Distributed Storage Mini Project

Group 6: Andreas Kaag Thomsen, Asger Song Høøck Poulsen,
Niels Viggo Stark Madsen, Nikolaj Kühne Jakobsen

January 5, 2025

1 Introduction

The aim of this project is to design and implement a distributed file storage system that leverages redundancy allocation schemes to ensure efficient and robust data management. Distributed storage systems provide a scalable and resilient solution for storing large volumes of data by distributing fragments across multiple nodes. This project focuses on understanding the performance trade-offs associated with different node selection and replication strategies, incorporating the theoretical principles and practical tools covered in coursework.

Our system employs a containerized architecture using Kubernetes for orchestration, with each container representing a node, each defined to facilitate either data management or distribution. To achieve redundancy, we explore three distinct node selection strategies for fragment placement: random placement, MinCopySets, and the Buddy approach. These strategies are evaluated for their efficiency in replication, storage, retrieval, and resilience to node loss. The system design is further enhanced with configurable parameters such as the number of nodes (N), number of replicas (k), and number of fragments (F).

The implementation is validated through rigorous analysis and measurement. Key metrics include file storage and retrieval times, as well as system robustness to node failures. By examining the expected and observed behaviors under various configurations, we aim to provide insights into the trade-offs among different replication strategies, thus addressing the central challenges of distributed storage systems.

2 System Design

We have implemented a distributed file storage system where files are split into smaller fragments and stored across multiple nodes. The system leverages containers for each node, with two distinct types of nodes: `storage_node` and `lead_node`. These nodes communicate with each other using a REST API to manage file storage, retrieval, and replication. The containers are orchestrated using Kubernetes to ease deployment and system scaling. A simplified view of the system design is shown in fig. 1.

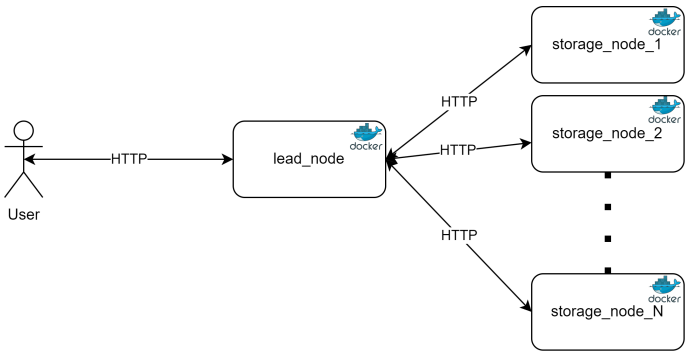

Figure 1: Simplified View of the System Architecture

Node Types and Roles The `storage_node` is responsible for storing and retrieving individual fragments of files in its local file system. In contrast, the `lead_node` manages the higher-level responsibilities of receiving files, fragmenting them into smaller parts, distributing these fragments across the `storage_nodes`, and reassembling the fragments when retrieving the file. The `lead_node` maintains a data structure in which it keeps track of which nodes keep each fragment of each file.

Configurable Parameters The system is designed to be flexible with the following configurable parameters:

- **Number of Nodes (N):** The total number of storage nodes in the system.
- **Number of Replicas (k):** The number of replicas for each fragment.
- **Number of Fragments (F):** The number of fragments to be created from a file.
- **Node Selection Strategy:** How the storage nodes are selected for each fragment.

Kubernetes Configuration The Kubernetes configuration defines two separate deployments: one for the lead node and another for the storage nodes. Each deployment manages the creation and scaling of pods that run specific containers. The lead-node deployment is configured with a single replica, ensuring only one pod is active. This pod runs a container using the therealnestor/ds:lead-node image[1], exposing port 4000 for communication. Resource requests and limits are specified to allocate between 2Gi and 4Gi of memory and 500m to 1 CPU. On the other hand, the storage-node deployment scales with N replicas. Each pod runs a container from the therealnestor/ds:storage-node image, exposing port 5000. The resource configuration for these pods is lightweight, with a memory allocation between 100Mi and 500Mi. Together, these deployments facilitate a distributed system where the lead node orchestrates operations, and the storage nodes handle data distribution.

File Store & Download When a file is uploaded, the `lead_node` splits it into smaller fragments based on the `F` configuration. Each fragment is uniformly sized at $\lfloor B/F \rfloor$ bytes (where B is the total file size), except for the final fragment, which contains the remainder of the file if B is not divisible by F. These fragments are then concurrently uploaded to a set of storage nodes selected using a specific node selection strategy, leveraging threads and HTTP requests for efficiency. During retrieval, the `lead_node` uses metadata to locate the nodes storing the fragments. Each fragment is downloaded from its respective node and reassembled into the original file.

Storage Node Discovery To manage dynamic storage nodes with non-deterministic names and IPs, we have implemented a lightweight node discovery protocol. This protocol retrieves the names and IP addresses of pods labeled as storage-node within the Kubernetes namespace. It begins by fetching a list of pods in the specified namespace, then iterates through each pod, waiting for it to reach the "Running" state within a 5-second timeout. If the pod is running and does not have a deletion timestamp, it updates the list of available storage nodes with the pod's name and IP address. Any changes to the set of available storage node pods trigger an update to the list of active nodes. This discovery protocol operates in a background thread,

[1] A snapshot of the image built by the Dockerfile

running every 5 seconds, ensuring that the system always maintains an up-to-date view of the available storage nodes.

2.1 Node Selection Strategies

When storing the different fragments of a file, it is necessary to determine the appropriate storage nodes for each fragment. To address this, we have implemented three distinct strategies, as described below:

- **Random Selection:** In this strategy, storage nodes are selected randomly for storing the fragments of the file.
- **MinCopySets:** The available storage nodes are randomly divided into $G = \lfloor \frac{N}{k} \rfloor$ disjoint copysets. When a fragment is to be stored, one primary storage node is randomly chosen, and the fragment is then deterministically replicated to the remaining (secondary) nodes within the same copyset [1].
- **Buddy:** The Buddy system is similar to MinCopySets with the storage nodes being divided into disjoint replication groups. The key difference is that nodes are chosen randomly within each replication group as opposed to being chosen deterministically. This implies that $G \leq \lfloor \frac{N}{k} \rfloor$, where G is the number of buddy groups. Each group consists of at least k nodes, ensuring that replication requirements are met. To ensure balance and difference from MinCopySets, we have chosen to set $G = \left\lfloor \sqrt{\frac{N}{k}} \right\rfloor$

2.2 Node Loss

To enable the termination of storage nodes, we have introduced an endpoint on the lead node that accepts the number s of nodes to be killed as input. If $s \leq N$, where N is the total number of available storage nodes, the system randomly selects s storage nodes from the N available ones and proceeds to delete the corresponding pods.

2.3 Loss Quantification

To assess the loss of files, we have introduced another endpoint on the lead node that quantifies the number of lost files based on the availability of their fragments across storage nodes. The algorithm for doing so is shown in algorithm 1. It first retrieves the current list of storage nodes and then iterates through all files and their associated replica metadata. For each file, it checks whether the fragments (represented by nodes) are available in the current storage nodes. If not all fragments of a file are found, the file is considered lost.

Algorithm 1 Quantify File Loss

Retrieve the list of storage node pods
$files\_lost \leftarrow 0$
for each file in $file\_metadata$ **do**
    $fragments\_found \leftarrow \emptyset$
    for each replica in the file's replicas do
        for each fragment in the replica do
            if fragment node exists in the list of storage nodes **then**
                $fragments\_found \leftarrow fragments\_found \cup \{fragment\}$
            end if
        end for
    end for
    if $fragments\_found$ is less than the expected number of fragments **then**
        $files\_lost \leftarrow files\_lost + 1$
    end if
end for
Return $files\_lost, total\_files$

3 Analysis

In the following, we answer the three analysis tasks. Throughout the analysis, we will use the same notation as in section 2, and in addition the following notations, which are all exactly as given in the project description:

- B = size of the original file in bytes.
- Splitting the file into 4 equal fragments results in each fragment being $\frac{B}{4}$ bytes.
- R = connection bandwidth in bits per second (same speed among any pair of nodes, and also between any node and the client i.e. all transfers occur over this bandwidth).

3.1 File Storage Time

For replication, there is no erasure coding overhead-each fragment is simply copied as-is. We assume that the lead node is the **only** sender. Furthermore, in many simple designs, even if you do parallel TCP connections to the k nodes, the lead node's aggregate outgoing bandwidth is limited to R.

Amount of Data Sent: We have that each fragment size is $\frac{B}{4}$ in bytes and $\frac{B}{4} \times 8 = 2B$ in bits. Thus, the total data sent by the lead node to store all replicas of all 4 fragments is:

$$\underbrace{4 \times k}_{\text{(4 fragments each with k copies)}} \times \left(\frac{B}{4} \times 8\right) = 4 \times k \times 2B = 8Bk \quad \text{(bits)}$$

Time to Store: Since the lead node can push data out at most R $\frac{\text{bits}}{s}$ (assuming it is the bottleneck), and we have $8Bk$ bits to send, the storing time is:

$$T_{\text{store}} = \frac{8Bk}{R} \tag{1}$$

That accounts for storing the data.

Note that, one may also factor in the local time to split the file into 4 fragments and select the nodes. Often, if disk I/O or local CPU is reasonably fast, that overhead is negligible compared to wide-area network speeds. If needed, you can add a small constant or a term $\frac{B}{\text{local speed}}$, but typically the network time is the main cost.

3.2 File Download Time

Amount of Data Sent to the Client Regardless of which k copies we use, the client needs all 4 fragments (one copy of each) to reassemble the original file. Each of the four fragments is $\frac{B}{4}$ bytes summing up to the total file size of B bytes ($8B$ bits).

Time to Download Since the lead node is assumed to have an aggregate outgoing bandwidth of R bits/s to the client, sending $8B$ bits of data will take

$$T_{\text{download}} = \frac{8B}{R}$$

which is the total time from when the client requests the file until the last byte of the file arrives at the client.

Why It Does Not Change Across the Three Node-Selection Schemes Regardless of the node selection scheme used, k identical copies of each of the 4 fragments are stored. When downloading, only *one* copy per fragment is fetched (via the lead node). Since the lead node's outgoing bandwidth is the bottleneck at rate R, the total transfer is $8B$ bits, thus:

$$T_{\text{download}} = \frac{8B}{R}$$

Optional Variation: Parallel Downloads If the client were to pull each fragment *directly* from different storage nodes, and each node–client link could run at R bits/s in parallel, then all 4 fragments could be downloaded simultaneously. In that ideal scenario:

- Total file size: $8B$ bits
- Effective aggregate bandwidth (4 nodes, each at R): $4R$

Then

$$T_{\text{download, parallel}} = \frac{8B}{4R} = \frac{2B}{R}$$

However, under our assumption that *the lead node is the only sender with a single R bits/s link*, the formula remains

$$\boxed{T_{\text{download}} = \frac{8B}{R}}$$

3.3 Loss Analysis

We wish to determine for which values of s (the number of failed/stopped nodes) we expect files to be lost, given the same assumptions as in section 3.1.
We define a key point:

A file is lost if *all* replicas of at least one fragment are lost.

Hence, for a single fragment that has k replicas on k distinct nodes, you lose that fragment if (and only if) those same k nodes are all among the failed set of s nodes.

3.3.1 Deterministic Threshold: $s = k$

It is only possible to observe a file loss when at least k nodes have failed, i.e.:

$$L(s) = \begin{cases} \text{No file can be lost} & s < k \leq N \\ \text{File can be lost} & s \geq k \leq N \end{cases}$$

This means that the minimum s to allow any data loss is $s = k$ under the assumption that all k replicas are distributed on k distinct nodes. In practice, this is not guaranteed across all allocation schemes.

3.3.2 Probability of Loss (Random Failures) as a Function of k and N

Often, we assume failures are random: exactly s nodes fail simultaneously and are chosen uniformly among the N total nodes. We analyze the probability that a particular file is lost under the three schemes.

Random Placement: Each of the 4 fragments is placed on k distinct nodes (assuming $k \leq N$) chosen randomly out of N. If s nodes fail simultaneously, the probability that a single fragment is completely lost is:

$$P(\text{fragment lost}) \;=\; \frac{\binom{s}{k}}{\binom{N}{k}} \quad (s \geq k).$$

where $\binom{s}{k}$ is the number of combinations of failed replicas and $\binom{N}{k}$ is the maximum number of combinations of replicas. Using this, we can write the probability that at least one fragment is lost in the case of s simultaneous node failures:

$$P_{random}(\text{file lost}) \;\approx\; 1 - \left(1 - \frac{\binom{s}{k}}{\binom{N}{k}}\right)^4$$

Notice that if $s < k$, then $p = 0$, so no loss occurs. As s grows beyond k, the probability of losing a fragment (and thus a file) increases. The formula shows that the more ways we can choose the nodes for the replicas, $\binom{N}{k}$, the larger the probability for file loss gets.

Min CopySets: Min CopySets aims to reduce the number of ways we can choose the nodes for the replicas, $\binom{N}{k}$, and thereby lowering the probability of file loss. However, the approach trades this off with a higher amount of data loss when an event happens. Comparing with the random selection approach, we expect that $P_{minCopySets} < P_{random}$ but when the loss event happens, the amount of data lost is greater, i.e. $D_{lost,minCopySets} > D_{lost,random}$.

Buddy: The buddy approach can be viewed as a tradeoff between Min CopySets and the random selection approach where nodes can be selected at random within a fixed number of copysets. This means that the probability of data loss is higher that for Min CopySets but when the loss event happens, the amount of data lost is less. That is, we expect that $P_{minCopySets} < P_{buddy} < P_{random}$ and that $D_{lost,minCopySets} < D_{lost,buddy} < D_{lost,random}$.

4 Results

In this section we present the experimental results of our distributed file system, showcasing store and download times as well as data loss when nodes are killed.

4.1 File Store & Download Times Using Different Node Selection Approaches

To evaluate the impact of different values of N (number of storage nodes), B (file sizes), and various node selection strategies on the performance of the distributed file storage system, we conducted experiments across the following configurations: $N \in \{3, 6, 12, 24\}$ (number of storage nodes), $k = 3$ (number of replicas), $B \in \{100\text{KB}, 1\text{MB}, 10\text{MB}, 100\text{MB}\}$ (file sizes). For each configuration, we uploaded and downloaded 100 files, measuring the total time required for successful upload and download operations. Results for file sizes 100kb, 1mb, and 10mb are shown in figs. 2 and 3. Since the results for 100mb are significantly different they are shown separately in figs. 4 and 5.

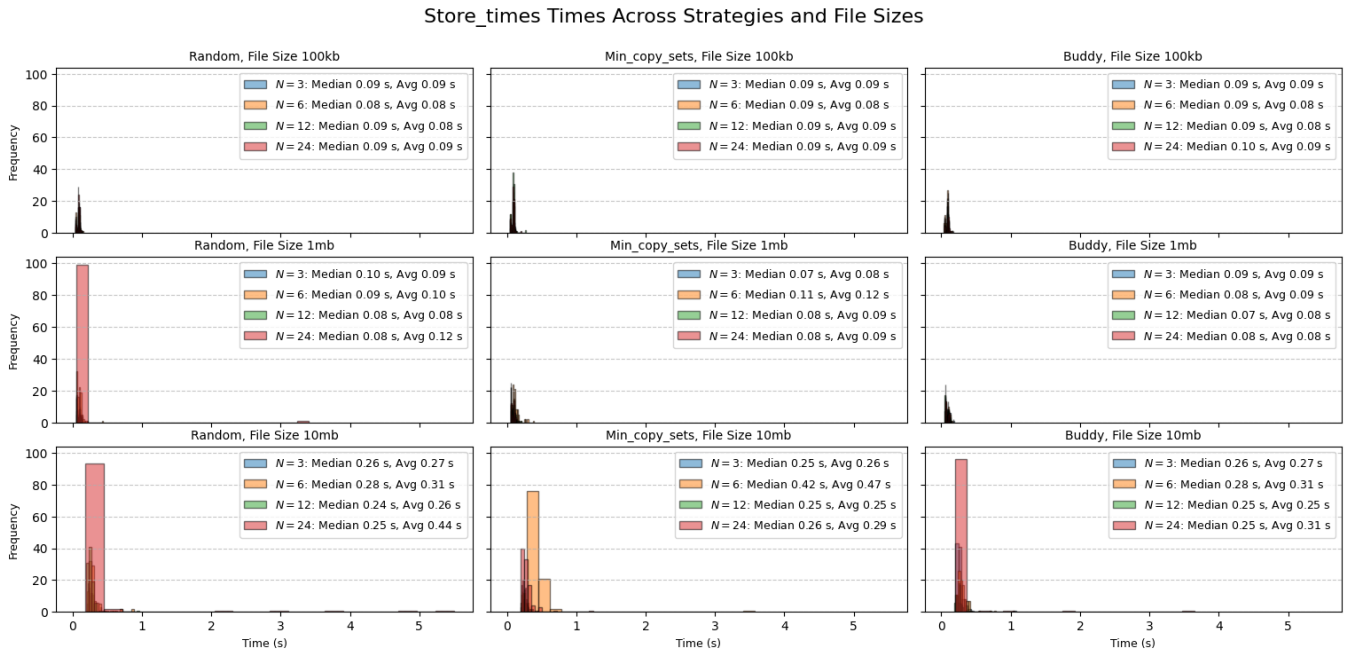

Figure 2: File storing times across node selection strategies and file sizes.

Figures 2 to 5 show that the different node selection strategies do not impact the store and download times. At equal file sizes and N, the different node selection strategies almost always exhibit similar behavior.

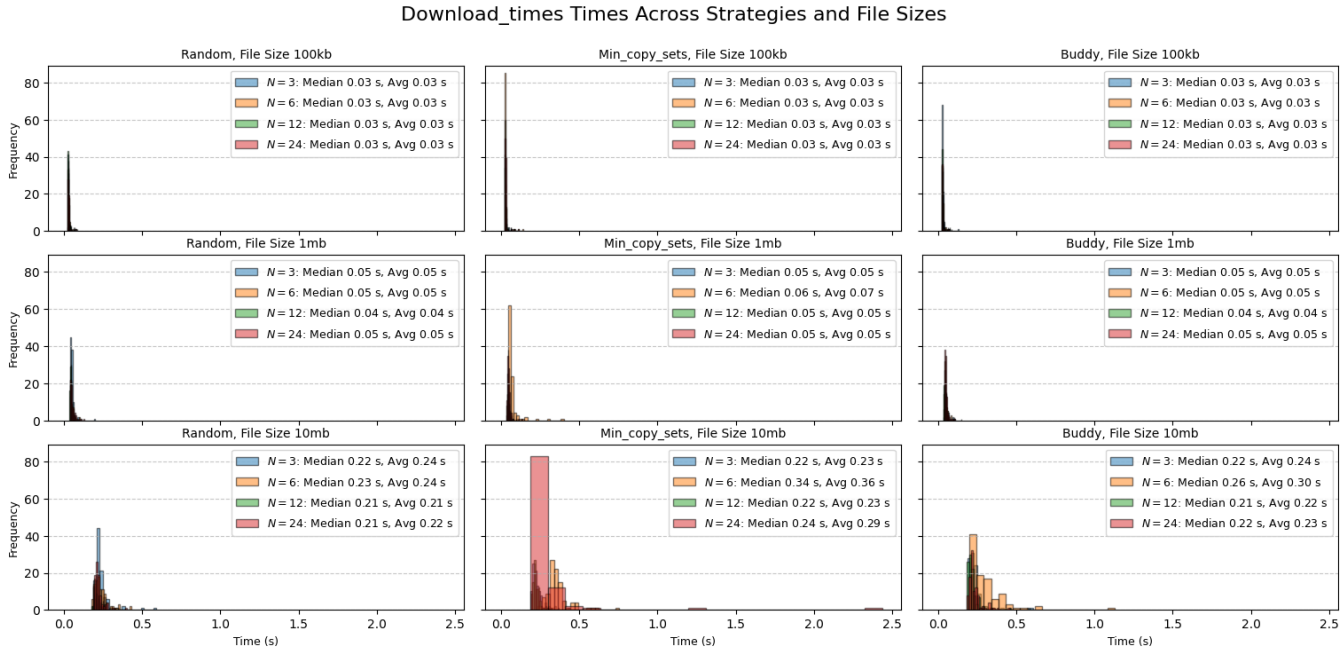

Figure 3: File download times across node selection strategies and file sizes.

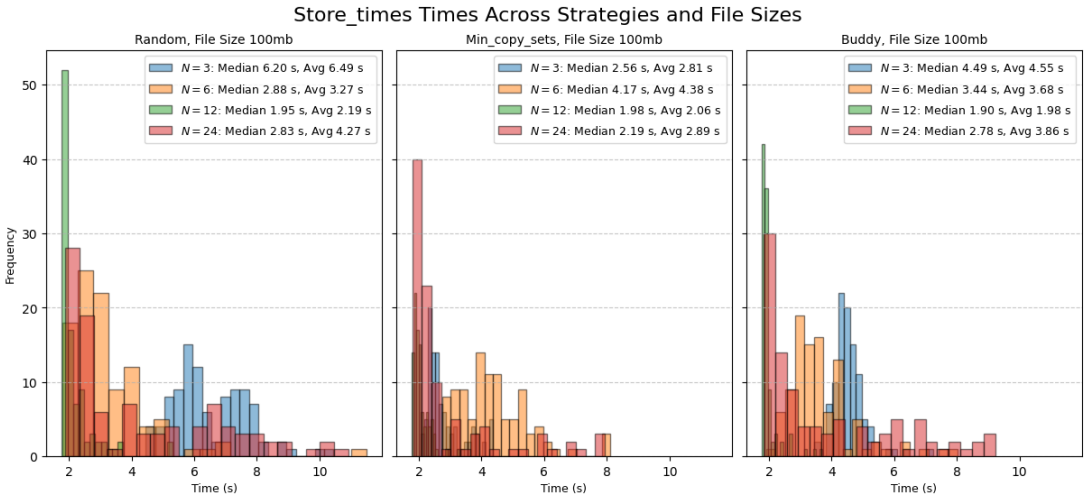

Figure 4: File storing times across node selection strategies and file sizes.

The number of storage nodes (N) appears to have a moderate impact on performance. For smaller file sizes (100KB and 1MB), changes in N have minimal to no effect on store and download times. However, for larger file sizes (10MB and 100MB), the impact of N becomes more noticeable. Generally, configurations with larger numbers of storage nodes ($N = 12$ or $N = 24$) tend to result in faster store and download times, though exceptions exist (e.g., for specific strategies such as min_copy_sets with 10MB files). This

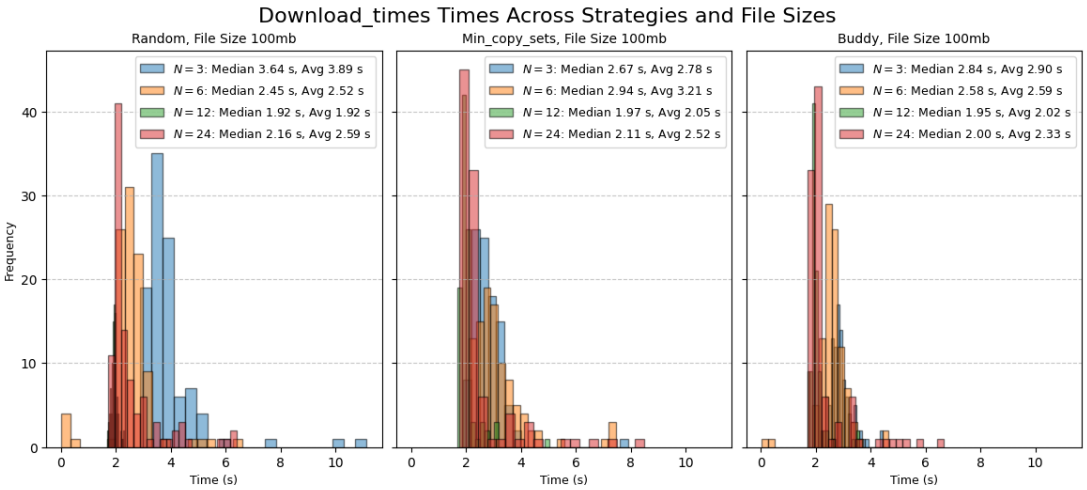

Figure 5: File download times across node selection strategies and file sizes.

trend is likely due to the distribution of workload. With smaller N, some nodes handle multiple fragments, potentially becoming bottlenecks. In contrast, with larger N, fragments are distributed more evenly across nodes, reducing the load on individual nodes. For larger file sizes, where each fragment takes more time to store or download, this effect becomes more pronounced, further emphasizing the benefits of a higher N.

The file sizes have the most important effect on the store and download times. This makes sense as larger files are split into larger fragments, increasing the total amount of data that must be transmitted and stored across the storage nodes. Each fragment requires network communication and write operations, both of which scale with the file size. Consequently, as file sizes grow, these operations take longer, leading to a significant increase in store and download times. This effect is especially evident for the largest file size (100MB).

4.2 Data Loss Measurements

To determine how robust the system is to node loss across the different node selection strategies and number of nodes, we have carried out experiments with the following configurations: $N \in \{12, 24, 36\}$, $S \in \{2, 3, 4, 6, 8, 10\}$ (number of lost nodes) and the three different node selection strategies. For each configuration, we have uploaded 100 files of 100KB each and deleted S nodes after which we quantified the file loss. All S values have been tested with a new set of 100 files. Thus they don't accumulate through the tests for each strategy. The results are summarized in fig. 6.

The figure illustrates that the random strategy results in file loss even at relatively low values of s. For instance, with $N = 12$ and $N = 24$, file loss begins after just two nodes are killed. Overall, the random strategy exhibits the highest file loss across most values of s, with the exception of $s = 10$. In the MinCopySets strategy, we see good results in fig. 6 where MinCopySets can lose the most nodes without losing any data in all three tests. In the case of $N = 36$ not a single file is lost in the test. However, when the loss event happens, we see a large jump in the file loss. The Buddy strategy performs slightly worse than MinCopySets but still shows strong resilience. A relatively large number of nodes can be killed before any data is lost, and the strategy also offers the advantage of simplified repair due to its structured pairing

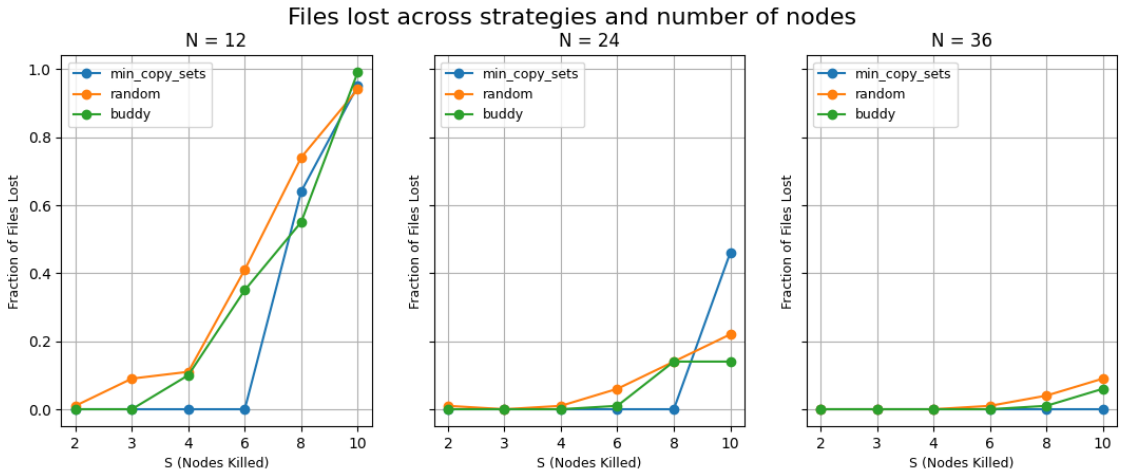

Figure 6: Files lost as a function of nodes killed across different number of nodes and node selection strategies.

of nodes.

In summary, the observed behavior aligns with expectations outlined in section 3.3, where data loss occurs most rapidly with the random strategy, slowest with MinCopySets, and the Buddy strategy falls in between. Additionally, MinCopySets experiences the highest data loss per event, while random has the least data loss per event, with Buddy once again positioned between the two.

References

[1] Asaf Cidon et al. "MinCopysets: Derandomizing replication in cloud storage". In: *Proc. 10th USENIX Symp. NSDI*. 2013, pp. 1–5.