COLLABORATE
CONSISTENT & ROBUST DISTRIBUTED STORAGE

# Let's Work Together:
## Building a Robust, Consistent, and Efficient Distributed Shared Storage System for Large Data Objects that Promotes Collaboration

Presented by: **Nicolas Nicolaou**

This work is supported by the Cyprus Research and Innovation Foundation under the grant agreement POSTDOC/0916/0090.

# In a nutshell!

- Distributed Storage – Problem Statement
- Fragmentation: How to handle Large Objects
  - Blocks
  - Erasure Coding
  - Hybrid Solutions
- Reconfiguration: dealing with failures
- DriveNest: a service to predict imminent node failures

# What is a Distributed Storage System?

- How to share data **robustly** in a **message-passing** system?

Dropbox

OneDrive

Google Drive

Nextcloud

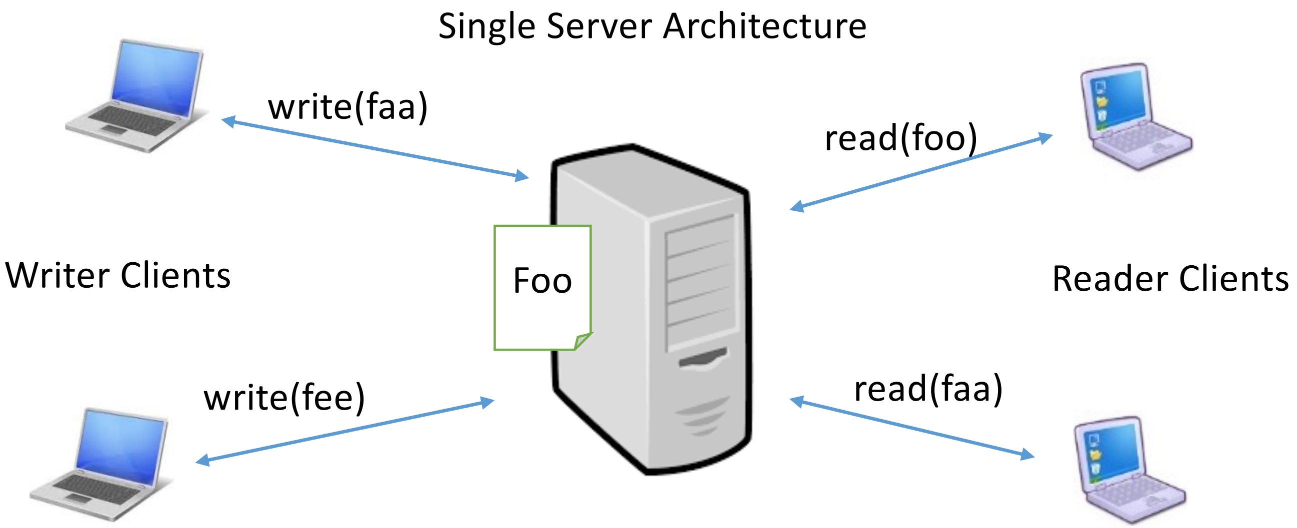
iCloud

Commercial Solutions:

- Cannot provide "sufficient guarantees" when shared objects are accesses concurrently
- Often rely on centralized solutions to enable collaboration
- Not offering "suitable guarantees" for application design

# What is a Distributed Storage System

- How to share data **robustly** in a **message-passing** system?

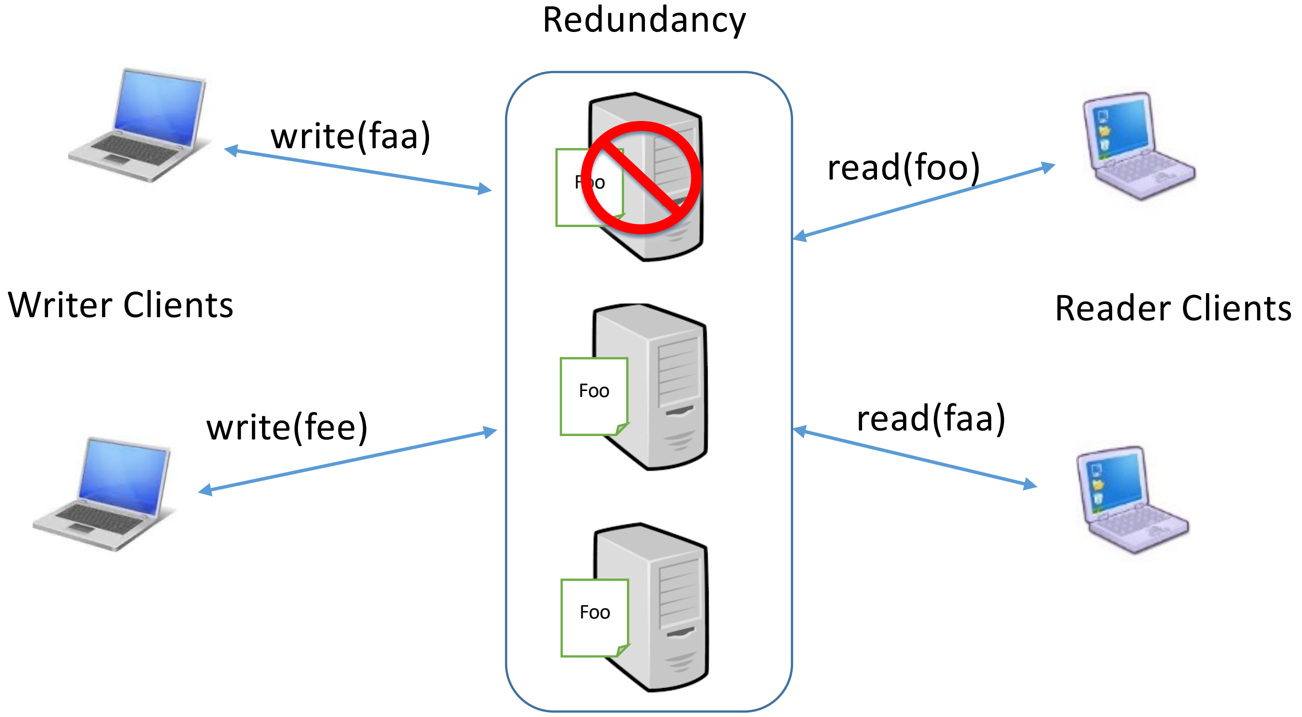



Pros:
- Easy to implement
- Strong consistency as the single server imposes the order of operations

Cons:
- Performance Bottleneck
- Single Point of Failure

# What is a Distributed Storage System

- How to share data **robustly** in a **message-passing** system?



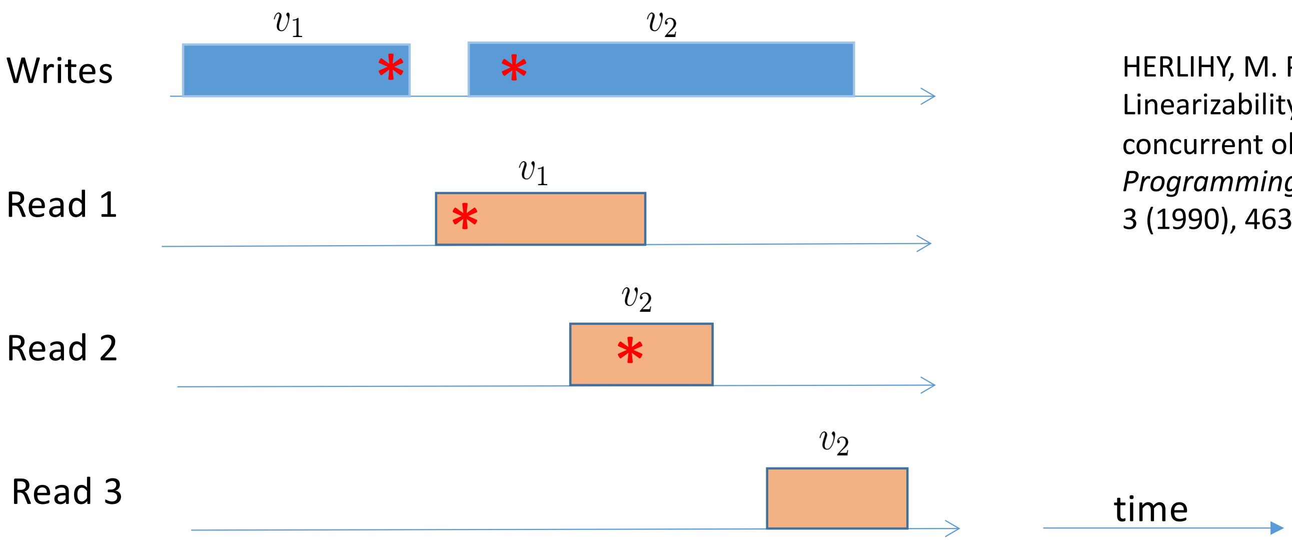

Shared read/write storage object

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

- Availability + Survivability => use redundancy
- Asynchrony + Redundancy => concurrent operations
- Behavior of concurrent operations => consistency semantics
  - Safety, Regularity, Atomicity [Lamport86]

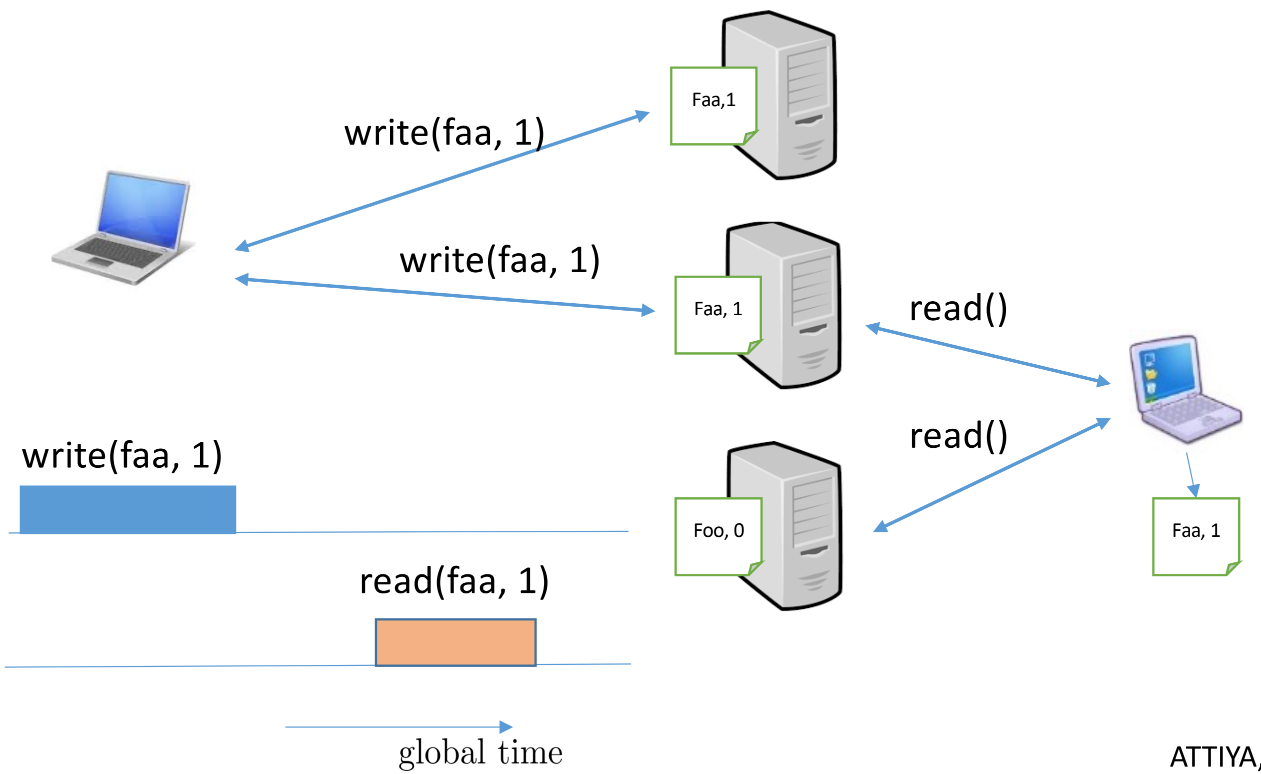
# Atomicity/Linearizability

- 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, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.

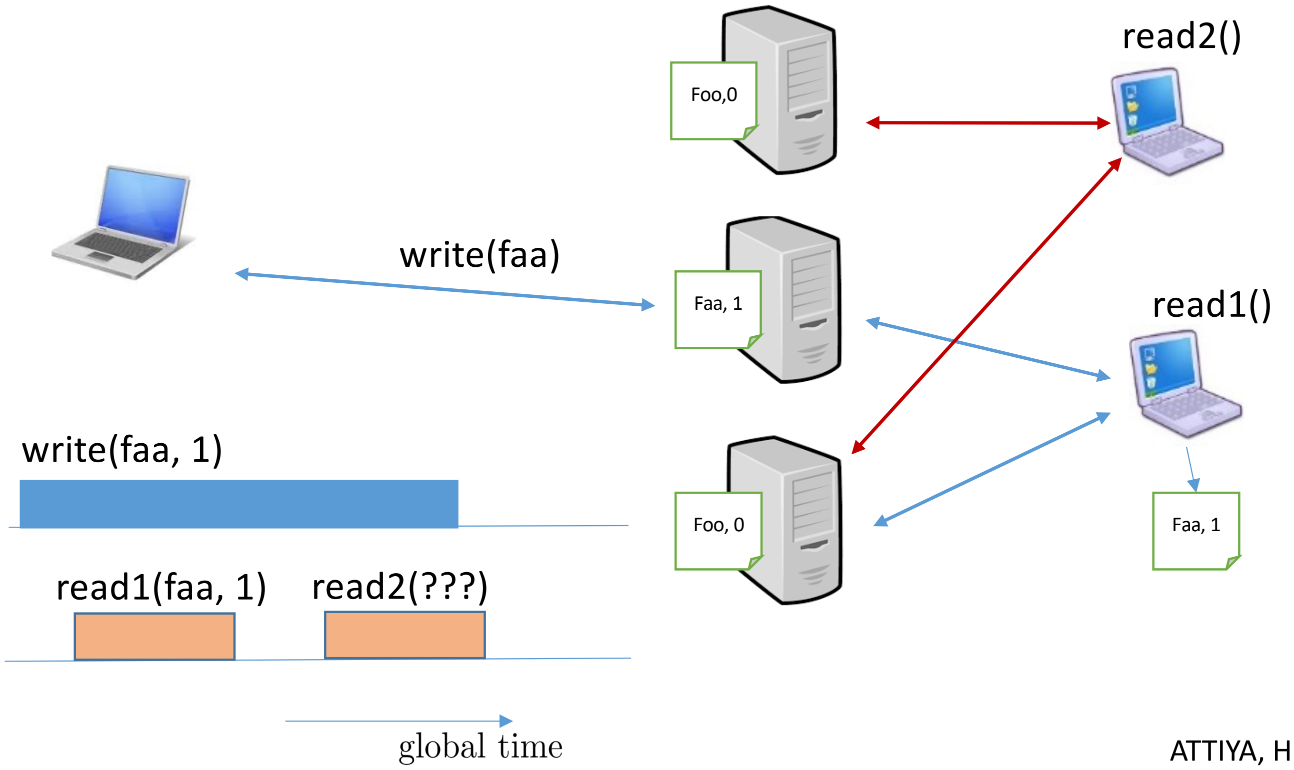
# A Simple Solution - ABD




- Attiya, Bar-Noy, Dolev: an elegant, intuitive solution
  - Use the power of the majority
  - Assign logical timestamps to written values
  - Wait-free solution

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

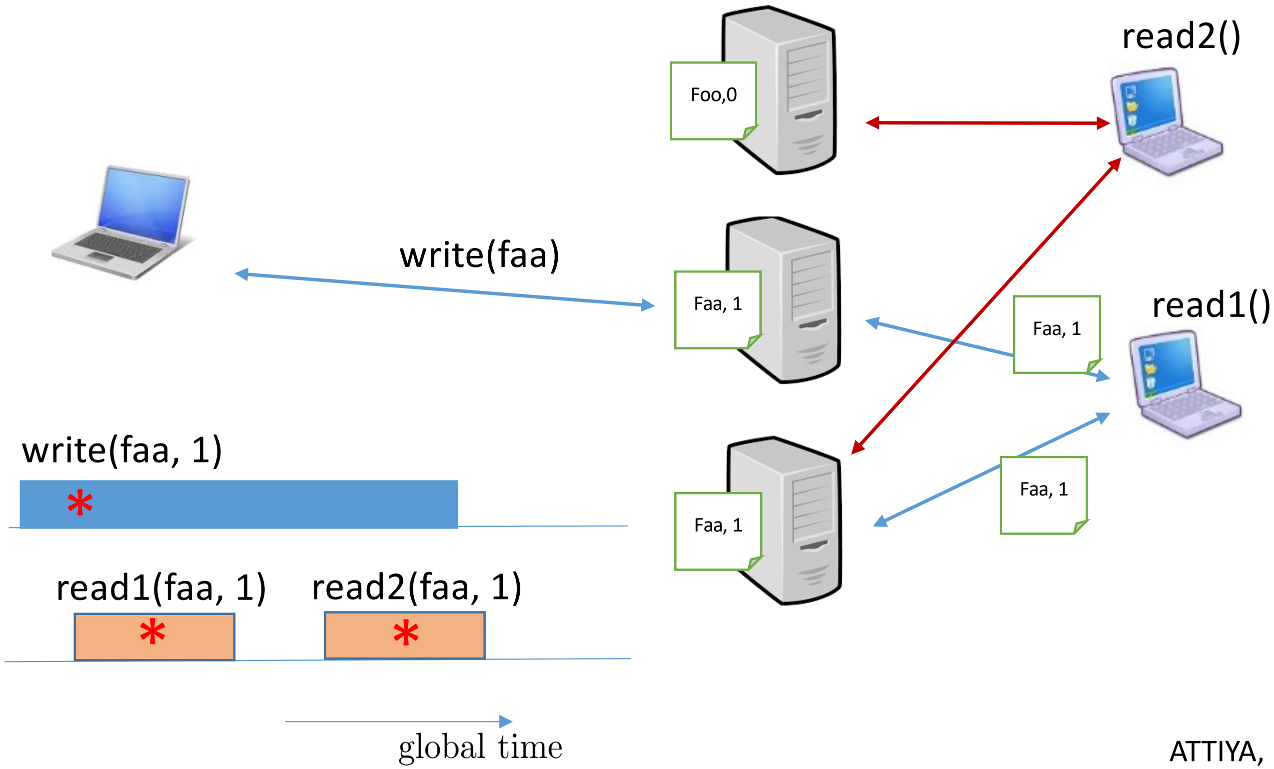
# A Simple Solution - ABD




- A "read" needs to "write"
  - Phase 1: query
  - Phase 2: propagate

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

# A Simple Solution - ABD

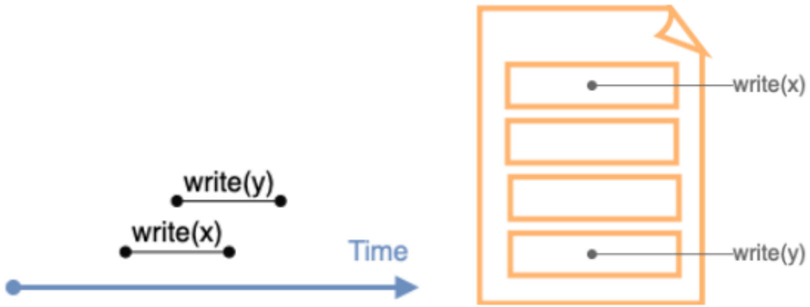



- A "read" needs to "write"
  - Phase 1: query
  - Phase 2: propagate

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

# Solutions for Large Objects

- ABD efficient for small objects
  - Each write operation sends (S/2)+1 copies of the object
  - Each read operation sends S copies of the object
- Moreover concurrent write operations may overwrite one another
  - Unbale to handle write operations working on different parts of a large object

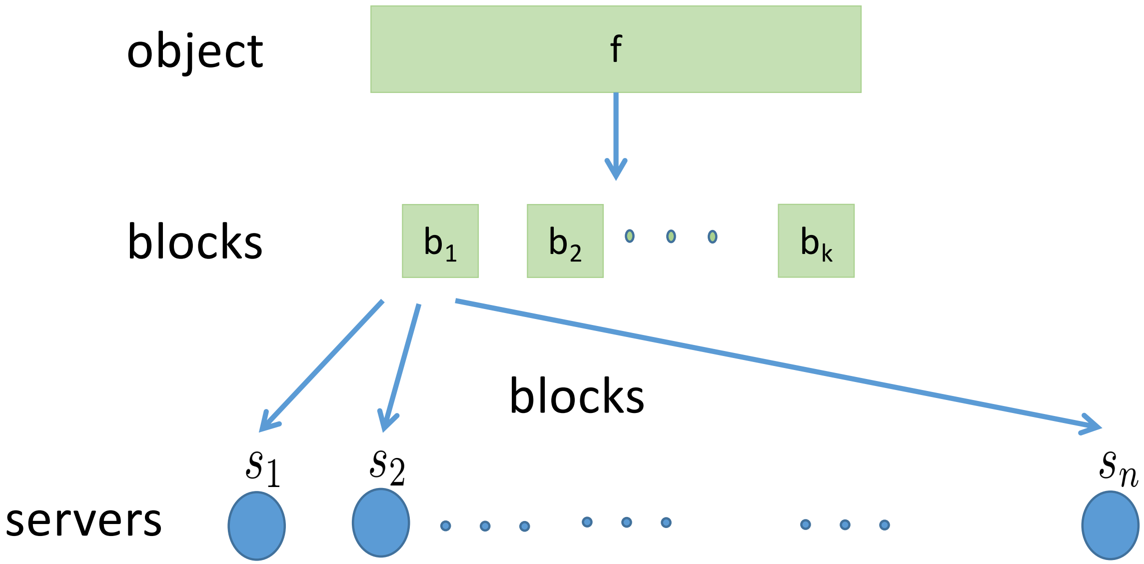


# Main Project Goal

Develop practical and robust DSS in the message-passing, asynchronous, environment while allowing high concurrency and preserving strong consistency.

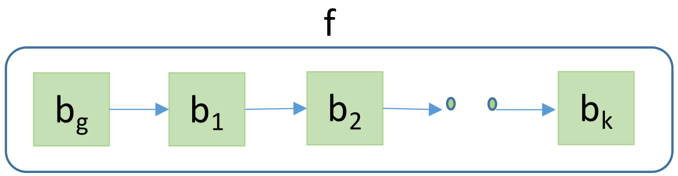
# Solution 1: Fragmentation

- Most intuitive solution
  - Split large objects into smaller fragments
  - Treat each individual block as an atomic object



# Solution 1: Algorithm CoBFS

- Fragmented Objects:
  - Connected list of blocks
  - Each block points to the next block



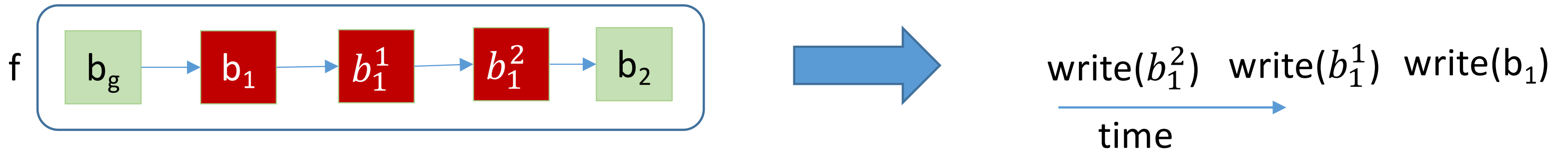

- Write Operation write(f)
  - Propagate only modified and new blocks

- Read Operation read(f)
  - Start from genesis block and read all the blocks
  - Optimization: Only blocks that have changed are send to the read

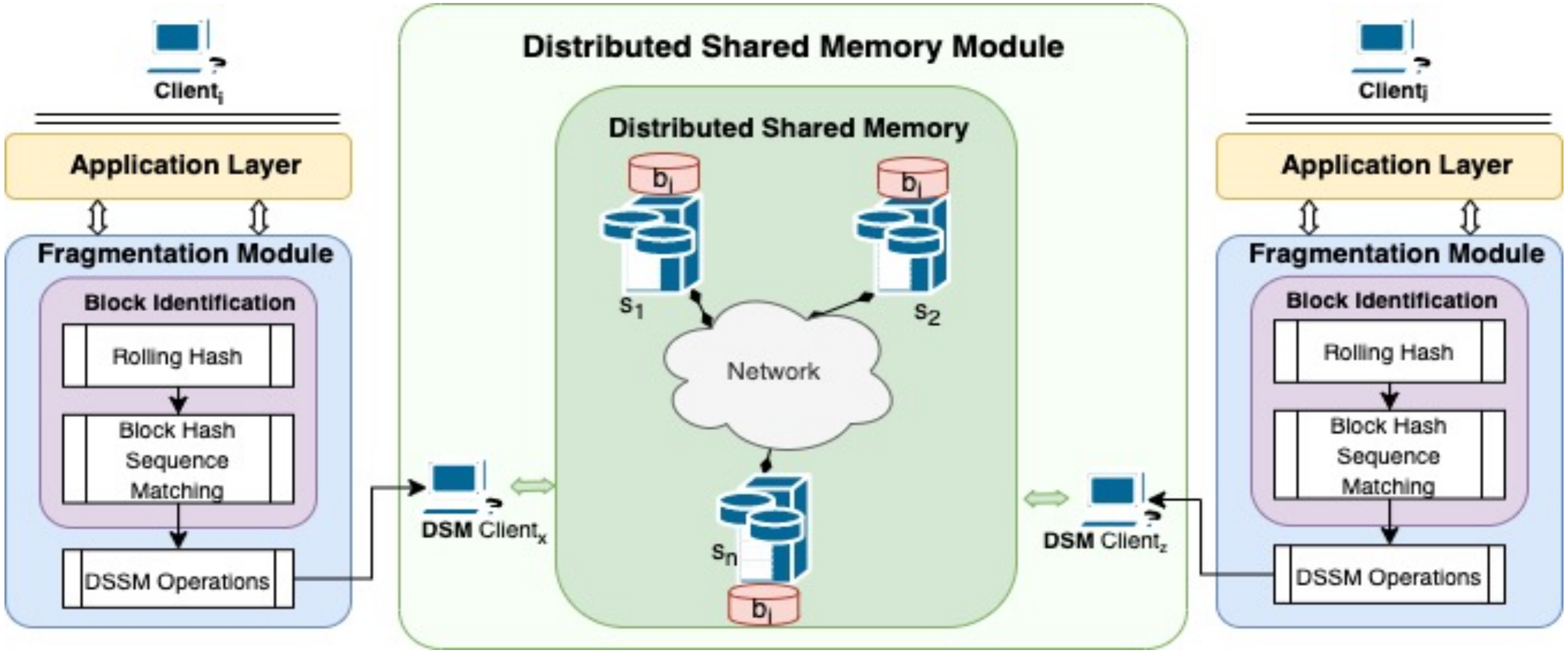
# Solution 1: Algorithm CoBFS

- Write/Update Operation:
  - Run fragmentation and block matching algorithms to determine
    - Modified blocks
    - New blocks
  - Case 1: Only a single block has changed


  - Case 2: Changed block overflowed and new blocks introduced

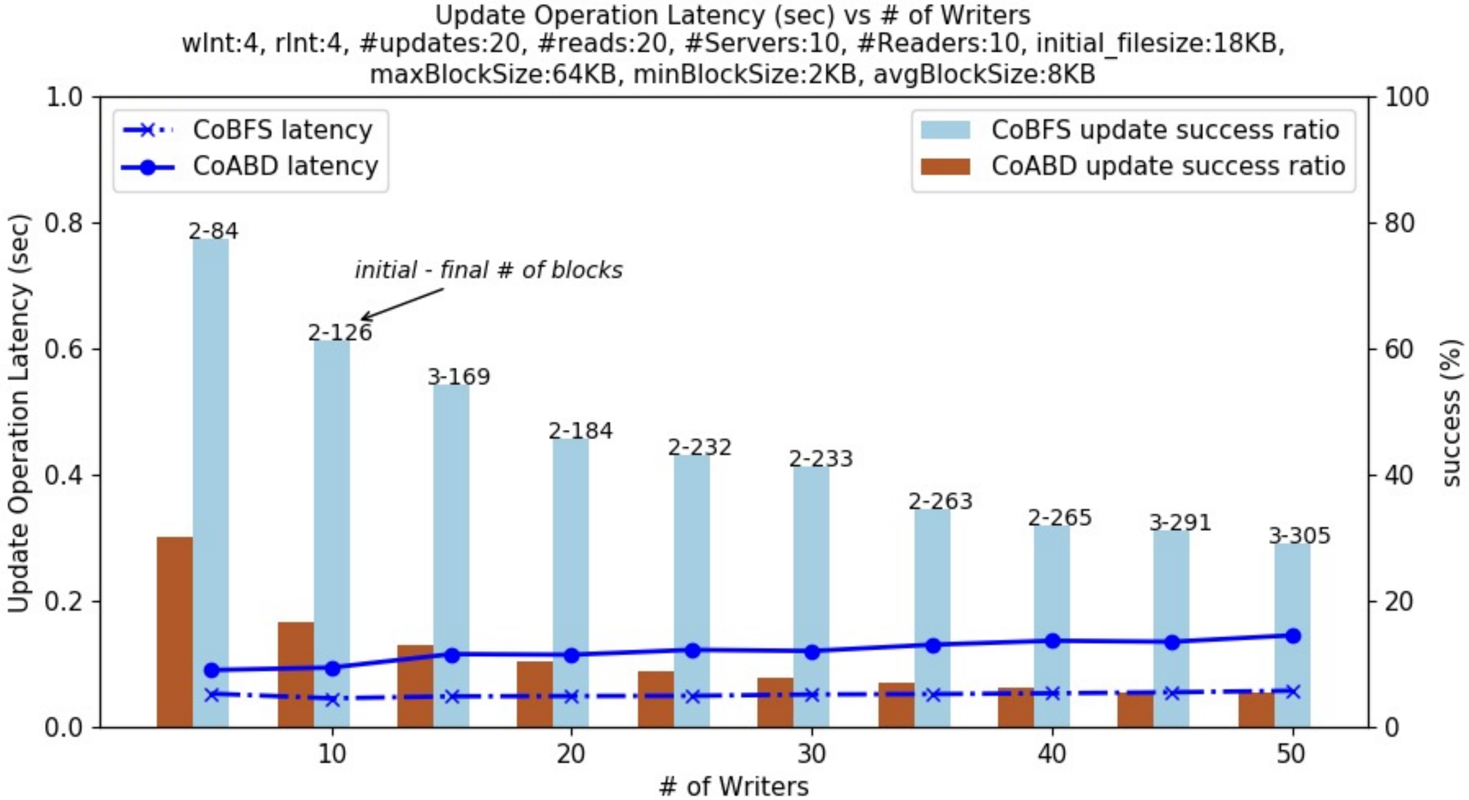


# Solution 1: Basic Architecture

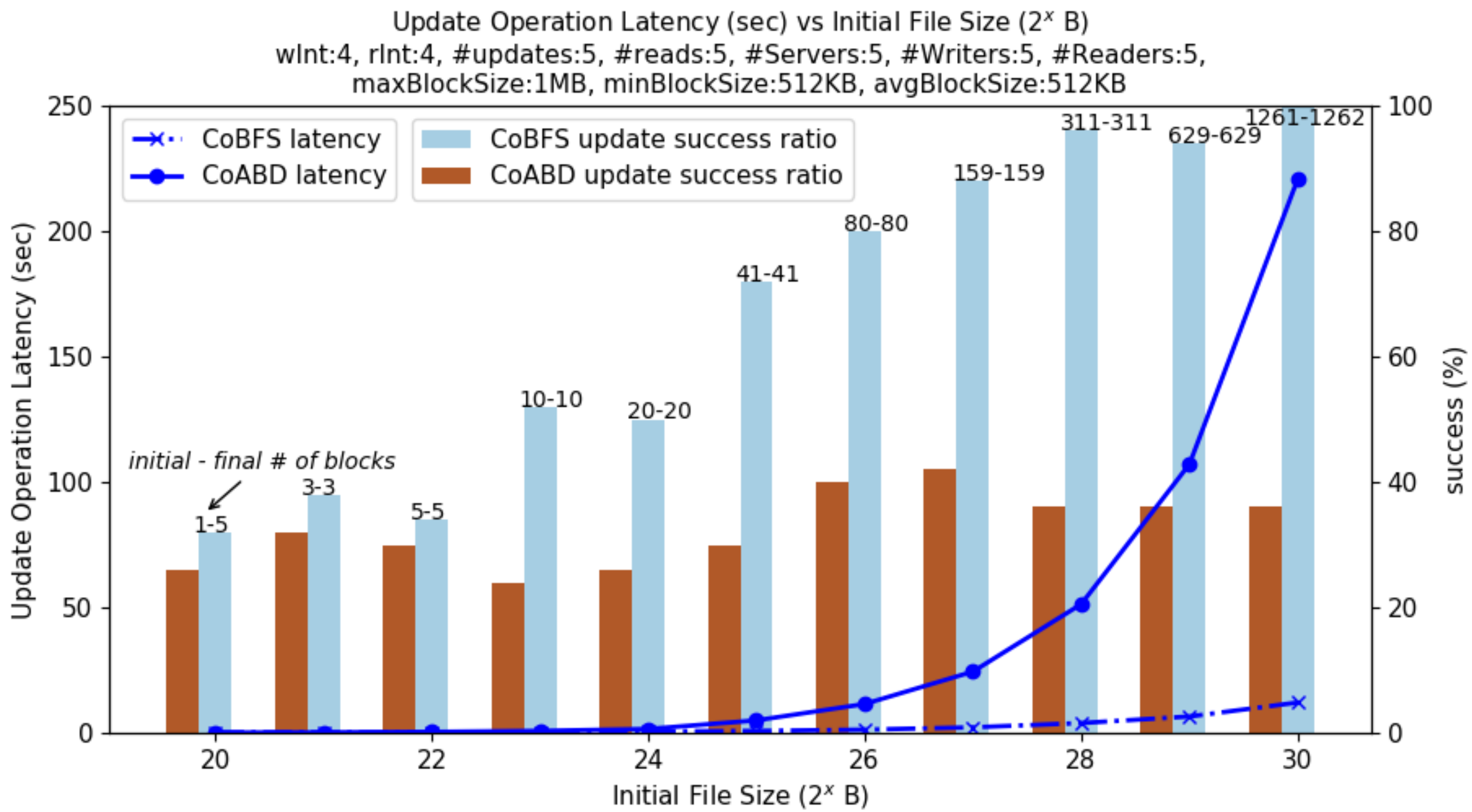



Rabin, M O. Fingerprinting by random polynomials, *Center for Research in Computing Techn., Aiken Computation Laboratory, Univ. , pp 15—18, 1981*

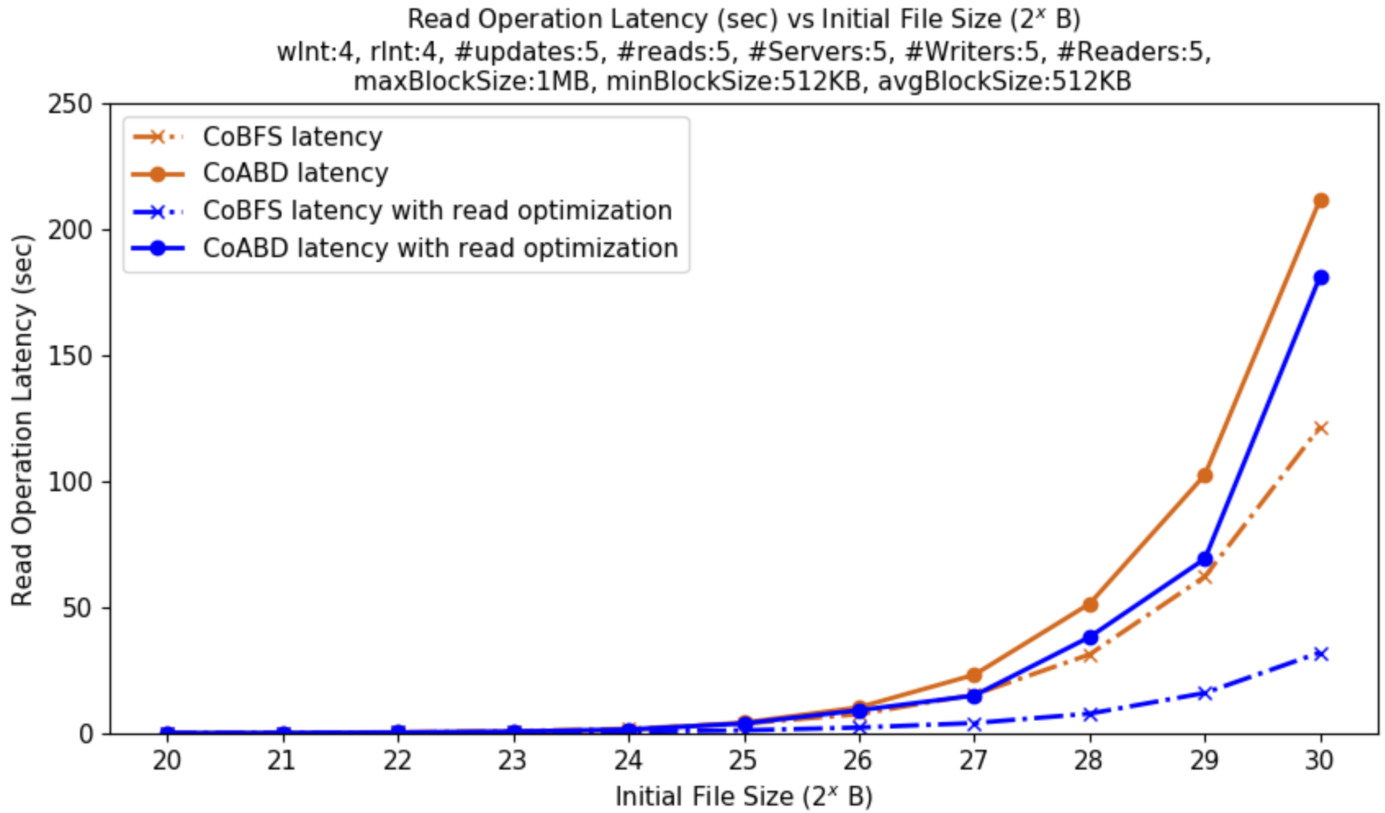
# Solution 1: Experimental Results - Scalability



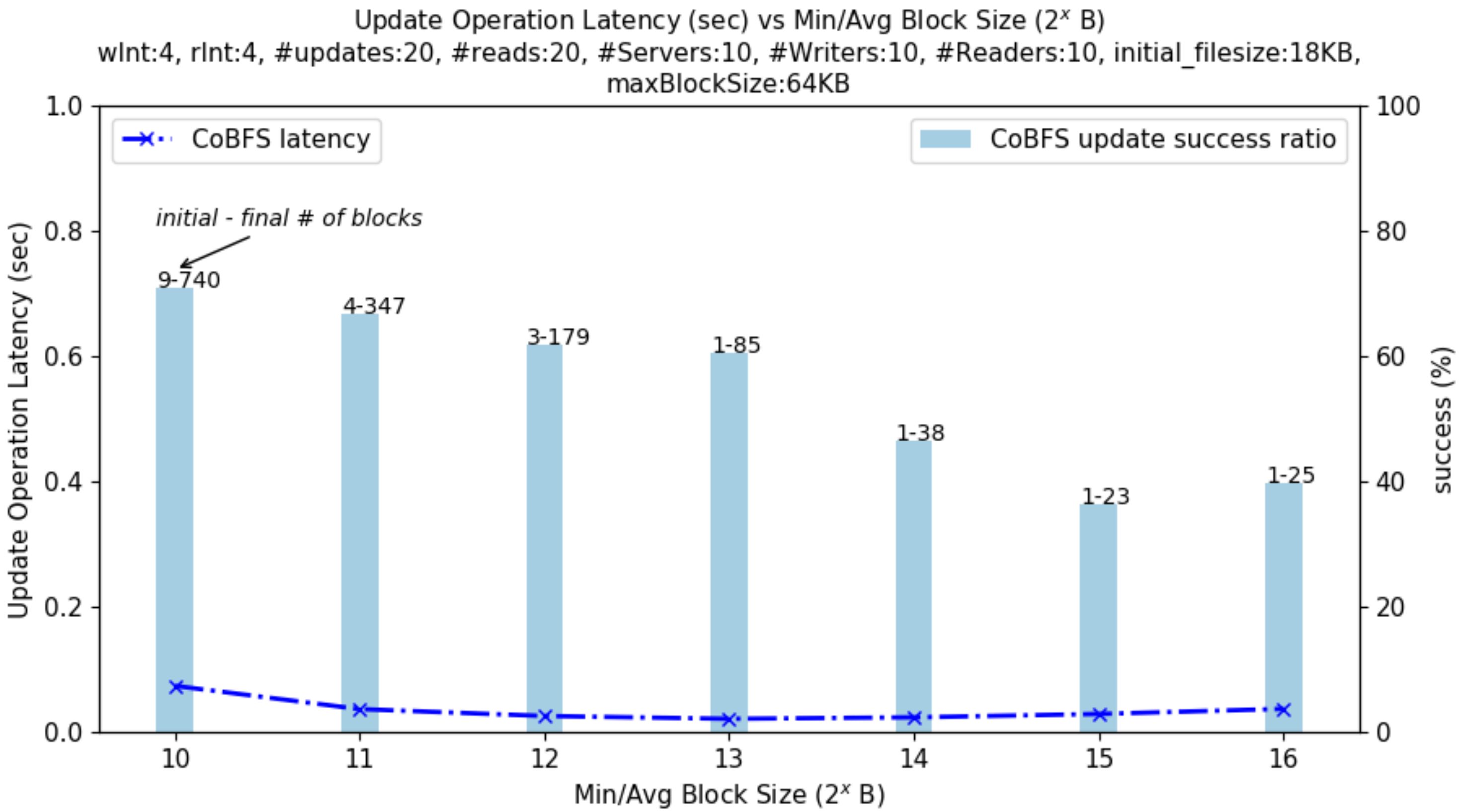
# Solution 1: Experimental Results - Filesize



# Solution 1: Experimental Results – Read Optimization

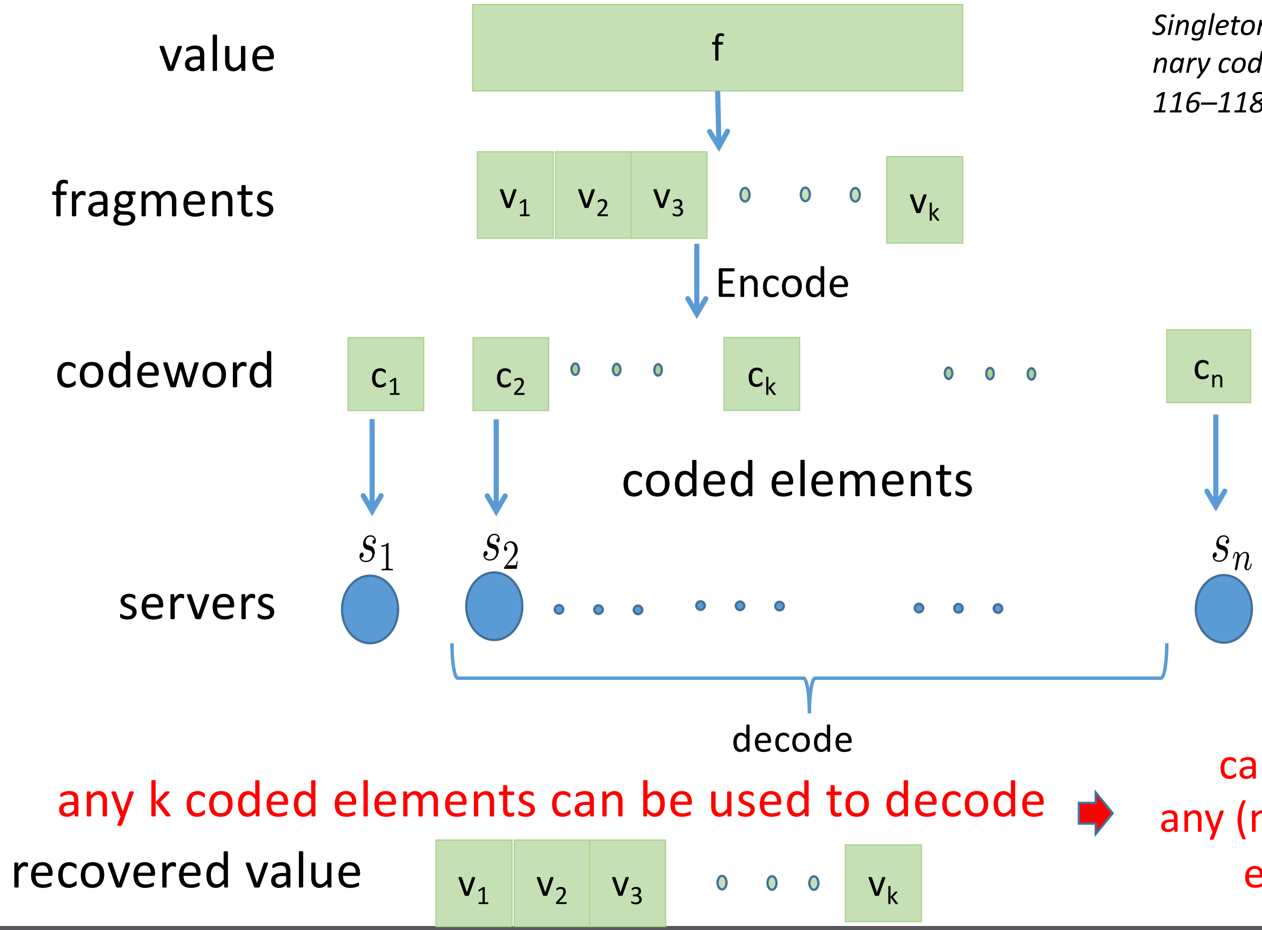


# Solution 1: Experimental Results – Block Size



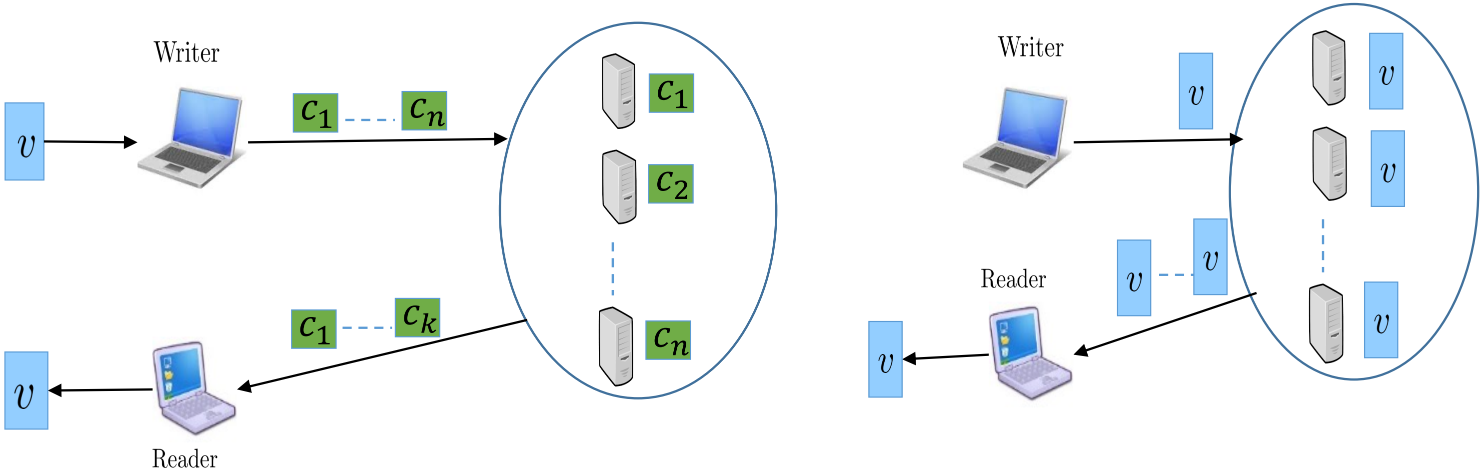
# Solution 2: Erasure Coding ([n, k] MDS Codes)

*Singleton, R.C. (1964), "Maximum distance q-nary codes", IEEE Trans. Inf. Theory, **10** (2): 116–118, doi:10.1109/TIT.1964.1053661*




can tolerate any (n – k) missing elements

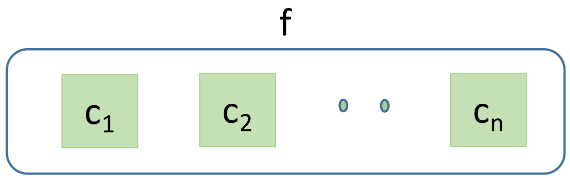
# Solution 2: Erasure Coding vs Replication



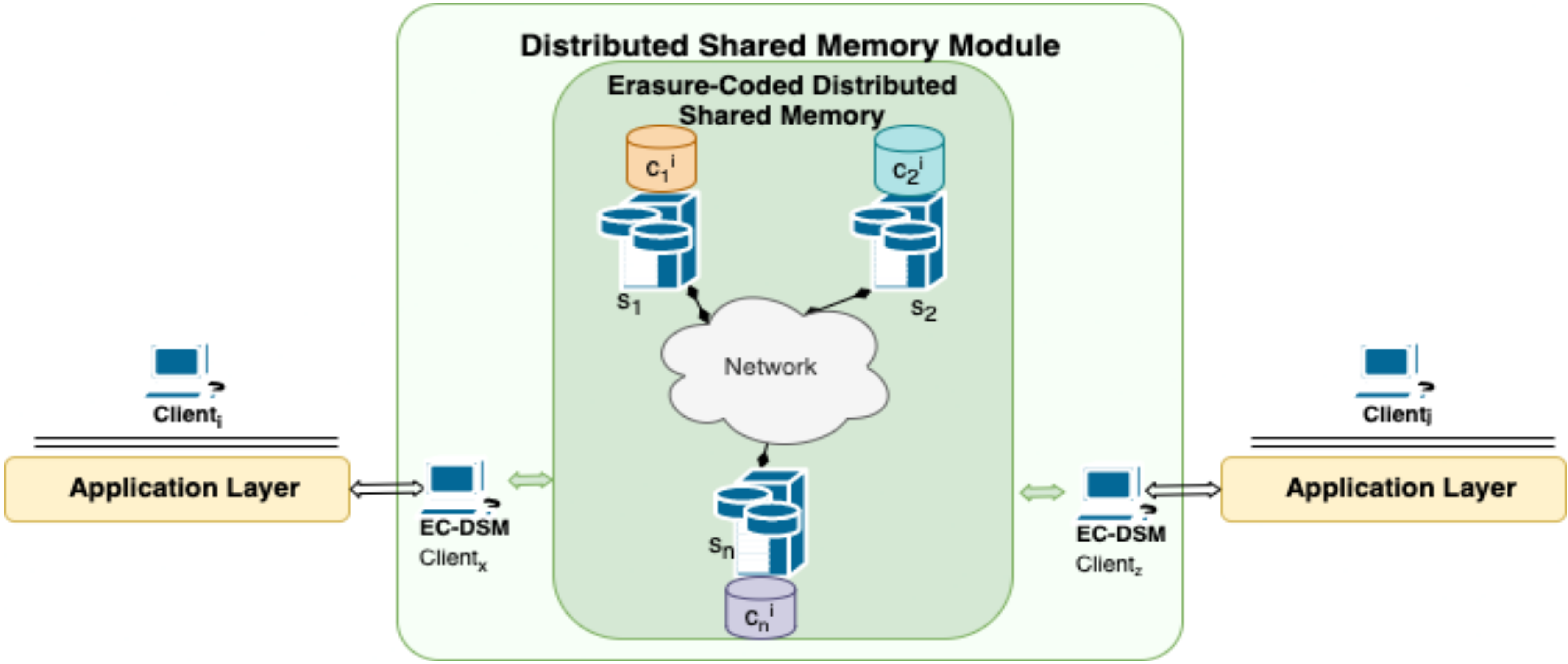

A well-designed algorithm has great potential to reduce storage and communication costs while using erasure codes

# Solution 2: Algorithm CoEC

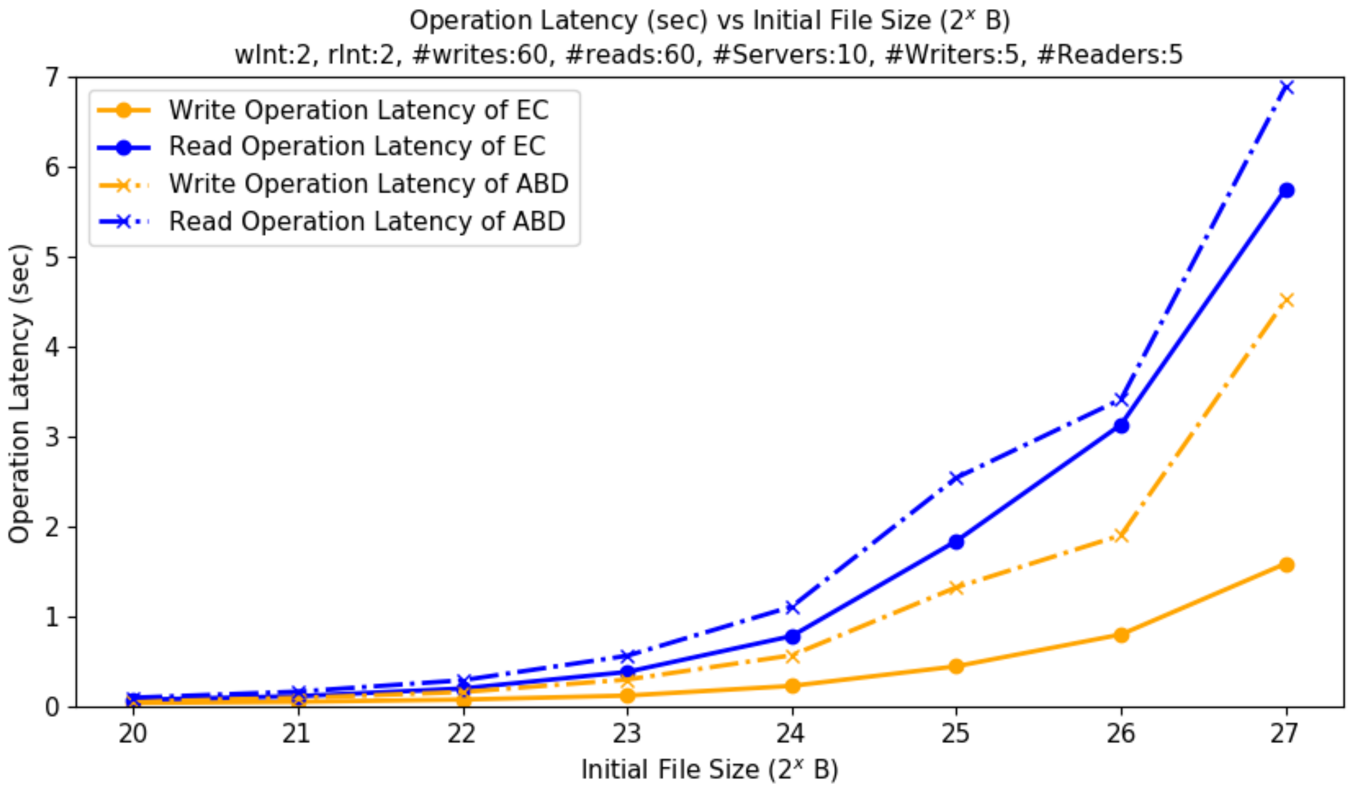
- Write Operation write(f)
  - Apply erassure coding on f
  - Send code $c_i$ to server $s_i$
- Read Operation read(f)
  - Collect *k* codes from the servers
  - Decode and return the value of *f*
- Ensures Strong Consistency
- Does not prevent overwriting



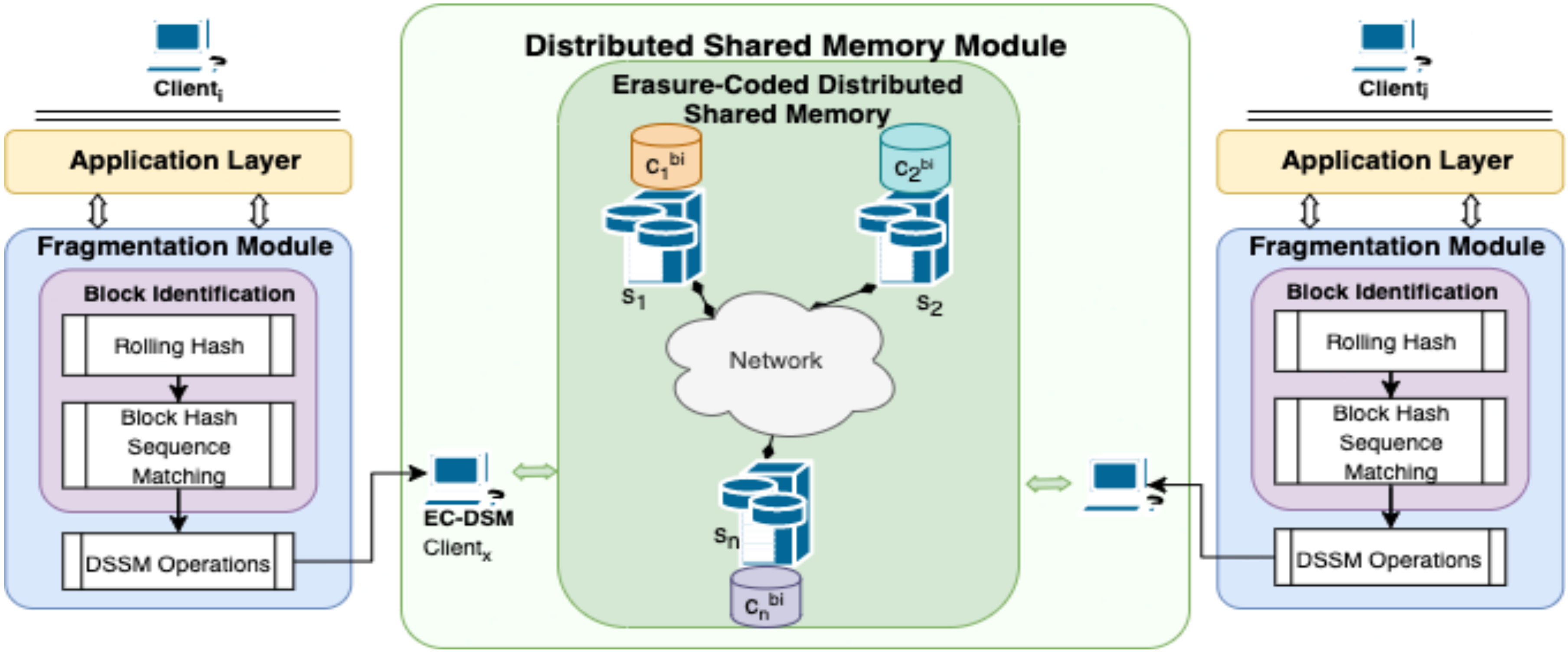
# Solution 2: Erasure Coding Architecture



# Solution 2: Experimental Results



# Solution 3: Hybrid



# What happens when things go wrong?

- Tolerate minority of failures

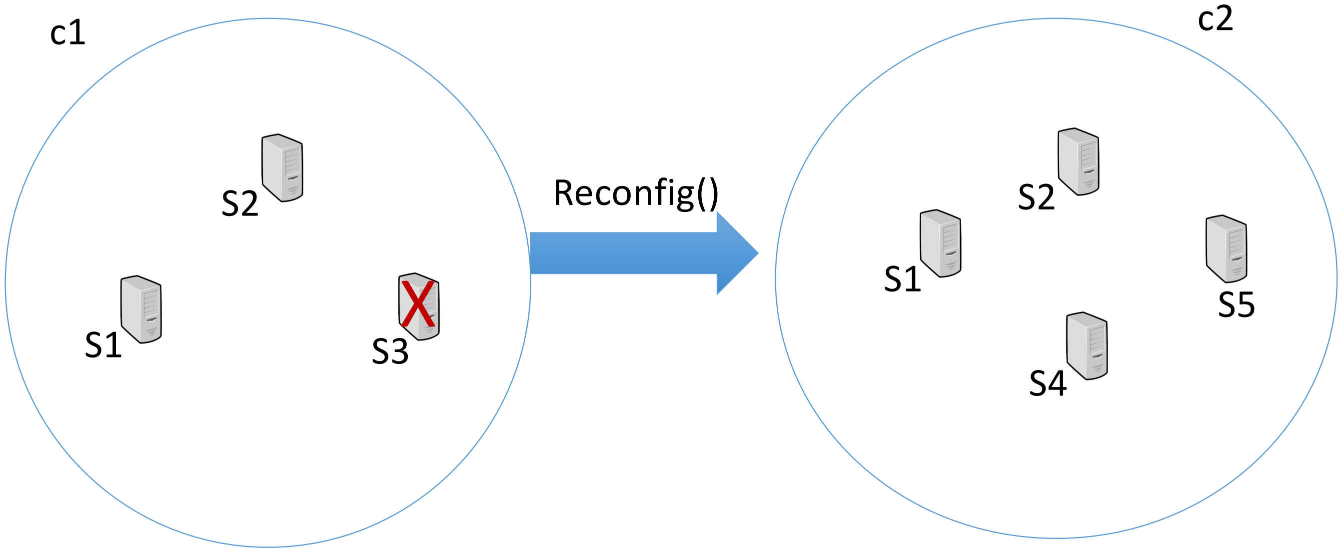
- What if more than minority fail?

- Replace failed with healthy servers => Reconfiguration

**Challenge:** Can we install a new configuration without stopping the service and without violating linearizability?

# Re-Configuration Operation

- Change the configuration parameters (add/replace servers)
  - Due to failures
  - Due to admin maintenance




LYNCH, N., AND SHVARTSMAN, A. RAMBO: A reconfigurable atomic memory service for dynamic networks. *In Proceedings of 16th International Symposium on Distributed Computing (DISC)* (2002), pp. 173–190.

# ARES: A modular and adaptive reconfiguration protocol

**Modular**

- Read/Write operations are not aware of the underlying shared memory implementation
  - They are using the same access primitives

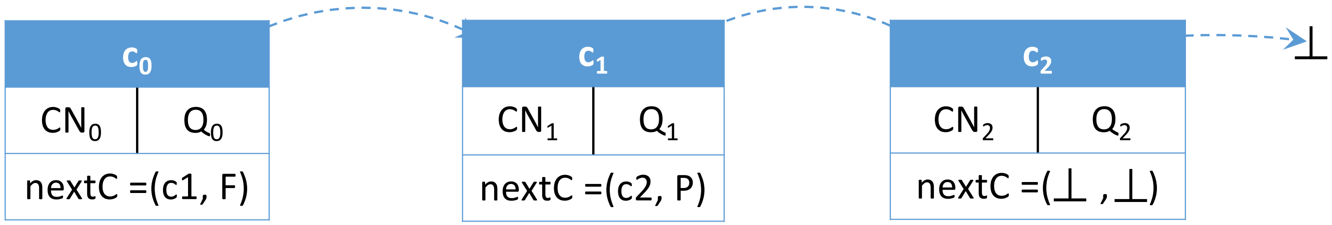
**Adaptive**

- Different shared memory algortihm may be used in every configuration
  - Satisfying application demands

NICOLAOU, N., CADAMBE, V., KONWAR, K., PRAKASH, N., LYNCH, N., AND MEDARD, M. ARES: Adaptive, Reconfigurable,Erasure Coded, Atomic Storage. *In Proc. of ICDCS, pp. 2195–2205 (2018*)

# Configuration Sequence

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



# Reconfiguration Service

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

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

(1):
- attempt get to the latest configuration

(2):
- introduce the new configuration
- migrate the data to the new config
- let servers know it is good to be finalized

# Reconfiguration Service Guarantees

For any two reconfig ops $\pi_1$, $\pi_2$ s.t. $\pi_1$ before $\pi_2$

- Configuration Consistency

| | | | | |
|---|---|---|---|---|
| $\pi_1$ | $c_0$ | $c_1$ | $c_2$ | … |
| $\pi_2$ | $c_0$ | $c_1$ | $c_2$ | … |

- Sequence Prefix

| | | | |
|---|---|---|---|
| $\pi_1$ | $c_0$ | $c_1$ | |
| $\pi_2$ | $c_0$ | $c_1$ | $c_2$ |

- Sequence Progress

| | | | | |
|---|---|---|---|---|
| $\pi_1$ | <$c_0$, F> | <$c_1$, P> | <$c_2$, P> | |
| $\pi_2$ | <$c_0$, F> | <$c_1$, P> | <$c_{2,}$ , F> | … |

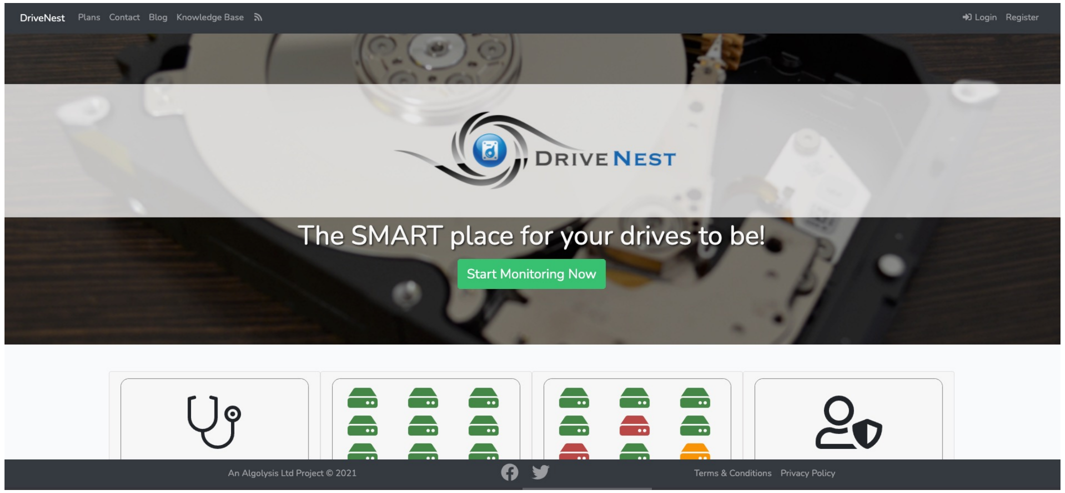
# ARES: Experimental Results

# When shall we reconfigure?

- Frequent reconfigurations => Slow Down the service
- Infrequent reconfigurations => May make the service anavailable

**Challenge:** How can we determine when is the best time to reconfigure?

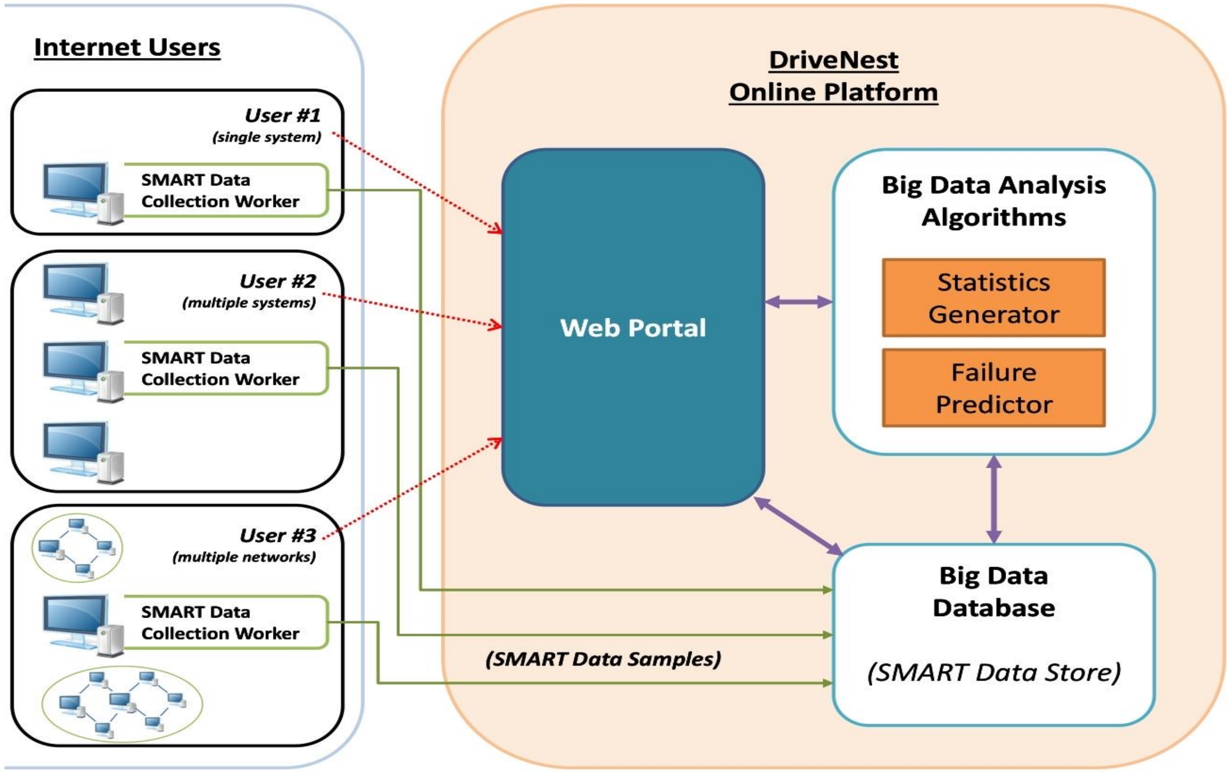
# DriveNest: Monitoring Node Health




www.drivenest.com

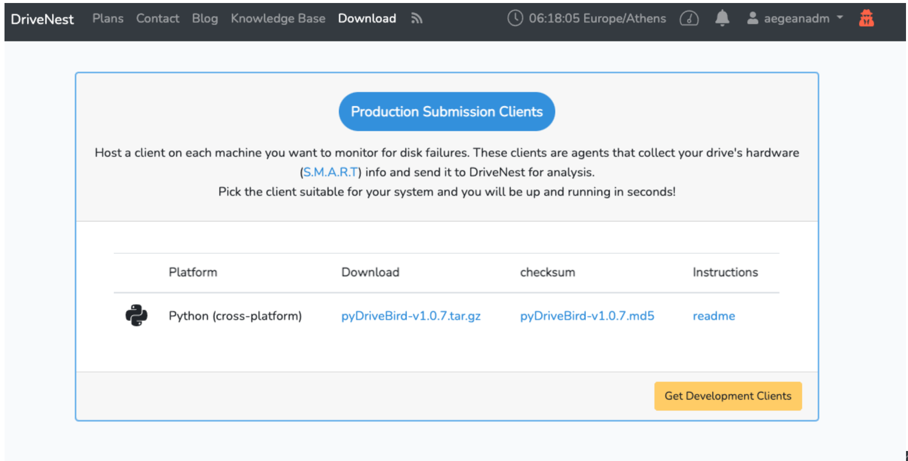
- Crowdsourcing Platform
  - Collects data from diverse setups, locations, conditions.
- Monitor storage device health by collecting S.M.A.R.T data
- Predict soon-to-fail drives
  - Prediction performance relies on the report of failed drives
- Integration with ARES
  - Initiate recon operation to remove drives that are predicted to fail
  - Replace them with healthy nodes and migrate data

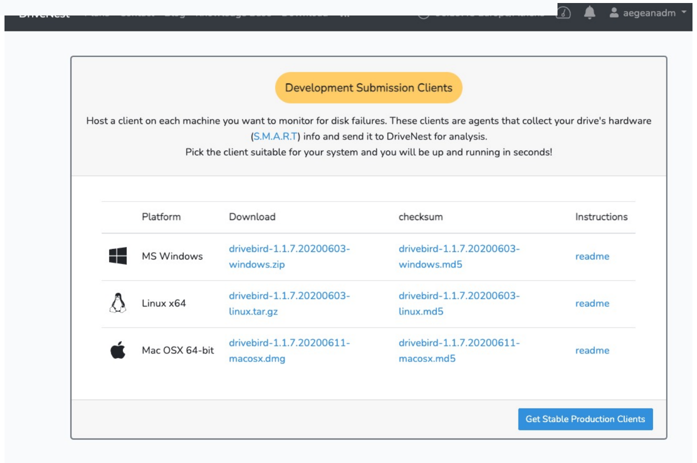
# Drivenest: Architecture




- **DriveBird**: Data Collection Clients
- **Web platform**: View your drives
- **Prediction Engine**: Applies a number of machine learning/ deep learning algorithms to predict soon-to-fail drives

# DriveNest: DriveBird Client




- Production Submission Clients
  - Python (cross platform)
- Development Submission Clients
  - GUI interface for all platforms
  - Tested up to Win 10 and MacOS Sierra

# Short Demo

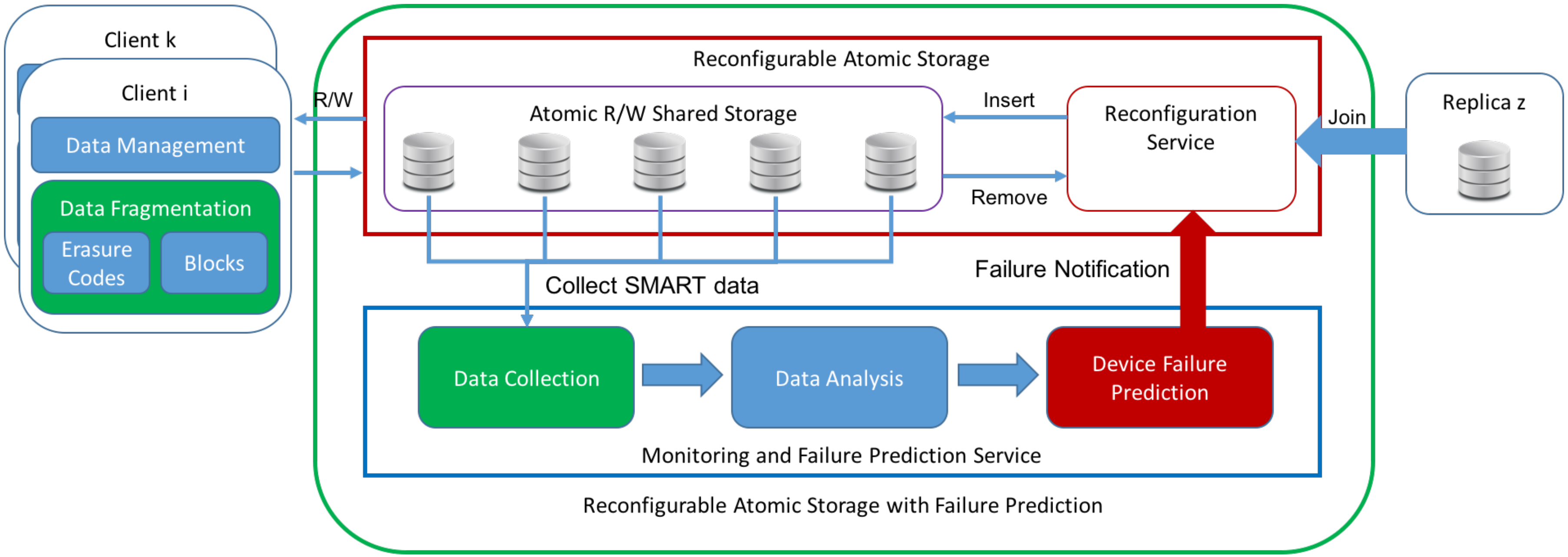
# DriveNest: Status

- Drivenest Collection Clients: **Alpha Testing**
- Drivenest Web Platform: **Alpha Testing**
- Drivenest Predictions: **Development Stage**

Feel free to register and give the service a test **“drive”**. ☺

We would be glad to hear your feedback

# COLLABORATE: Overall Architecture



# COLLABORATE: What's ahead

- Complete the prediction algorithms to be used in DriveNest
- Complete the integration of DriveNest with the Shared Storage algorithms
- Deploy the algorithms on real systems (AWS, Rpis)
  - Collect and Analyse the data
- Embedd the developed algorithms in a production level Distributed Storage Service!

# Thank you!

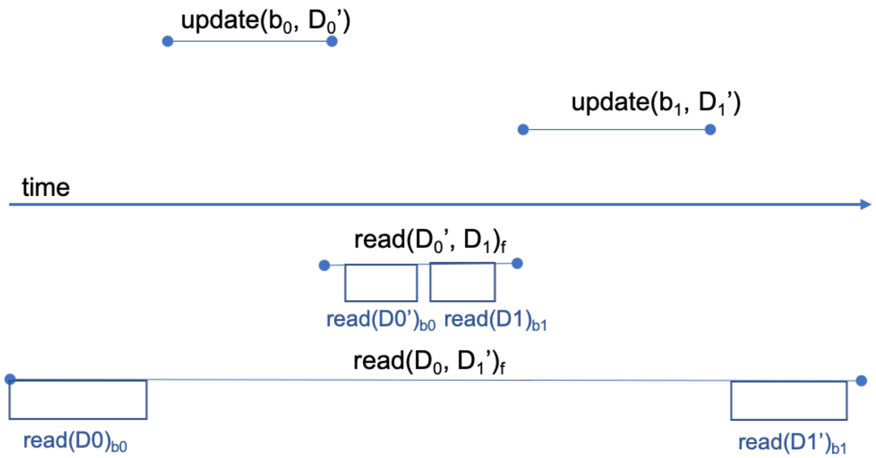
algolysis

algorithmic solutions

www.algolysis.com

research@algolysis.com

# Solution 1: Fragmented Linearizability




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