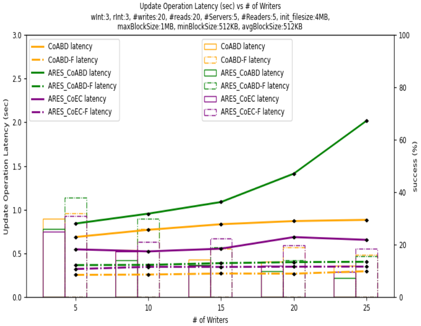


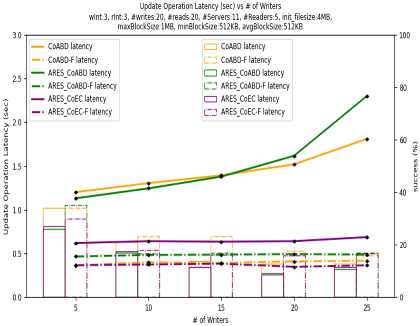


only Fragmented Algorithms

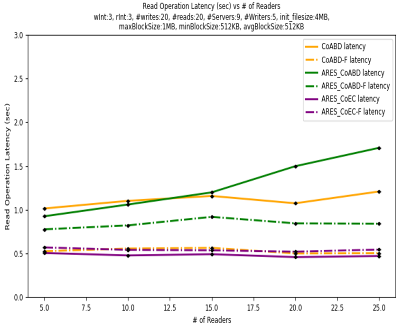
- the fragmented algorithms has lower read latency.

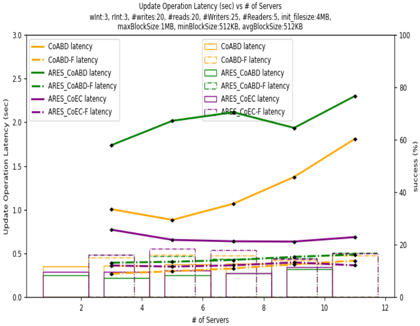
Scalability Results



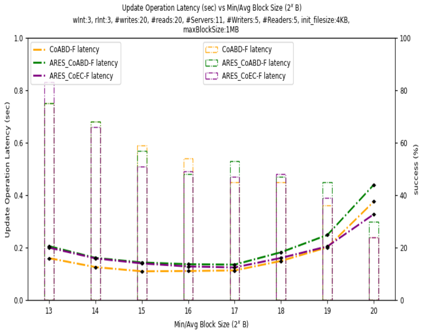


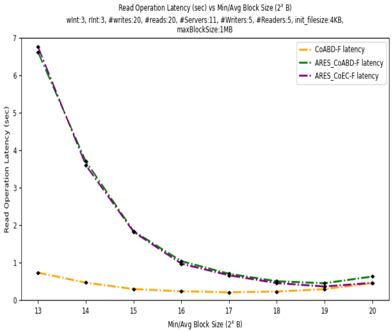
Scalability Results



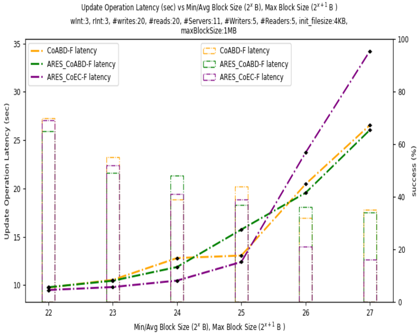


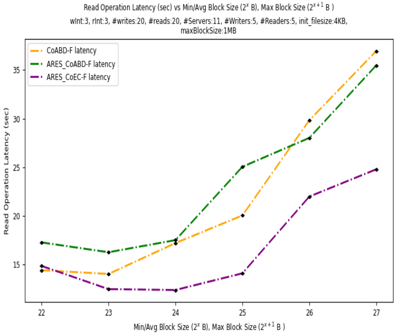
Min/Avg Block Sizes results



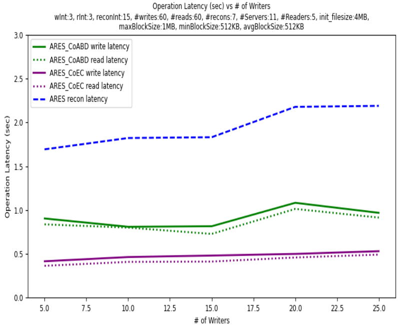


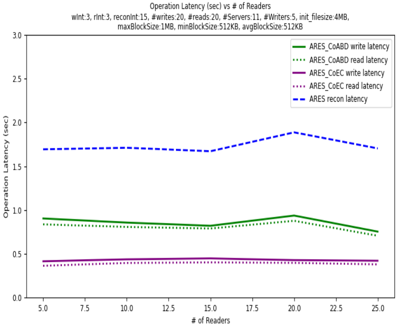
Min/Avg/Max Block Sizes' results



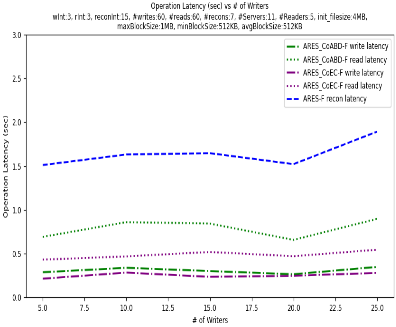


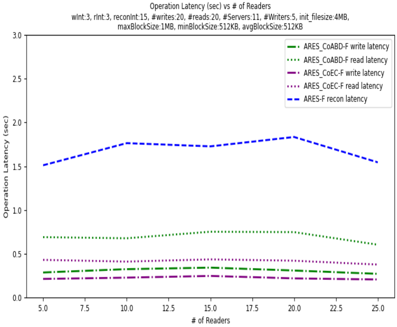
Changing Reconfigurations results - Non Fragmented ARES





Changing Reconfigurations results - Fragmented ARES





Overview

Trade-offs and Conclusions

Block size of FM. trade-off between smaller blocks in order to improve the concurrency and the cost of reading these blocks.
Parity of EC. trade-off between operation latency and fault-tolerance in the system: the further increase of the parity (and thus higher fault-tolerance) the larger the latency.

Our algorithm, CoBFS , has the following advantages:

- High Concurrent accesses
- Strong consistency
- Large file sizes (tested up to 1GB file)

Thanks for your attention! Any questions?

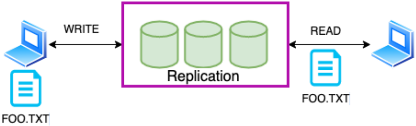
Overview

- Challenges for Distributed Shared Storage Systems
- Steps on Emulab
- Execute the Scenarios using Ansible

Challenges for Distributed Shared Storage Systems

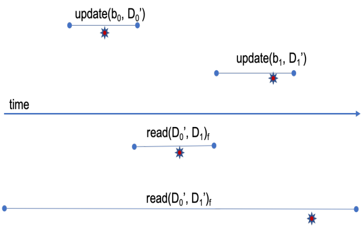
- **Data scalability**
- **Data survivability** + **System availability** $\Longrightarrow$ **Data replication**
- **Storage efficiency**
- **Communication overhead**
- **Concurrent access**
- **Consistency Semantics**

  Linearizability: if $t_{op1} < t_{op2}$, then the $op1$ must occur before $op2$ in the sequence seen by all processes.

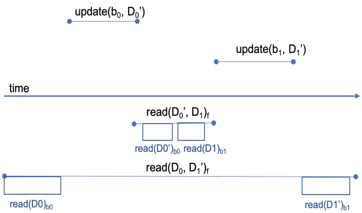


M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," ACM Trans. Program. Lang. Syst., vol. 12, no. 3, pp. 463–492, 1990.

Fragmented Linearizability

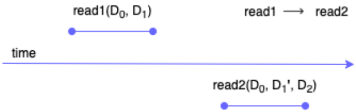


(a) Linearizability on the whole object



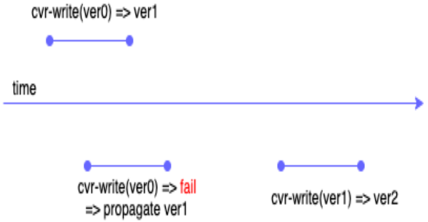
(b) Fragmented Linearizability

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



M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," ACM Trans. Program. Lang.

Versioning - Coverability



Coverability guarantees that an update succeeds when the writer has the latest version of the object before updating it. Otherwise, an update becomes a read.

The selected emulation to ensure consistency in our system is the **coverable version of MWMR ABD (CoABD)**.

C. G. Nicolas Nicolaou, Antonio Fernández Anta, "Cover-ability: Consistent versioning in asynchronous, fail-prone, message-passing environments."

Overview

- Challenges for Distributed Shared Storage Systems
- Steps on Emulab
- Execute the Scenarios using Ansible

3 Node Types

writer $w \in W$: a client that dispatches write requests to servers.

reader $r \in R$: a client that dispatches read requests to servers.

server $s \in S$:listens for requests & maintains the object replicas.

Performance metric

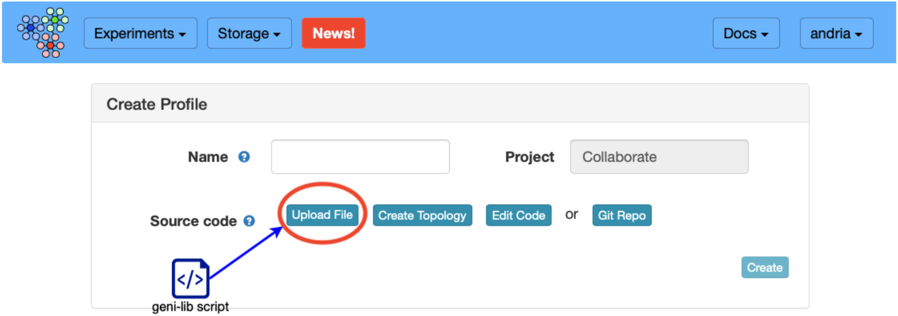
Operation latency: the time it takes for a write/read operation to complete (from the client's point of view)

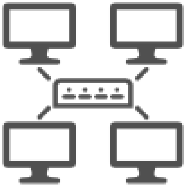
Scenario

examine the operation latency as the number of writers increases.

$|W|$ in the set $\{5, 10, 15, 20, 25, 30, 35, 40, 45, 50\}$
$|R|, |S| = 10$

Create Profile - Upload a geni-lib script in *Python*

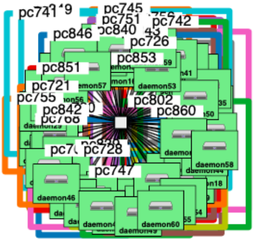




Hardware

Software

Hardware & Software



a **routable control ip** on the Proxy Server

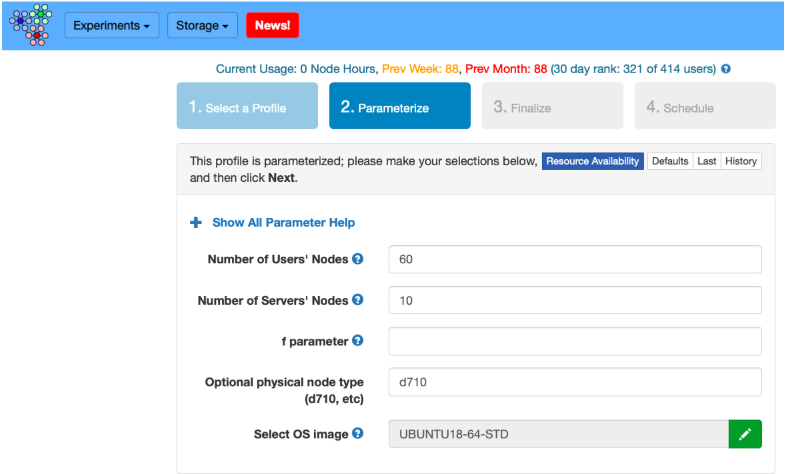
Parameters:
OS: 'UBUNTU 18.04'
Hardware Type: d710 with two 2.4 GHz 64-bit 8-Core E5-2630 "Haswell" processors and 64 GB RAM.

Tunable Parameters

Default Traffic shaping parameters

100Mb bandwidth on VMs, and no delay or packet loss.

User-specified parameters



Important!

To access a VM node through ssh, it needs a public IP!

Important!

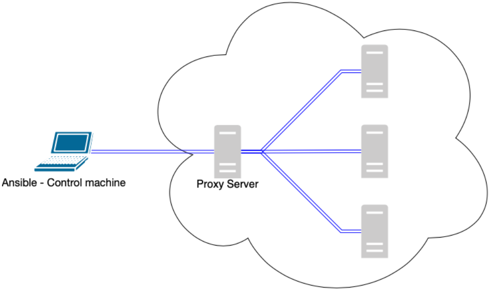
To access a VM node through ssh, it needs a public IP!

Routable IPs are a limited resource!

Important!

To access a VM node through ssh, it needs a public IP!

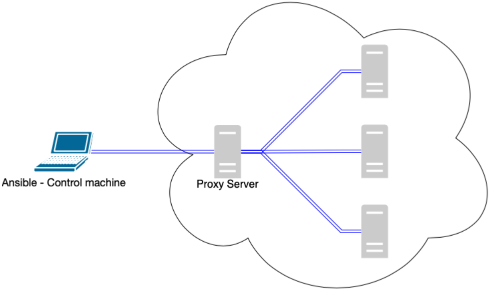
Routable IPs are a limited resource!



Important!

To access a VM node through ssh, it needs a public IP!

Routable IPs are a limited resource!



Increase the limit of the number of ssh connections on the proxy server (update the file "/etc/ssh/sshd_config")

Overview

- Challenges for Distributed Shared Storage Systems
- Steps on Emulab
- Execute the Scenarios using Ansible

Create a config file with the remote hosts

```
[bastion_server]
server1.emulabTest1.collaborate.emulab.net  ansible_user=andria

[servers]
server[2:10]

[servers:vars]
ansible_user=andria
ansible_port=22
ansible_ssh_common_args='-o ProxyCommand="ssh -q -W %h:%p
andria@server1.emulabTest1.collaborate.emulab.net"'

[writers]
daemon[1:50]

[writers:vars]
ansible_user=andria
ansible_port=22
ansible_ssh_common_args='-o ProxyCommand="ssh -q -W %h:%p
andria@server1.emulabTest1.collaborate.emulab.net"'

[readers]
daemon[51:60]

[readers:vars]
ansible_user=andria
ansible_port=22
ansible_ssh_common_args='-o ProxyCommand="ssh -q -W %h:%p
andria@server1.emulabTest1.collaborate.emulab.net"'
```

Playbooks in Sequence

Playbook 1: Stop and Start all the nodes again with the new parameters.

Playbooks in Sequence

Playbook 1: Stop and Start all the nodes again with the new parameters.

Playbook 2: Run the Baseline phase where all the nodes will be notified of the file.

Playbooks in Sequence

Playbook 1: Stop and Start all the nodes again with the new parameters.

Playbook 2: Run the Baseline phase where all the nodes will be notified of the file.

Playbook 3: Readers and writers run a specific number of operations.

Playbooks in Sequence

Playbook 1: Stop and Start all the nodes again with the new parameters.

Playbook 2: Run the Baseline phase where all the nodes will be notified of the file.

Playbook 3: Readers and writers run a specific number of operations.

Playbook 4: Wait until the shell command of previous phase is completed for all clients.

Playbooks in Sequence

Playbook 1: Stop and Start all the nodes again with the new parameters.

Playbook 2: Run the Baseline phase where all the nodes will be notified of the file.

Playbook 3: Readers and writers run a specific number of operations.

Playbook 4: Wait until the shell command of previous phase is completed for all clients.

Playbook 5: Execute a read operation to read the final file.

Playbooks in Sequence

Playbook 1: Stop and Start all the nodes again with the new parameters.

Playbook 2: Run the Baseline phase where all the nodes will be notified of the file.

Playbook 3: Readers and writers run a specific number of operations.

Playbook 4: Wait until the shell command of previous phase is completed for all clients.

Playbook 5: Execute a read operation to read the final file.

Playbook 6: Fetch logs.