D-rex: Dynamic Data Replication for Heterogeneous Storage Nodes

June 23, 2024

Maxime Gonthier, Dante D. Sanchez-Gallegos
Jesus Carretero, Kyle Chard, Ian Foster

(Disclaimer: ongoing work - topic defined by a new postdoc (me :o))

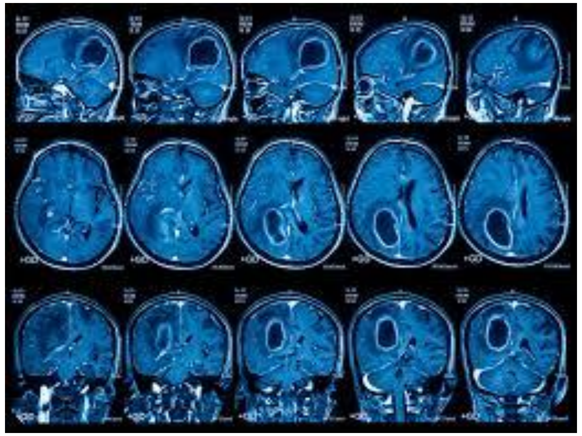
Table of contents:

Context: Data needs long-term storage

Data: Users generate large amounts of data that must be stored for long periods of time.

Resilience: Given the importance of some data, storage systems implement replication policies to tolerate failures.

Cloud: A cloud solution enables users to access a diverse range of heterogeneous storage nodes distributed across multiple locations.

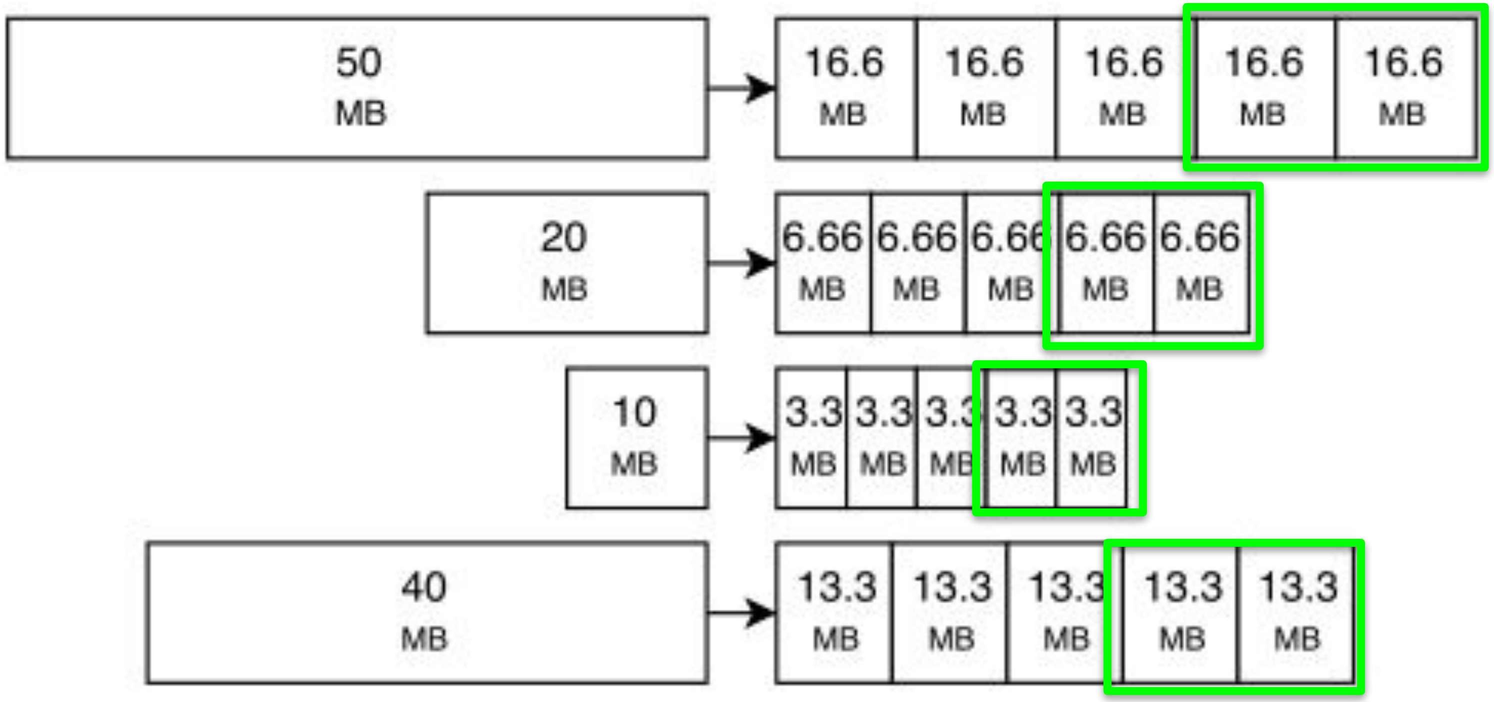


Context: Erasure coding reduces storage overhead

Erasure coding: Encodes data into chunks so that when the data is needed, only a subset of the chunks is needed to recover the original data

- N: Number of chunks
- Parity chunks: Added chunks to ensure data reconstruction from a subset
- K: N - Parity chunks.

Example with N=5 and K=3: Can survive N - K (2) nodes failures



Each chunk is stored in a different location.

Motivations

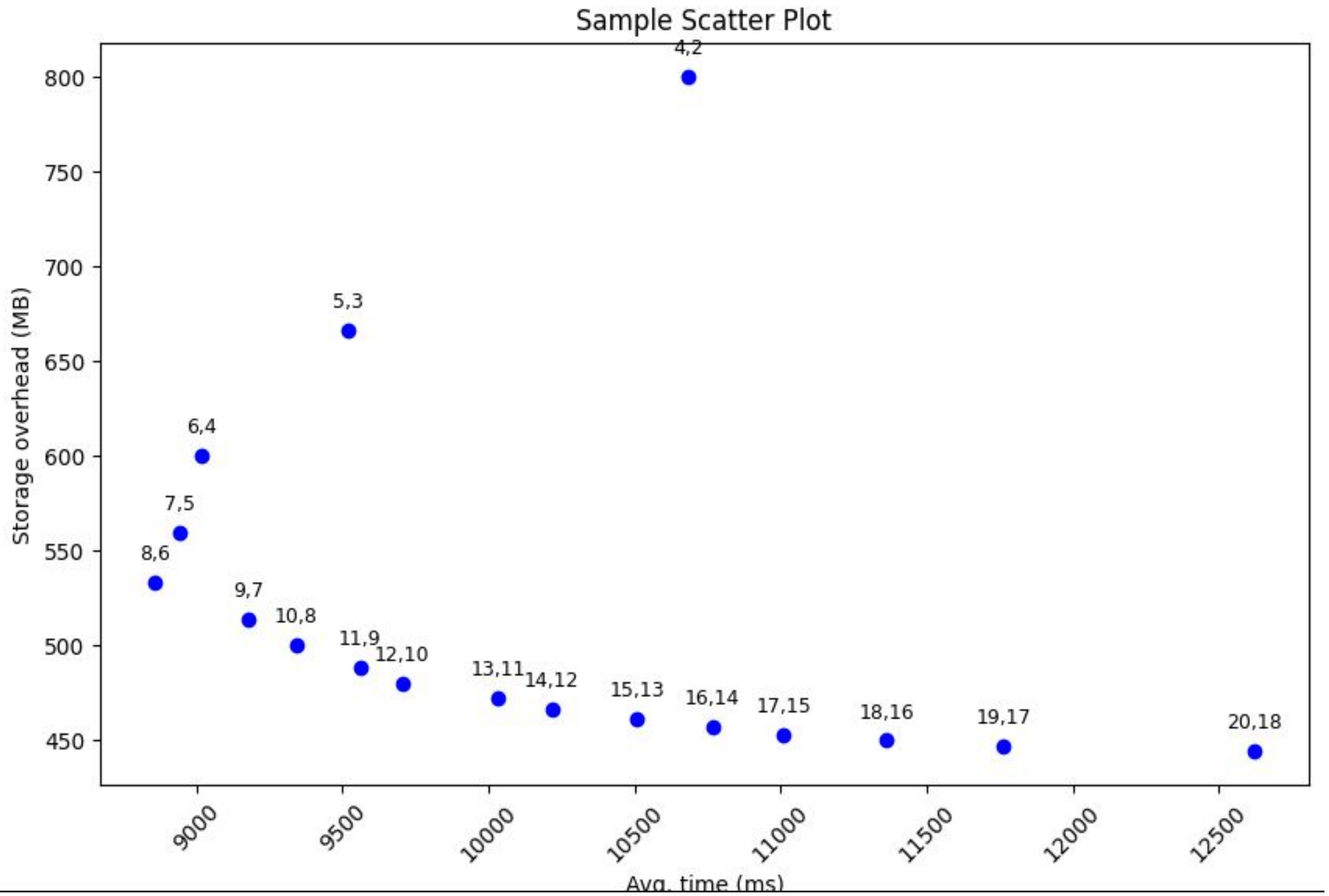
Storage is not free

Heterogeneity

Chunking cost time

Current solutions are static, even though the workload can be dynamic

Time to chunk 400 MB of data. Varying N and K



Model

Nodes:

$\mathbb{S} = \{S_1, \ldots, S_m\}$

Size $M(S_i)$

Probability $0 < P_{Failure}(S_i, t) \leq 1$ to fail at least once

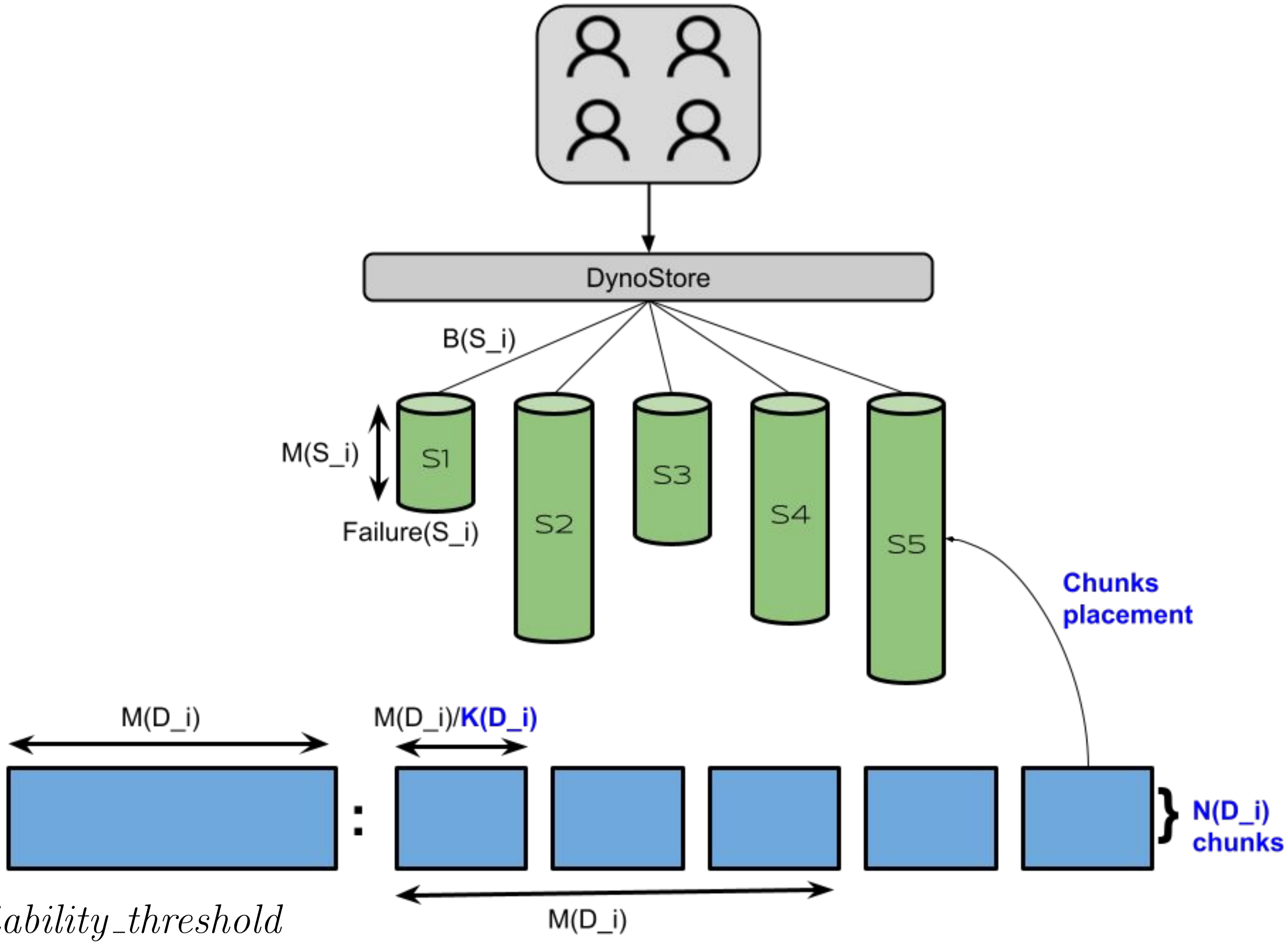
Write bandwidth $B(S_i)$

Data:

Need to store $\mathbb{D} = \{D_1, \ldots, D_{m'}\}$

m' is unknown

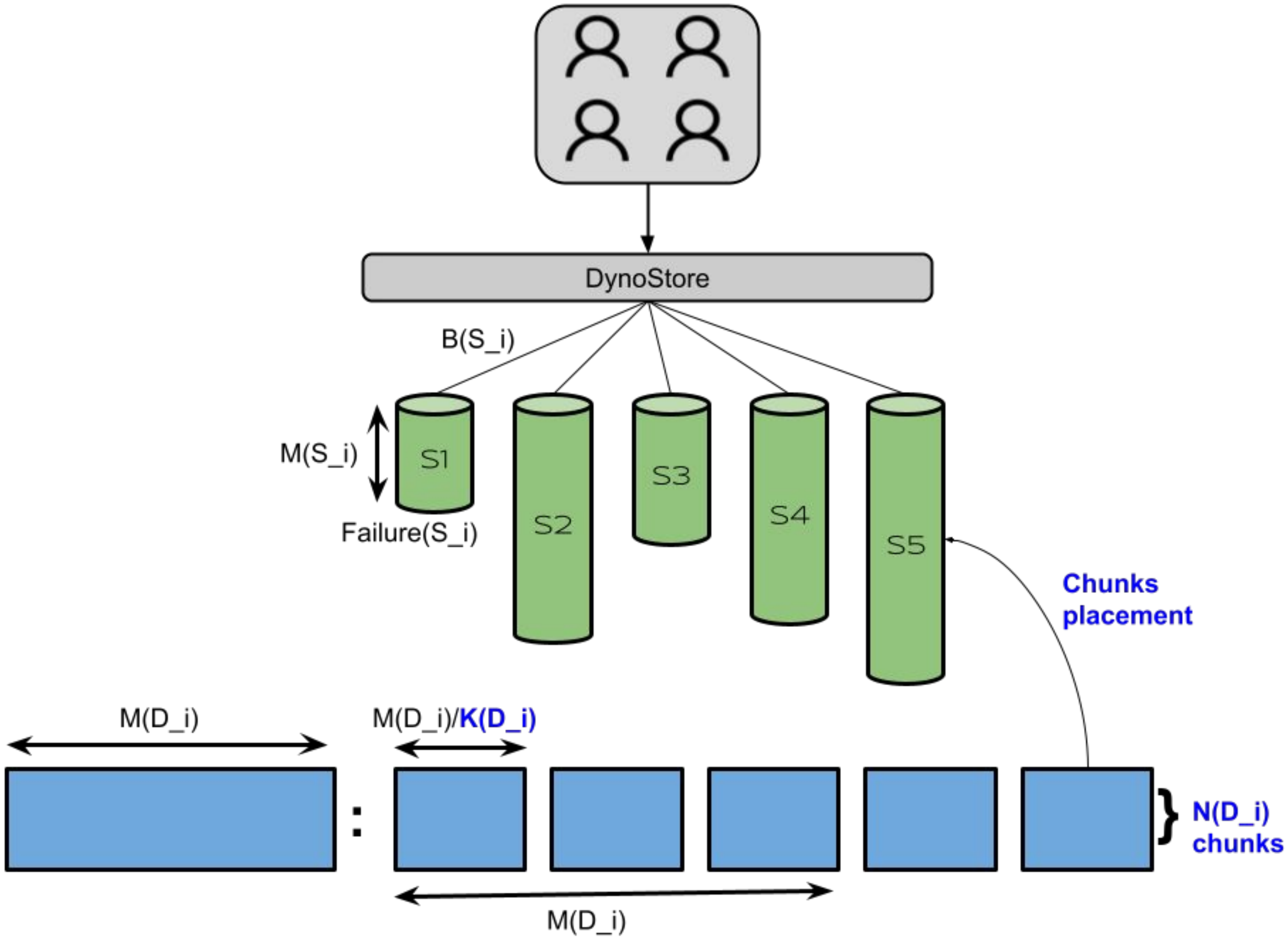
Size M(D_i)



Constraint: for each data $P_{Available}(D_i) \geq Reliability\_threshold$ computed using $P_{Failure}(\mathcal{S}(D_i), t)$ and a Poisson binomial distribution function

Problem Statement

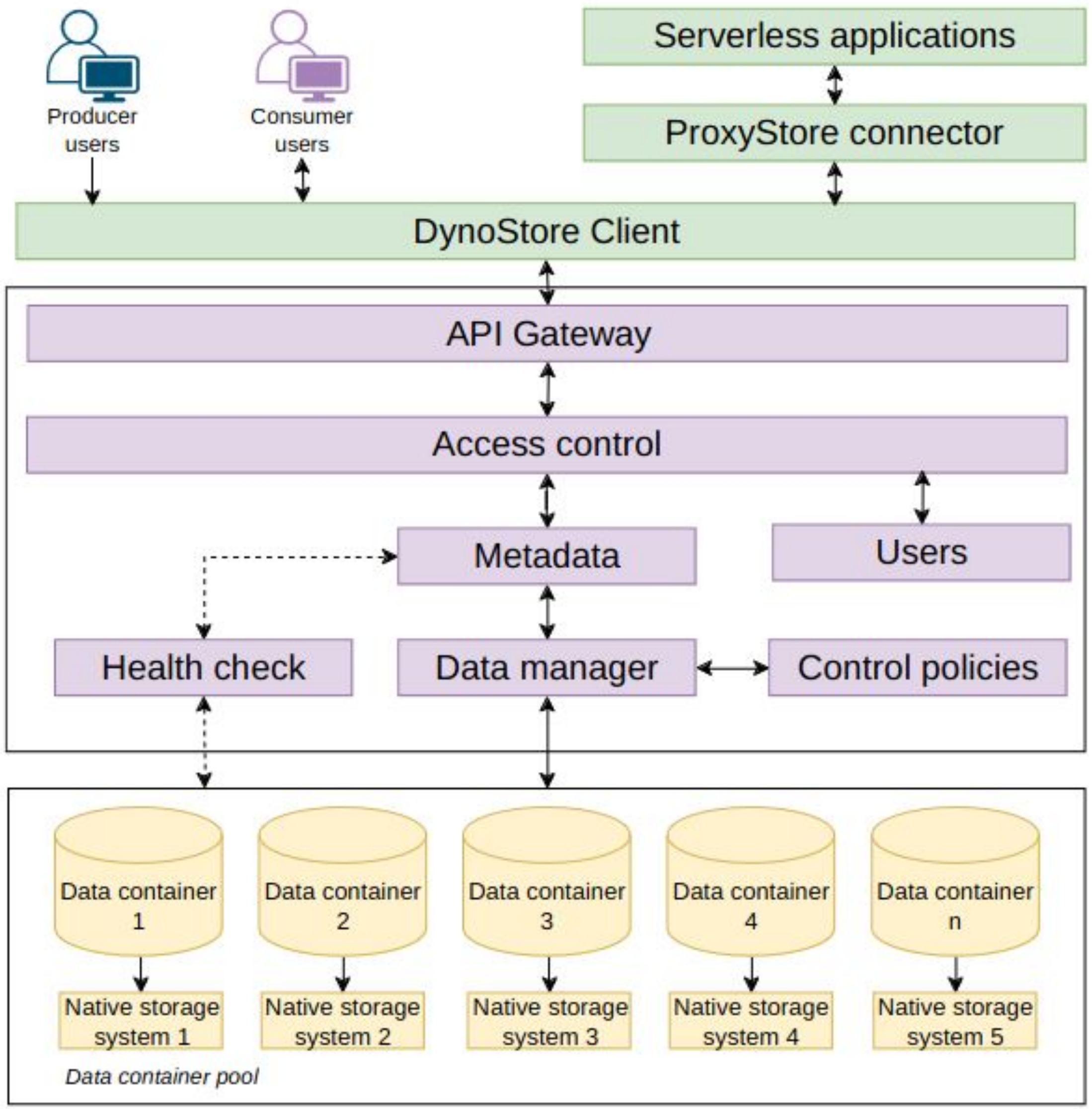
Given a set of **heterogeneous nodes**, for each of the m' (unknown) data to be stored, what values of **N** and **K** and which **subset of nodes** should be chosen to **reduce storage overhead**, **chunking**, and **upload time**, and to **store as much data as possible** while maintaining **reliability** greater than 99%?



DynoStore

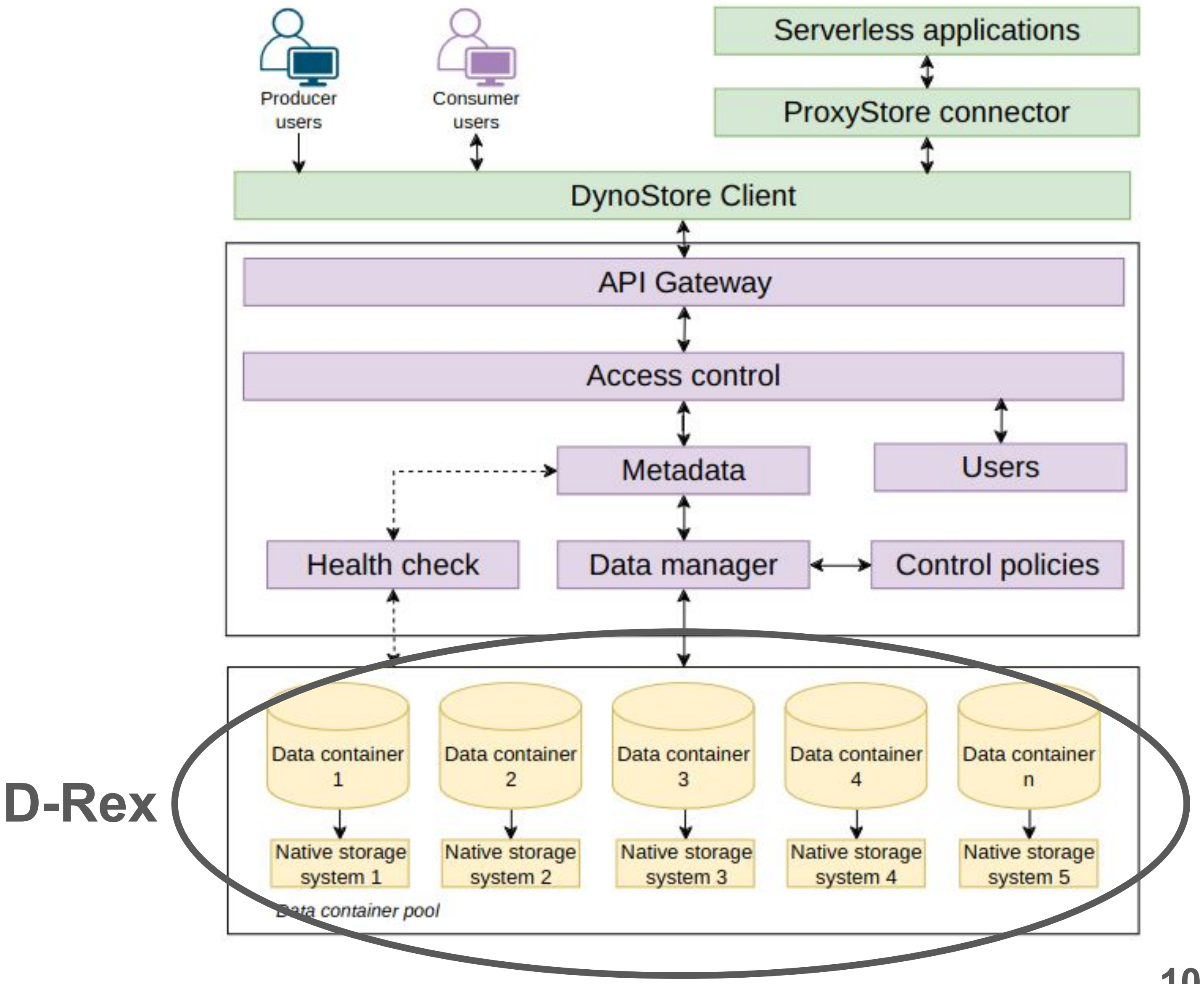
Design of DynoStore

A wide-area distribution system designed to manage data across heterogeneous storage systems



Design of DynoStore

A wide-area distribution system designed to manage data across heterogeneous storage systems



Algorithms

From the state-of-the-art

HDFS 2.0: 3x replication ⟶ 200% storage overhead

HDFS 3.0 default configuration: Erasure coding using Reed-Solomon (6,3): A data is split into 6 blocks and 3 blocks of parity are added ⟶ 50% storage overhead

HDFS 3.0 alternative configuration: Erasure coding replication using Reed-Solomon (3,2) ⟶ 66% storage overhead

GlusterFS: Erasure coding with 4 blocks of data for 2 blocks of parity ⟶ 50% storage overhead

Greedy algorithm: Min Storage

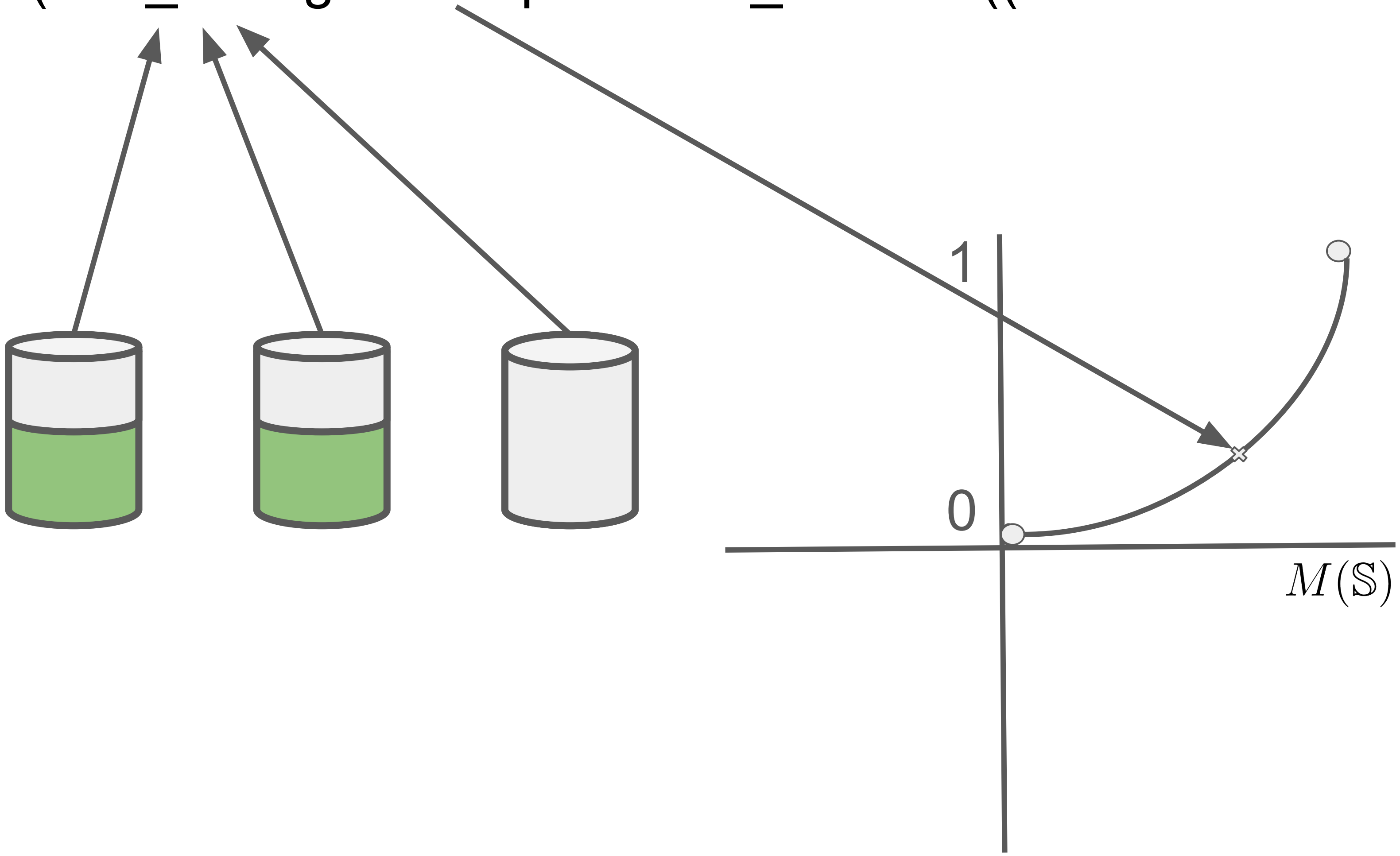
Goal: Minimum use of disk space for each new piece of data to store

Steps to store Data D_i:

1. x = 0
2. N = Number of nodes - x
3. *Candidate* = N nodes with largest available memory
4. Choose K as big as possible such that: $P_{Available}(D_i) \geq Reliability\_threshold$
5. For each node in *Candidate*:
    - If node’s memory < M(D_i)/K
        - x += 1
        - Goto 2.

Proposed algorithm

1. system_saturation = 1 - (free_storage on exponential_function((smallest data,0),($M(\mathbb{S})$,1)))



Proposed algorithm

1. system_saturation = 1 - (free_storage on exponential_function((smallest data,0),($M(\mathbb{S})$,1)))
2. For every combination of nodes, with K as big as possible:
   a. time_overhead = chunking+upload time using (N,K)
   b. space_overhead = (M(D_i)/K)*N
   c. saturation_score = mean system_saturation on each node after adding M(D_i)/K

S1, S2, S3 (N=3, K=1)

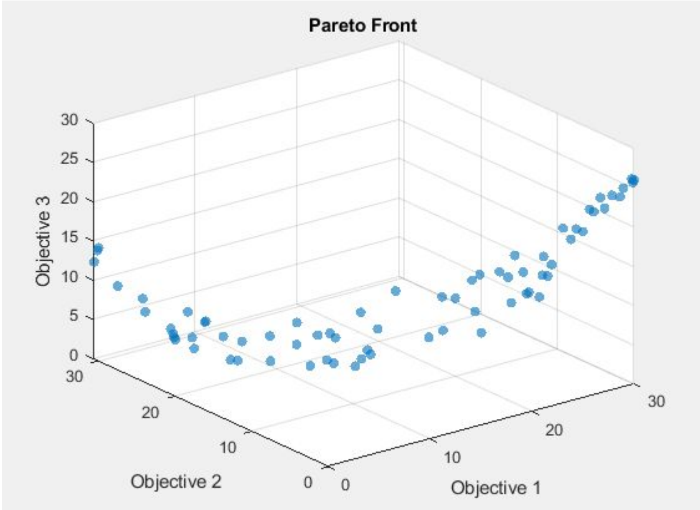
S1, S2, S4 (N=3, K=1)

S1, S2, S3, S4 (N=4, K=2)

...

Proposed algorithm

1. system_saturation = 1 - (free_storage on exponential_function((smallest data,0),($M(\mathbb{S})$,1)))
2. For every combination of nodes, with K as big as possible:
   a. time_overhead = chunking+upload time using (N,K)
   b. space_overhead = (M(D_i)/K)*N
   c. saturation_score = mean system_saturation on each node after adding M(D_i)/K
3. *Candidates* = set of combinations on pareto front using time_overhead, space_overhead and saturation_score



Proposed algorithm

1. system_saturation = 1 - (free_storage on exponential_function((smallest data,0),($M(\mathbb{S})$,1)))
2. For every combination of nodes, with K as big as possible:
   a. time_overhead = chunking+upload time using (N,K)
   b. space_overhead = (M(D_i)/K)*N
   c. saturation_score = mean system_saturation on each node after adding M(D_i)/K
3. *Candidates* = set of combinations on pareto front using time_overhead, space_overhead and saturation_score
4. For each *candidate* j, and for x in time_overhead, space_overhead and saturation_score, compute x_progress = 1- (x(j) - min(x))/(max(x) - min(x))

Proposed algorithm

1. system_saturation = 1 - (free_storage on exponential_function((smallest data,0),($M(\mathbb{S})$,1)))
2. For every combination of nodes, with K as big as possible:
   a. time_overhead = chunking+upload time using (N,K)
   b. space_overhead = (M(D_i)/K)*N
   c. saturation_score = mean system_saturation on each node after adding M(D_i)/K
3. *Candidates* = set of combinations on pareto front using time_overhead, space_overhead and saturation_score
4. For each *candidate* j, and for x in time_overhead, space_overhead and saturation_score, compute x_progress = 1- (x(j) - min(x))/(max(x) - min(x))
5. For each *candidate* j, compute total_score(j) = time_overhead_progress + (space_overhead_progress + saturation_score_progress/2)*system_saturation

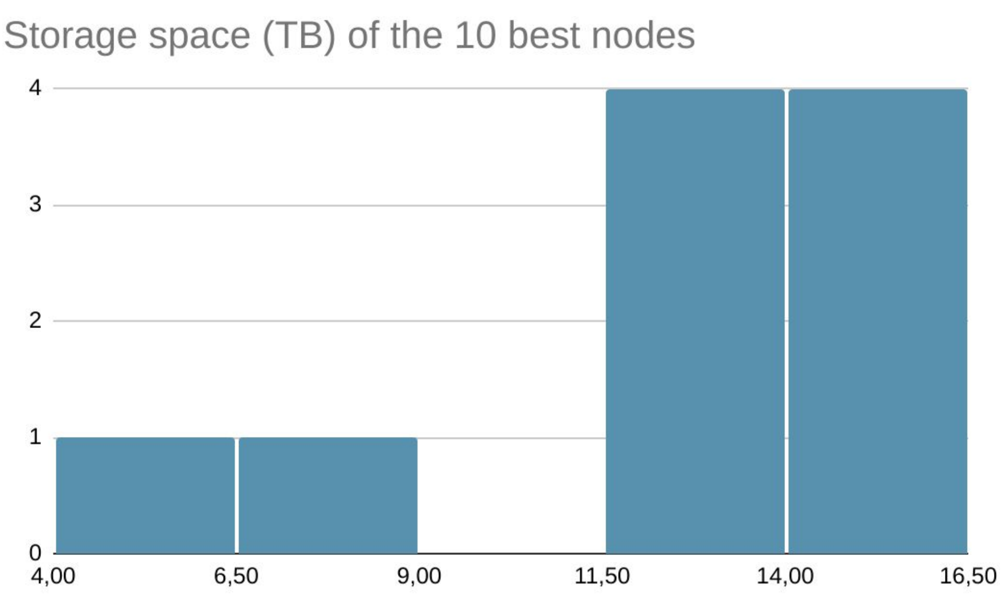
Proposed algorithm

1. system_saturation = 1 - (free_storage on exponential_function((smallest data,0),($M(\mathbb{S})$,1)))
2. For every combination of nodes, with K as big as possible:
    a. time_overhead = chunking+upload time using (N,K)
    b. space_overhead = (M(D_i)/K)*N
    c. saturation_score = mean system_saturation on each node after adding M(D_i)/K
3. *Candidates* = set of combinations on pareto front using time_overhead, space_overhead and saturation_score
4. For each *candidate* j, and for x in time_overhead, space_overhead and saturation_score, compute x_progress = 1- (x(j) - min(x))/(max(x) - min(x))
5. For each *candidate* j, compute total_score(j) = time_overhead_progress + (space_overhead_progress + saturation_score_progress/2)*system_saturation
6. Return combination associated with max(total_score)

Experiments

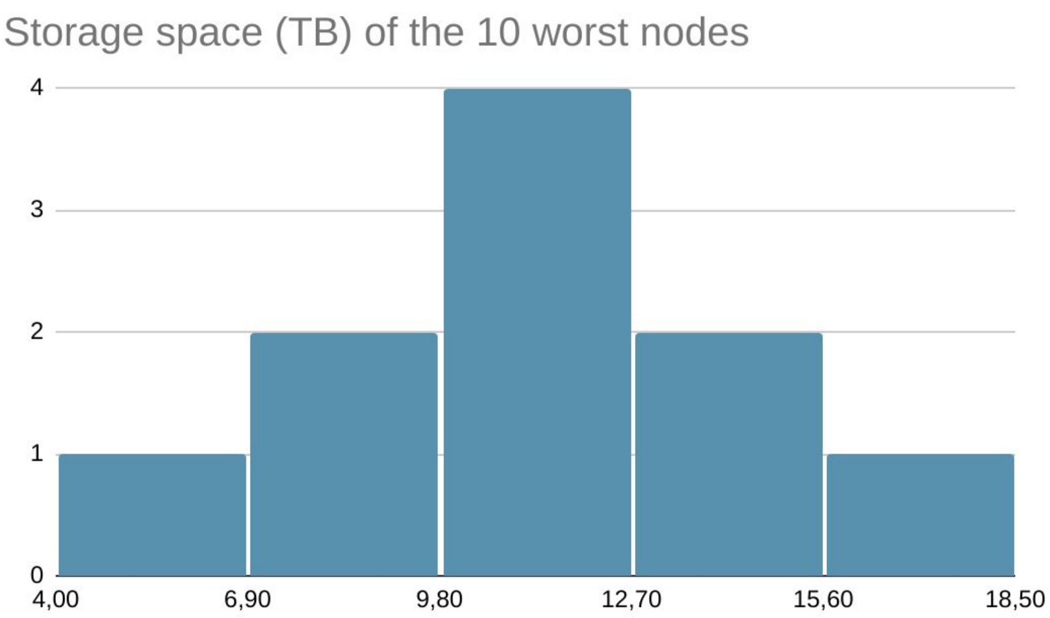
500 data to store using **the 10 most used nodes** or **the 10 worst nodes** from backblaze

Fast and reliable nodes



Mean AFR = 1.2

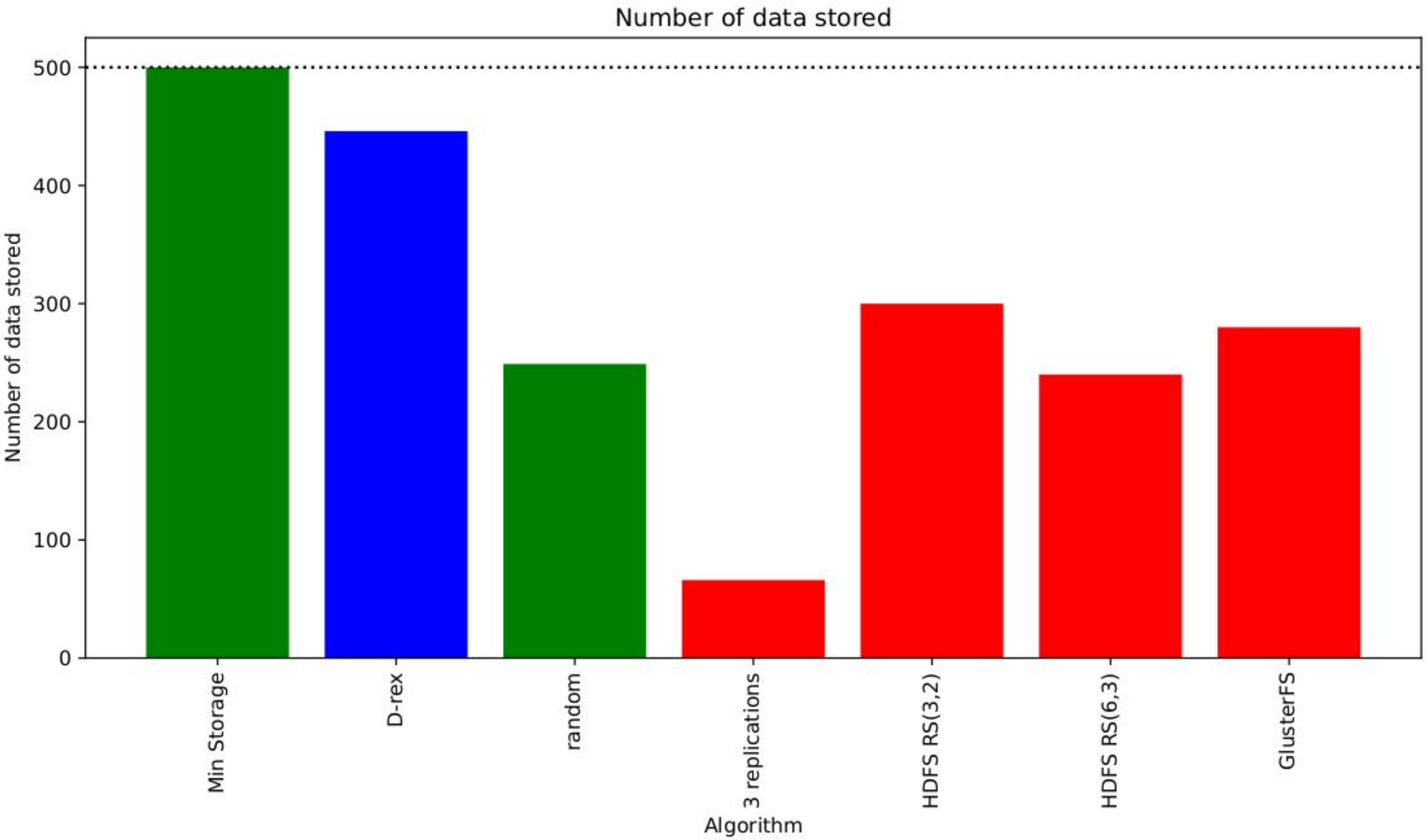
Slowest, and least reliable nodes



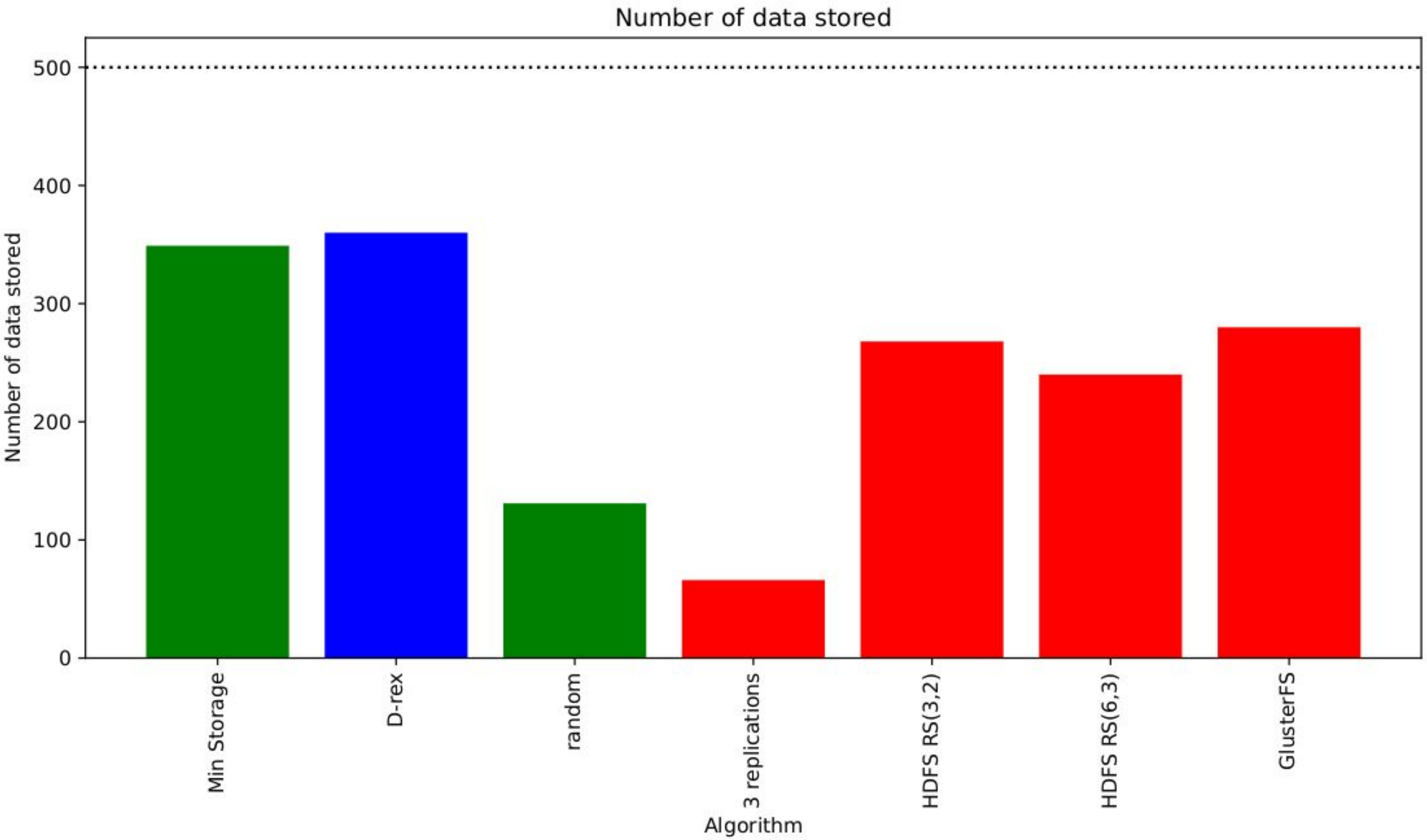
Mean AFR = 6.5

Algorithms: Random - Greedy - D-Rex - HDFS - GlusterFS

500 data to store using **the 10 most used nodes**: Number of data stored

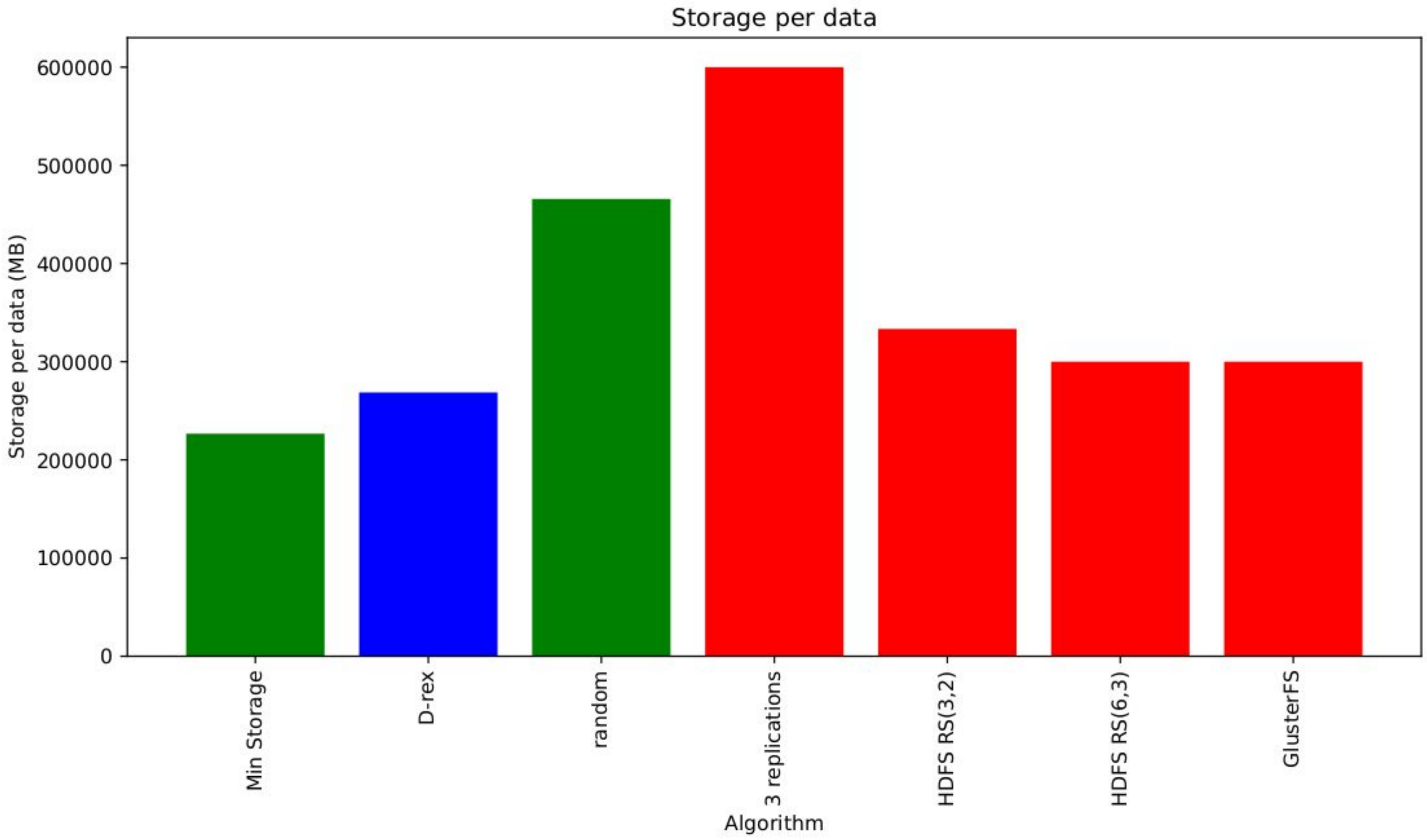


500 data to store using **the 10 worst nodes**: Number of data stored

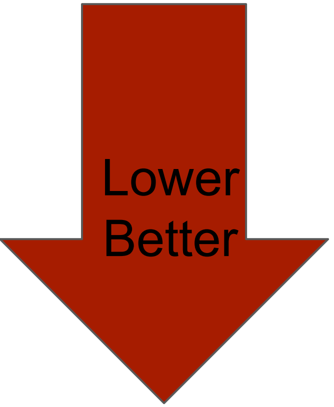


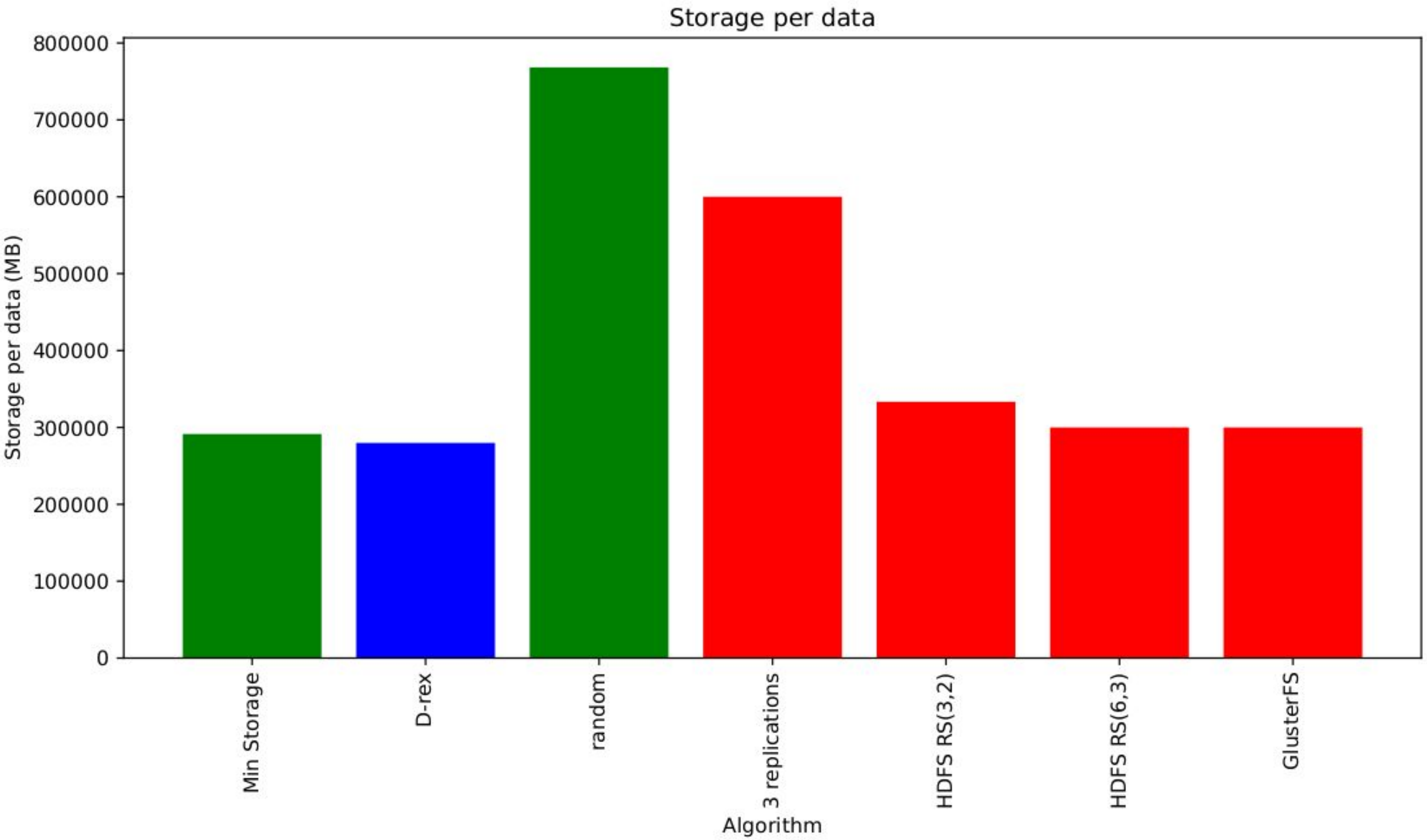
500 data to store using **the 10 most used nodes**: Storage per data

Lower Better

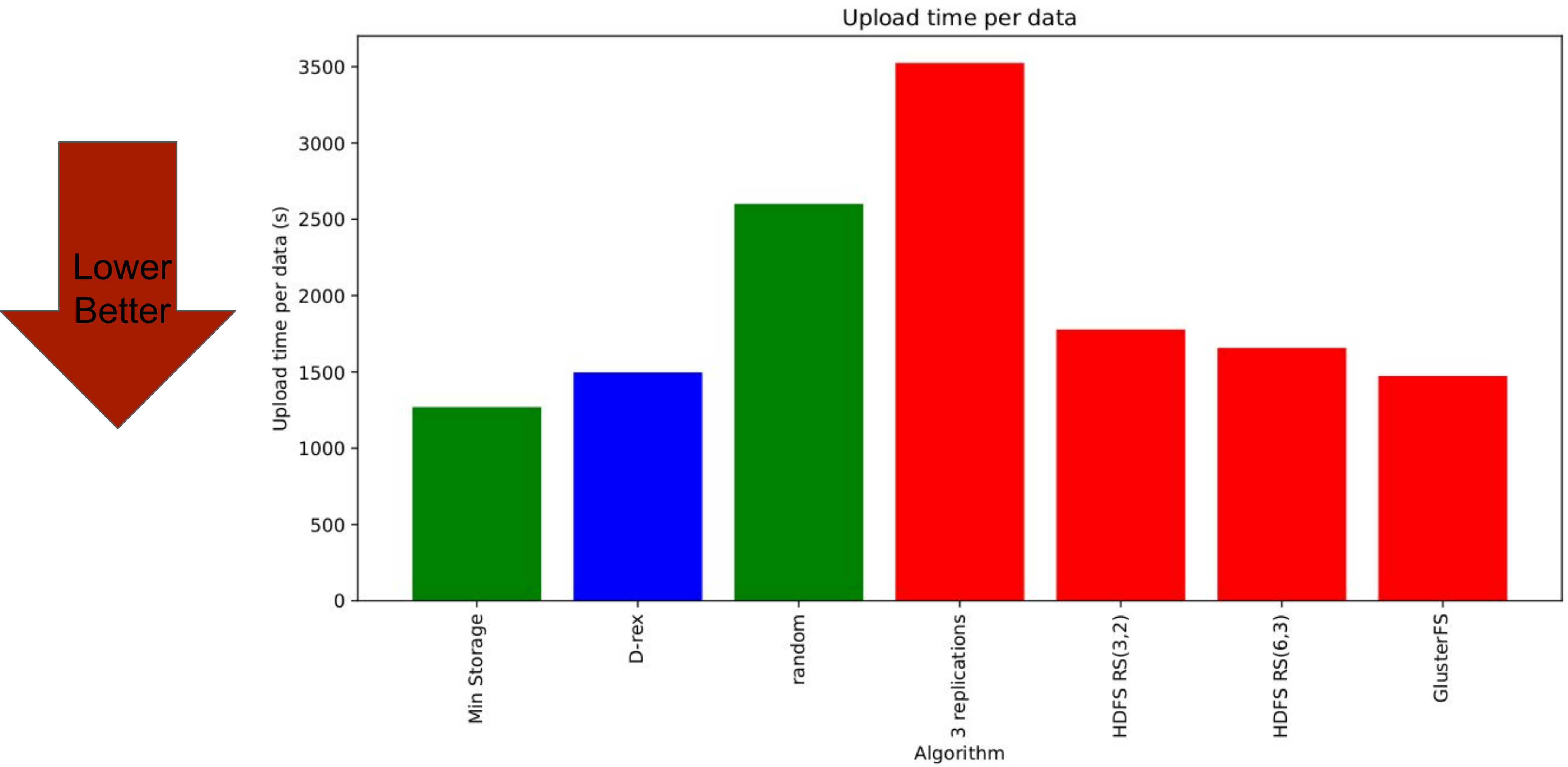


500 data to store using **the 10 worst nodes**: Storage per data



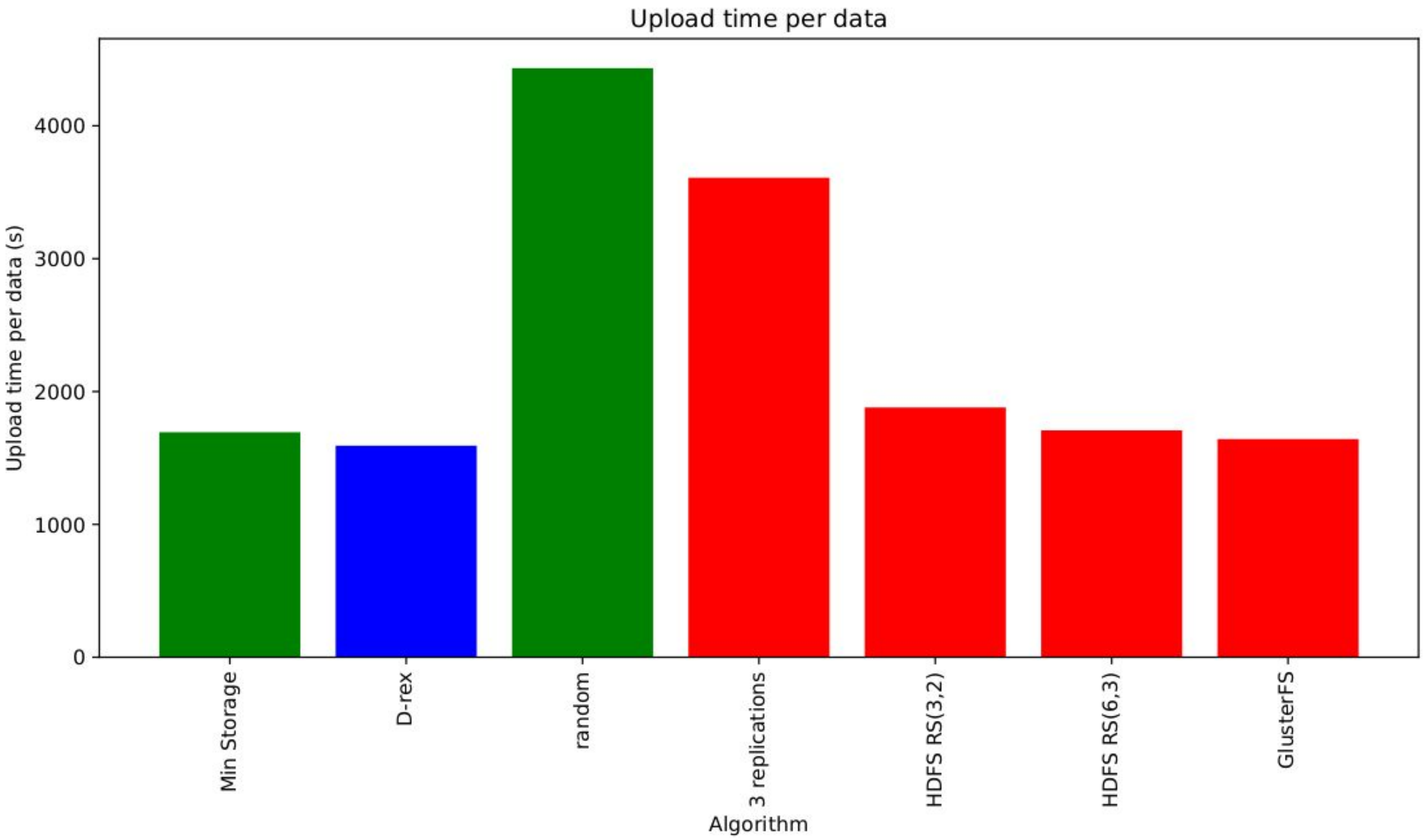


500 data to store using **the 10 most used nodes**: Sum of upload time per data



500 data to store using **the 10 worst nodes**: Sum of upload time per data

Lower
Better



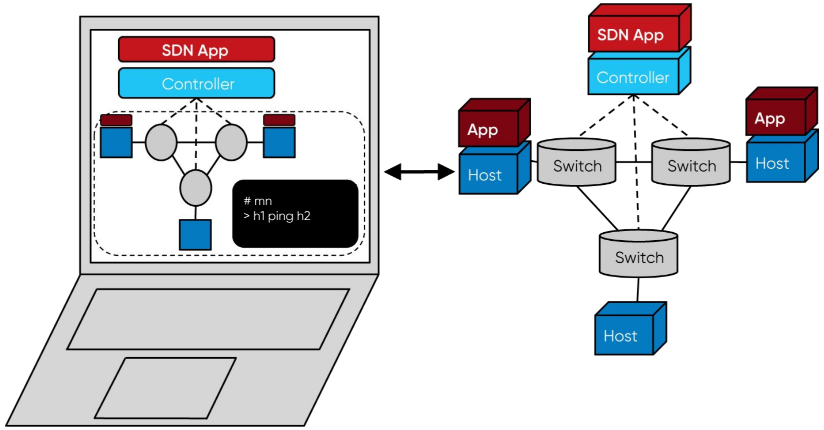
Conclusion

- Summary:
  - Current replication schemes are static and do not manage heterogeneous storage
  - We show that it is possible to reduce chunking + upload times and storage cost by exploiting system saturation and nodes specifications

- Limitations:
  - The use case of slow and unreliable storage nodes is not common
  - Past disk failures rate do not necessarily represent future failure rates

Future works

1. Network simulator (mininet)
2. “Real life” experiments using Globus Compute
3. Real applications (cctv, …)
4. Dynamically add/remove nodes



Emulated Network

Hardware Network

globus