ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
Master's Degree in Computer Science

Merkle-tree-based integrity verification and repair protocol for distributed storage systems

Supervisor:
Prof. Özalp Babaoğlu

Author:
Santo Cariotti

II Session
Academic Year 2024/2025

Abstract

This thesis wants to show bla bla.

Sommario

This thesis wants to show bla bla.

Contents

1 Introduction

This thesis emerged from a practical problem encountered during the development of a modern cloud storage: how can a geo-distributed storage system efficiently detect file corruption without assuming that all nodes are always online? The answer, as explored in this work, lies in rethinking corruption detection through the use of hierarchical data structures and consensus algorithms.

1.1 Motivation

File corruption can silently compromise stored data, reduce reliability, or increase recovery costs. Traditional approaches such as checksums, replication, or RAID [1] provide partial solutions, but they also show limitations when applied to large-scale and geo-distributed environments. In particular, these methods often assume constant node availability or require full scans that become increasingly expensive as data grows.

During my internship at Cubbit[1], the first geo-distributed cloud storage provider, I studied the company's existing corruption detection system, which relies on checksum-based verification. While functional, this approach has two key weaknesses: it assumes that nodes are always online, and it requires heavy operations to verify integrity. Moreover, Cubbit stores files using Reed-Solomon codes, meaning that data is distributed into shards across multiple agents, and files are reconstructed from subsets of those shards. This introduces additional challenges, since nodes holding useful shards may be offline at any time.

Motivated by these challenges, I explored an alternative approach based on Merkle trees. Merkle trees, widely used in blockchain and distributed databases, allow efficient and hierarchical verification of large datasets: instead of re-checking entire files, integrity can be confirmed by verifying only a logarithmic number of hashes along a path in the tree. Combined with a consensus algorithm, this structure allows folder-level integrity verification even in clusters where nodes frequently join and leave.

[1] `https://cubbit.io`

1.2 Problem Statement

Verifying file integrity at the file level quickly becomes inefficient as the dataset grows. In Cubbit's environment, the problem is further complicated by node churn: data may be temporarily unavailable, yet the system must still guarantee correctness.

Cubbit's use of Reed-Solomon coding ensures that files can be reconstructed even when some shards are missing or corrupted. However, Reed-Solomon alone does not provide a way to verify the integrity of individual shards. This means that a reconstructed file could include corrupted data without the system being able to detect where the corruption occurred.

Merkle trees address this gap, providing an efficient mechanism to detect and localize corruption by verifying a logarithmic number of hashes along an authentication path. The key challenge lies in constructing and verifying Merkle trees in a distributed setting, where some shards are stored on agents that may be offline.

This thesis addresses the need for a corruption detection system that:

- avoids repeated full-file scans by verifying integrity at the folder and subfolder level;
- tolerates offline nodes by storing and synchronizing integrity metadata across the cluster;
- integrates with Cubbit's existing Reed-Solomon codes, ensuring that verification is possible even during partial uploads and recoveries.

The central question is: how can a Merkle-tree-based corruption detection system, combined with Raft consensus and compatible with Reed-Solomon codes, provide efficient, scalable, and fault-tolerant integrity verification in a geo-distributed cluster?

1.3 Research Objectives

The primary objective of this thesis is to design and evaluate a corruption detection system for distributed storage clusters that is both efficient and resilient to node failures. Unlike checksum-based approaches, the proposed system leverages:

- an ad-hoc Merkle tree library optimized for folder-based hierarchies;
- coordination via Raft consensus to ensure consistent integrity metadata across nodes;

- integration with Cubbit's Reed-Solomon-based infrastructure to handle partial uploads and recoveries;

The overarching goal is to demonstrate that an optimized, folder-oriented Merkle tree, combined with consensus, can provide a scalable, fault-tolerant, and efficient alternative to classical checksum-based corruption detection in geo-distributed clusters.

1.4 Structure of the Document

This document is organized as follows:

Chapter 2 provides the necessary background and serves as a literature review. It examines the key technologies and theoretical foundations relevant to this work, including Merkle trees, cryptographic hash functions, consensus algorithms, and the core component of the Cubbit infrastructure. For each of these elements, different solutions proposed in the literature are analysed and compared. The chapter then introduces the selected solutions, together with the rationale for their adoption, explaining why they are best suited to the corruption detection system under consideration.

Chapter 3 describes the architecture of the proposed corruption detection system. It explains how the components introduced in Chapter 2 are combined and how they interact within the overall design.

Chapter 4 presents the prototype implementation and the experimental evaluation. The system is tested under different scenarios, varying both the number of agents in the cluster and the number of files stored per agent. Controlled corruption is deliberately injected into the data, and the evaluation measures the elapsed time between the initiation of a corruption check and the correct identification of the corrupted file.

Chapter 5 concludes the thesis by summarizing the main contributions, highlighting limitations, and outlining possible directions for future work.

2 Background

This chapter is intended to provide the reader with some background information on the technologies used in writing this thesis project.

Some of them are essential to better know why the given Merkle tree solution is useful and works for Cubbit's infrastructure.

2.1 Merkle Trees

Merkle trees [2] are a fundamental data structure first introduced by R. Merkle in his PhD dissertation. This section presents the theoretical foundations of Merkle trees, their construction, their practical applications, and the rationale for the implementation chosen in this thesis.

A Merkle tree is a binary tree T of height H with 2^H leaves and $2^H - 1$ internal nodes. Each leaf stores the cryptographic hash of the underlying data, rather than the raw data itself. The same cryptographic hash function is applied recursively at internal nodes, which store the hash of the concatenation of their two children. For a more detailed discussion of collision resistance in cryptographic hash functions, see [3].

Formally, given two child nodes n_{left} and n_{right}, their parent node is defined as:

$$n_{\text{parent}} = f(n_{\text{left}} \,||\, n_{\text{right}}) \tag{2.1}$$

where $||$ denotes bit-string concatenation and f is a cryptographic hash function.

Consider now a Merkle tree of height $H > 2$. A leaf node is indexed by $\phi \in \{0, \ldots, 2^H - 1\}$. A node at height h and position j (counting from left to right) is denoted as $y_h[j]$, where $h = 0, \ldots, H$ and $j = 0, \ldots, 2^{H-h} - 1$. Given a cryptographic hash function $f : \{0,1\}^\star \mapsto \{0,1\}^n$, the recursive definition of an internal node is:

$$y_h[j] = f\big(y_{h-1}[2j] \,||\, y_{h-1}[2j+1]\big). \tag{2.2}$$

The root node of the tree, known as the *Merkle root*, serves as a compact commitment to all data contained in the leaves. Because hashes propagate upwards, even a single-bit modification in any leaf causes a change in the root hash.

This property makes Merkle trees powerful tools for integrity verification in large, distributed datasets.

2.1.1 Merkle Proofs

One of the most powerful features of Merkle trees is the ability to prove that a given piece of data is part of a larger set, without revealing or recomputing the entire dataset. Given a Merkle tree leaf, one can reconstruct the root by traversing the path to the root and successively combining the node with its siblings.

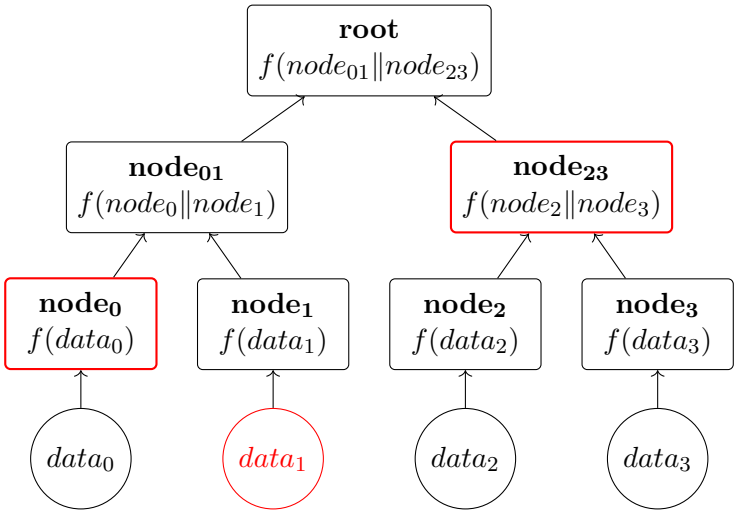

Figure 2.1: Merkle tree authentication path for $data_1$. Leaves are hashed as $node_i = f(data_i)$, and internal nodes are computed as $f(node_{\text{left}} || node_{\text{right}})$ (Equation 2.1). The proof requires only the sibling nodes $\{node_0, node_{23}\}$ to recompute the root.

For each height $h < H$, we define $Auth_h$ to be the value of the sibling node along the path from the leaf to the root. The set of all such siblings $\{Auth_h\}_{h=0}^{H-1}$ is called the *authentication path*. With this path, anyone can recompute the root and verify inclusion by comparing against the published Merkle root.

For instance, Figure 2.1 shows the authentication path corresponding to the second leaf.

This property, known as a *Merkle proof*, enables efficient verification of data integrity. In a naive implementation, the entire Merkle tree is stored in memory. In such a case, generating a proof for a leaf - as illustrated in Algorithm 1 - involves traversing only the path from the leaf to the root. This reduces proof generation from a potentially linear scan of all data ($O(n)$) to a logarithmic traversal of the tree ($O(\log n)$).

Algorithm 1: Merkle Proof Generation

Input: Leaf index i, full tree levels $L_0, L_1, \ldots, L_H$
Output: Merkle proof π for leaf l_i (or `None` if i invalid)

$\pi \leftarrow \emptyset$
$current \leftarrow i$
foreach $level \in \{L_0, \ldots, L_{H-1}\}$ **do**
  $sibling\_index \leftarrow \min(current \oplus 1, |level| - 1)$
  $sibling \leftarrow level[sibling\_index]$
  $position \leftarrow$ `Left` if $sibling\_index < current$, else `Right`
  append $(sibling.hash, position)$ to π
  $current \leftarrow \lfloor current/2 \rfloor$
end
return π

Merkle Proof Verification algorithm for a proof π is illustrated in Algorithm 2.

Algorithm 2: Merkle Proof Verification

Input: Data d, proof π, expected root R, hash function f
Output: `true` if valid, `false` otherwise

$h \leftarrow f(d)$
foreach $(sibling, position)$ *in* π **do**
  if $position = Left$ **then**
    $h \leftarrow f(sibling||h)$
  else
    $h \leftarrow f(h||sibling)$
  end
end
return $h = R$

Complexity Analysis

- **Proof generation:** $O(\log n)$, because only the sibling nodes along the path from leaf to root are collected.
- **Proof verification:** $O(\log n)$, as each step requires a single hash operation per tree level.
- **Memory:** $O(n)$ to store the full tree in memory, which allows logarithmic-time proof generation.

This approach ensures that Merkle proofs remain efficient even for large datasets, while keeping the implementation simple and compatible with our folder-level integrity checks across a Raft-coordinated cluster.

2.1.2 Applications

Merkle trees are widely used in distributed systems to ensure data integrity:

- **Blockchains**: Bitcoin [4], Ethereum [5], and other systems use Merkle roots to verify transactions efficiently.
- **Version control systems**: Git stores commits as Merkle trees, ensuring history integrity.
- **Distributed storage**: Systems such as IPFS [6] and Amazon DynamoDB [7] use Merkle trees for consistency checks and conflict resolution.

2.1.3 Alternative Implementations

In the literature, several advanced variants of Merkle trees exist, such as XMSS (eXtended Merkle Signature Scheme) [8] and the BDS (Buchmann-Dahmen-Szydlo) traversal algorithm [9]. These schemes were developed in the context of *post-quantum cryptography* and digital signatures. XMSS is standardized by the IETF (RFC 8391) and provides strong security guarantees by organizing one-time signatures (OTS) under a large Merkle tree, where the root of the tree serves as the public key.

In XMSS, to sign a message i, the authentication path of the i-th leaf is needed. In a native way, recomputing this path would require rebuilding large parts of the tree, which becomes impractical when the tree contains millions of leaves. To solve this, the BDS traversal algorithm was introduced: it incrementally maintains and updates the authentication path in $O(h)$ time and $O(h)$ space (where $h = \log_2(n)$ is the tree height). This makes XMSS practical for very large trees.

In our case, however, the scenario is fundamentally different. We are not designing a post-quantum signature scheme but a corruption detection system for folders in a distributed storage cluster. The size of our Merkle trees is modest: typically on the order of tens of leaves per folder. For trees of this size:

- Proofs can be recomputed directly, without significant computational overhead.
- The space-time optimizations of BDS provide no practical benefit.

- The additional complexity of XMSS and BDS would introduce unnecessary implementation overhead.

For this reason, this thesis opted for a *simple* Merkle tree implementation, applied independently at the folder and sub-folder level. This keeps the system lightweight, efficient, and easy to integrate with a consensus mechanism. It also avoids the pitfalls of managing very large Merkle trees (as in XMSS) or the stateful requirements of post-quantum signature schemes, which are irrelevant in our use case.

2.2 Cryptographic Hash Functions

In the previous section, Merkle trees were discussed in detail, with particular attention to the use of cryptographic hash functions for node construction. One of the questions that emerged during my internship was: *"what is the fastest cryptographic hash function?"*.

This section explores the motivation behind testing different cryptographic hash functions (SHA-256, Keccak-256, and BLAKE3) within the context of this project and why, among these, BLAKE3 was selected as the primary candidate for benchmarking in this thesis due to its performance and modern design.

2.2.1 SHA-256

SHA-256 is part of the SHA-2 family of cryptographic hash functions, standardized by NIST in 2001 [10]. It produces a 256-bit output from input messages of arbitrary length and is widely used in security protocols such as TLS, digital signatures, and blockchain systems like Bitcoin.

The algorithm processes data in 512-bit blocks using 32-bit words. On 32-bit architectures, this design choice makes SHA-256 relatively efficient. However, on modern 64-bit CPUs, the reliance on 32-bit operations leads to extra instructions, making it slower than SHA-512 and significantly less efficient than more modern designs such as BLAKE3. By contrast, SHA-512 processes 1024-bit message blocks with 64-bit operations, making it more efficient on such architectures. In this thesis, however, SHA-512 was not benchmarked, as the focus was on SHA-256 and the comparison against Keccak-256 and BLAKE3.

Despite these performance limitations, SHA-256 remains a cornerstone in cryptography due to its simplicity, standardization, and lack of practical vulnerabilities. It is often used as a reference point in performance evaluations of newer hash functions.

2.2.2 Keccak-256

Keccak-256 [11] is the 256-bit variant of the Keccak family, which won the NIST SHA-3 competition in 2012 [12]. Unlike SHA-2, Keccak is based on a *sponge construction* that alternates between absorbing input blocks and squeezing output. This design provides strong theoretical guarantees and a high level of security against known cryptanalytic attacks.

Although Keccak-256 is cryptographically very robust, its performance is generally slower than SHA-2 and significantly slower than BLAKE3 in software implementations. However, its adoption is widespread in domains where security guarantees are paramount. A prominent example is Ethereum, where Keccak-256 is used extensively in transaction validation, block hashing, and smart contract execution.

In this thesis, Keccak-256 is included not because of raw speed but because of its relevance in production distributed systems, where it demonstrates the trade-off between cryptographic strength and computational efficiency.

2.2.3 BLAKE3

The BLAKE3 cryptographic hash function [13] is an evolution of the BLAKE2 cryptographic hash function [14], providing higher performance and introducing several additional features:

- Support for hashing, keyed hashing, and key derivation modes.
- No additional space cost for keyed hashing.
- Parallelizable output generation.

BLAKE3 employs a binary tree structure that splits the input into 1024-byte chunks, which are treated as the leaves of the tree. The final chunk may be shorter, but it cannot be empty (unless the entire input is empty). This design enables unlimited parallelism, as each chunk can be compressed independently, allowing efficient use of modern CPUs with SIMD instructions.

BLAKE3 achieves significantly better performance than both SHA-256 and SHA-512 on modern 64-bit architectures. Within the SHA family, SHA-512 generally outperforms SHA-256 on 64-bit machines [15].

BLAKE3's superior performance stems from its fundamentally different design philosophy. Unlike the inherently sequential SHA algorithms, its tree-based parallelism, fewer rounds, more efficient mixing function, and better cache locality enable it to outperform both SHA variants regardless of word size alignment considerations.

It provides a 128-bit security level and a 256-bit output.

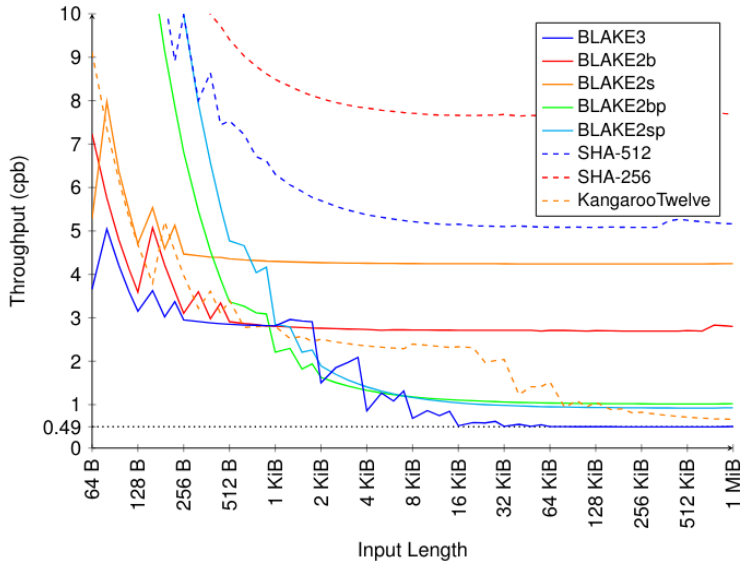

Figure 2.2: Single-threaded throughput of BLAKE3 and other hash functions on an AWS c5.metal instance, measured in cycles per byte (cpb). Lower values indicate fewer CPU cycles needed per byte.

Formally, for a message of length $n > 1024$ bytes, the left subtree covers a number of bytes equal to:

$$2^{10+\lfloor \log_2 \left(\frac{n-1}{1024}\right) \rfloor}$$

The right subtree consists of the remainder. BLAKE3 supports input of any length $0 \leq \ell < 2^{64}$.

This section does not aim to present the full specification of BLAKE3 (its compression function or operational modes) but rather to highlight its practical performance advantages.

Performance

Figures 2.2 and 2.3 show benchmark results from an AWS `c5.metal` instance equipped with dual Intel Xeon Platinum 8275CL (Cascade Lake-SP) processors supporting AVX-512. These results highlight BLAKE3's superior performance compared to other widely used hash functions.

In the context of this thesis, benchmarks were also conducted using the `mt-rs`[1] library to evaluate Merkle tree creation and proof generation under these three different cryptographic hash functions. For each function, three tests were performed

[1] `https://crates.io/crates/mt-rs`

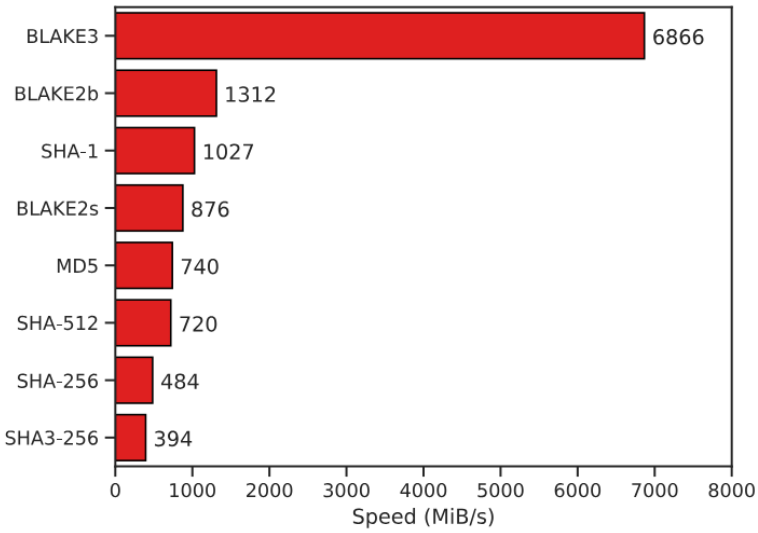

Figure 2.3: Hashing speed comparison of BLAKE3 and other hash functions on an AWS c5.metal instance with a 16 KiB input, using a single thread. Higher values (MiB/s) indicate faster processing.

with node data sizes of 5 MB, 10 MB and 15 MB. In each test, the benchmark measures the time required to construct the authentication path for a leaf, verify the corresponding root from that path, and repeat this process for all 10 nodes of the Merkle tree.

The results, reported in Table 2.1, demonstrate that BLAKE3 consistently outperforms both SHA-256 and Keccak-256 in terms of execution time.

Hash function	**5 MB**	**10 MB**	**15 MB**
SHA-256	89.901 ms	178.42 ms	268.53 ms
Keccak-256	521.49 ms	1.1334 s	1.3438 s
BLAKE3	73.091 ms	154.68 ms	219.79 ms

Table 2.1: Merkle tree benchmarks with 10 nodes per dataset size (5 MB, 10 MB, and 15 MB).

2.3 Consensus Protocols

This section provides background on the consensus protocols that were studied during the internship in order to evaluate how to integrate our Merkle tree file corruption detection system into a distributed setting. Although corruption de-

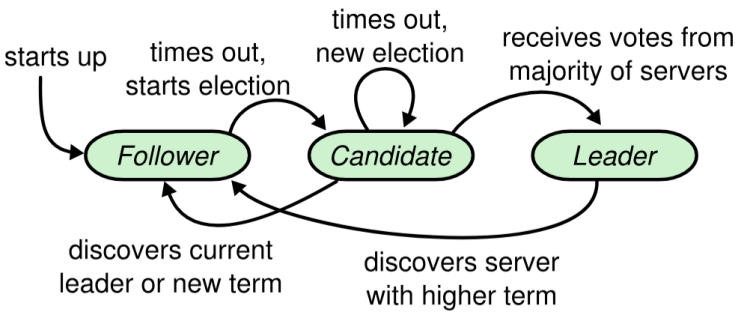

Figure 2.4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

tection could be applied in a centralized environment using simple checksums, in a distributed cluster, a consensus algorithm is needed to coordinate nodes and ensure consistent state.

Several consensus mechanisms were examined, reflecting the long-standing debate between *leader-based* protocols (such as Raft, PBFT, and PB-Raft) and *leaderless* approaches (such as Flutter+Blink). Each design choice has implications for performance, fault tolerance, and implementation complexity.

2.3.1 Raft

Raft [16] is a consensus protocol designed to be understandable and practical, and it has become widely adopted in distributed systems such as `etcd` [17] and TiKV [18]. It tolerates *crash faults*, but not Byzantine behaviour [19].

Each server on a Raft cluster is modelled as a finite state machine with three possible states:

- **Follower:** Passive state that responds to requests from the leader and candidates.
- **Candidate:** Initiates an election when a follower times out without hearing from a leader.
- **Leader:** Elected through majority vote, responsible for handling client requests and replicating logs.

In Figure 2.4, the three server states are illustrated using a finite state machine.

Leader election in Raft is randomized: followers start an election if they do not receive a heartbeat in a randomized timeout window (e.g., 150–300 ms). Candidates send `RequestVote` messages to all nodes, including their last log index and term, to ensure that outdated nodes cannot become leader. If a candidate obtains a majority of votes, it becomes the leader. Otherwise, it retries after another randomized timeout.

Once elected, the leader replicates client requests in the form of log entries using the `AppendEntries` message. Followers acknowledge receipt and once a majority confirms an entry, it is committed and applied to the state machines. This ensures safety (no two servers commit different values at the same log index) and availability (progress can be made as long as a majority of nodes are reachable).

Raft is crash fault-tolerant, but not Byzantine fault-tolerant. However, its simplicity and efficiency make it well-suited for small to medium-sized clusters.

2.3.2 Flutter+Blink

Flutter and Blink [20] are Byzantine fault-tolerant protocols that operate without a leader and without cryptographic signatures. They require at least $5f+1$ servers to tolerate f Byzantine faults.

- **Blink:** Provides binary consensus using *Representative Binary Consensus (RBC)*, where a proposal is considered valid only if at least $f + 1$ correct servers support it. This in contrast to the single correct server required by Binary Consensus.
- **Flutter:** Builds on Blink to provide total-order broadcast, ensuring all servers agree on the same sequence of client messages. It achieves low latency with a best-case of $2\Delta + \epsilon$, where Δ is the network delay and ϵ is negligible.

Flutter uses a betting mechanism where clients attach timestamps (bets) to messages. Servers then order messages according to these bets, ensuring global consistency without requiring a centralized leader.

This makes Flutter+Blink interesting for highly adversarial settings, as they are leaderless, resilient to Byzantine behaviour and even quantum-attack resistant (due to the absence of signatures). However, the $5f + 1$ replication requirement can be costly in practice.

2.3.3 Practical Byzantine Fault Tolerance (PBFT)

Practical Byzantine Fault Tolerance (PBFT), introduced by Castro and Liskov in 1999 [21], was the first protocol to show that Byzantine fault tolerance could be

practical in asynchronous environments. PBFT tolerates up to f Byzantine faults among $3f + 1$ replicas, using cryptographic signatures and message authentication codes (MACs) to prevent spoofing and replay attacks.

The protocol proceeds in three phases:

1. **Pre-prepare:** The leader proposes an order for client requests.
2. **Prepare:** Replicas exchange messages to confirm the proposal and ensure consistent ordering.
3. **Commit:** Replicas agree to execute the request once a quorum of matching prepare messages is observed.

A client considers its request successful once it receives $f + 1$ valid replies. If a timeout occurs, the request is retransmitted.

PBFT guarantees both safety (no two correct replicas decide differently) and liveness (progress is eventually made under partial synchrony). However, the communication overhead is high: the preparation and committing phases require $O(n^2)$ messages, limiting scalability.

Despite being somewhat outdated, PBFT remains foundational and continues to inspire more efficient Byzantine consensus protocols.

2.3.4 PB-Raft: A Byzantine Extension of Raft

Raft is widely used due to its simplicity and efficiency, but it only tolerates crash faults. PBFT tolerates Byzantine faults but at a high communication cost. PB-Raft [22] is a recent proposal that aims to combine the strengths of both.

Key features of PB-Raft include:

- **BLS signatures [23]:** Allow short, aggregable multi-signatures that improve efficiency in log replication.
- **PageRank-inspired leader election:** Nodes are ranked by a probability score. Nodes with higher scores use shorter timeouts, balancing fairness and responsiveness.
- **Semi-synchronous model:** Assumes bounded network delays while tolerating Byzantine behaviour.

Compared to PBFT, PB-Raft reduces message complexity by adopting Raft's two-phase replication approach while retaining Byzantine resilience.

2.3.5 Summary

In summary:

- **Raft** is practical, simple and widely adopted for crash fault tolerance.
- **PBFT** provides strong Byzantine fault tolerance but at high communication cost.
- **Flutter+Blink** achieves leaderless, low-latency Byzantine consensus but requires many servers.
- **PB-Raft** is a hybrid approach that adapts Raft for Byzantine environments.

For the scope of this thesis, Raft was chosen as the consensus algorithm due to its simplicity, maturity and suitability for a small cluster without Byzantine assumptions.

2.4 Reed-Solomon

This section provides background on Reed-Solomon coding to give the reader a clearer picture of Cubbit's infrastructure and the context in which the proposed corruption detection mechanism was tested.

2.4.1 Classical Reed-Solomon codes

Reed-Solomon error correction [24] is one of the most widely used error correction codes, applied in digital communications, storage systems, and network protocols.

A Reed-Solomon code is defined over a finite field F_q, where F denotes the finite field and q is the size of its alphabet (typically a power of two, e.g. $q = 2^8$ for byte-oriented operations).

Classically, a Reed-Solomon code is parameterized by two values: the block length n, representing the total number of symbols in a codeword (data plus redundancy), and the message length k, representing the number of original data symbols, with $k < n \leq q$.

The encoder maps the k data symbols $m_0, m_1, \ldots, m_{k-1}$ to a polynomial of degree at most $k - 1$:

$$P(x) = m_0 + m_1 x + m_2 x^2 + \cdots + m_{k-1} x^{k-1}. \tag{2.3}$$

This polynomial is evaluated at n distinct points $x_1, x_2, \ldots, x_n$ in F_q, producing n encoded symbols. The first k symbols correspond to the original data, while the remaining $n - k$ symbols are redundancy.

The key property of Reed-Solomon codes is that any subset of k symbols from the n encoded symbols is sufficient to reconstruct the original message using polynomial interpolation (e.g. Lagrange interpolation). This allows recovery even if some symbols are lost or corrupted.

2.4.2 Cubbit's adaptation

Cubbit employs Reed-Solomon erasure coding to store files reliably across multiple nodes in its geo-distributed network. Unlike the classical definition, in Cubbit's implementation the total number of shards stored across the network is $n + k$, where n denotes the number of data shards and k the number of redundancy shards.

For example, with three agents, a file could be split into three data shards, distributed one per agent. If the system is configured with $k = 1$, only $n = 2$ shards are required to reconstruct the original file. Thus, even if one agent is offline, the user can still recover the file successfully.

3 Architecture

This chapter describes the architecture of the proposed corruption detection system. First, the Rust library developed specifically for this project is introduced, designed to operate efficiently both at the folder level and on individual data items. Then, the organization of the Merkle tree within the system is discussed, including how the agents (also referred to as *nodes*) exchange information through the Raft cluster. Particular attention is given to scenarios in which some agents are temporarily unavailable (during uploads or during corruption checks) and how the system leverages the Raft log to maintain a consistent view of per-agent root hashes. This ensures that corruption verification can proceed safely and correctly, even when some agents lag behind or remain offline, independently of the Reed-Solomon requirement.

3.1 Merkle Tree Library

As discussed in Section 2.1, the corruption detection system relies fundamentally on the Merkle tree data structure. Despite the widespread use of Merkle trees in the literature and in applications such as blockchains, there are relatively few general-purpose and reusable libraries available online. This is largely because implementations are often developed *ad hoc*, tailored to the needs of a specific infrastructure or application domain.

To overcome this limitation, a dedicated library was developed in Rust[1], named `mt-rs`. The library is distributed under a BSD-3 Licence on Crates.io[2], with its source code publicly available at `https://github.com/boozec/mt`.

Hasher The construction of a Merkle tree begins with the definition of a *hasher*, i.e., the cryptographic hash function applied to the leaves and internal nodes of the tree. This enables direct experimentation with different trade-offs between performance and security. The design allows developers to implement custom

[1] `https://www.rust-lang.org`

[2] `https://crates.io/crates/mt-rs`

hashers by instantiating the trait `Hasher`, which requires only the implementation of the `hash` method. Listing 1 illustrates a simple example of a user-defined hasher.

```rust
use mt_rs::hasher::Hasher;

pub struct FooHasher;

impl Hasher for FooHasher {
    fn hash(&self, input: &[u8]) -> String {
        let sum: u32 = input.iter().map(|&b| b as u32).sum();
        format!("foo_{:x}", sum)
    }
}
```

Listing 1: Example of a custom hasher `FooHasher`, which hashes an input as a string with the prefix "foo_" followed by the sum of the integer values of its bytes, in hexadecimal format.

The library provides three default hashers: `SHA256Hasher`, `Keccak256Hasher`, and `Blake3Hasher`. They correspond to the functions analysed in Section 2.2.

Tree construction A Merkle tree is represented by the structure shown in Listing 2. It can be instantiated either from raw in-memory data using the `new` method or from the contents of files and folders using the `from_paths` method (Listing 3). This dual approach supports both synthetic testing and real-world scenarios, such as integrity verification of storage systems, by avoiding repeated disk reads.

```rust
pub struct MerkleTree {
    /// Leaf nodes at the base of the tree
    /// (may include a duplicate for even pairing).
    leaves: Vec<Node>,
    /// Height of the tree (number of levels including root).
    height: usize,
    /// Root node of the Merkle tree.
    root: Node,
}
```

Listing 2: `MerkleTree` structure definition, where `Node` is an ad-hoc structure that includes additional information and methods.

```rust
impl MerkleTree {
    pub fn new<I, T, H>(hasher: H, data: I) -> Self
    where
        I: IntoIterator<Item = T>,
        T: AsRef<[u8]>,
        H: Hasher + 'static + std::marker::Sync,
    { /* ... */ }

    pub fn from_paths<H>(hasher: H, paths: Vec<String>) -> Self
    where
        H: Hasher + 'static + std::marker::Sync + Clone,
    { /* ... */ }
}
```

Listing 3: Signatures of the `new` and `from_paths` methods. A concrete `Hasher` is always provided when defining a Merkle tree.

Internally, both methods translate the input data into leaf nodes of type `Node` and then invoke the builder function (Listing 4), which assembles the tree level by level. The construction algorithm ensures binary balance by duplicating the last node when the number of nodes is odd.

Parallelization is achieved through the `par_chunks` method of the Rayon crate[3], which splits slices into disjoint chunks and computes parent nodes concurrently.

The tree is organized into *levels*: the leaves at Level 1, the root at the highest level, and internal nodes in between (Figure 3.1). This layered representation makes the structure conceptually simple. The root node is accessible through the `root()` method, and its hash can be retrieved directly. Unlike leaves and the root, internal nodes are not stored explicitly in the `MerkleTree` structure.

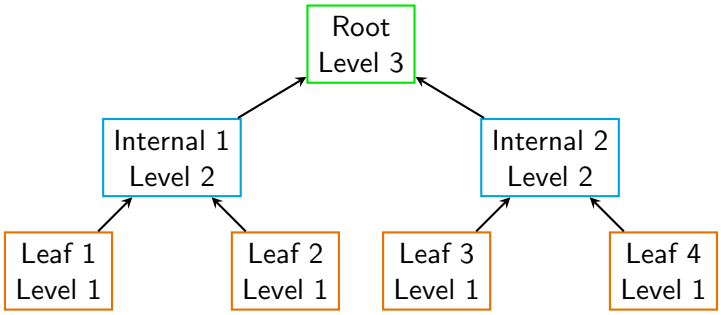

Figure 3.1: An example of a binary Merkle tree with 4 leaves, showing the different levels: leaves (Level 1), internal nodes (Level 2), and the root (Level 3).

[3] `https://crates.io/crates/rayon`

```
impl MerkleTree {
    fn build<H>(hasher: H, mut leaves: Vec<Node>) -> Self
    where
        H: Hasher + 'static + std::marker::Sync,
    {
        let original_leaves = leaves.clone();
        let mut height = 1;

        while leaves.len() > 1 {
            if leaves.len() % 2 != 0 {
                leaves.push(leaves.last().unwrap().clone());
            }

            leaves = leaves
                .par_chunks(2)
                .map(|pair| {
                    let combined = [
                        pair[0].hash().as_bytes(),
                        pair[1].hash().as_bytes()
                    ]
                    .concat();

                    let hash = hasher.hash(&combined);

                    Node::new_internal(hash, pair[0].clone(), pair[1].clone())
                })
                .collect();

            height += 1;
        }

        MerkleTree {
            leaves: original_leaves,
            height,
            root: leaves.into_iter().next().expect("root not found"),
        }
    }

}
```

Listing 4: Build method for constructing the Merkle tree from the leaves upward. The `height` variable tracks the number of levels.

Listing 5 demonstrates printing the Merkle tree root hash. The `hash()` method of each node returns the computed hash as a string.

```rust
let data = &["hello".as_bytes(), "world".as_bytes()];
let tree = MerkleTree::new(Blake3Hasher::new(), data);
println!("Merkle root: {}", tree.root().hash());
```

Listing 5: Snippet of code that prints the Merkle root hash of a tree with two byte strings as leaves.

Proof generation and verification The library also supports Merkle proofs, which enable verification that a given leaf belongs to a specific Merkle tree. Proofs are generated and verified via implementations of the `Proofer` trait (Listing 6). A Merkle proof is expressed as sequences of `ProofNode` elements, which encode the sibling hashes encountered on the path from the leaf to the root. Users may define custom proofers if needed.

```rust
/// Represents a single step in a Merkle proof path.
pub struct ProofNode {
    pub hash: String,
    pub child_type: NodeChildType, // Left or Right
}

/// A Merkle proof containing the path from a leaf to the root.
pub struct MerkleProof {
    pub path: Vec<ProofNode>,
    pub leaf_index: usize,
}

pub trait Proofer {
    /// Generates a Merkle proof for the data at the specified index
    fn generate(&self, index: usize) -> Option<MerkleProof>;

    /// Verifies that a piece of data exists in the tree using a Merkle proof
    fn verify<T>(&self, proof: &MerkleProof, data: T, root_hash: &str) -> bool
    where
        T: AsRef<[u8]>;
}
```

Listing 6: The `Proofer` trait.

In this work, the `DefaultProofer` is used. Its implementation corresponds to proof generation (Algorithm 1) and proof verification (Algorithm 2). Unlike the `MerkleTree` structure, which does not store internal nodes, the `DefaultProofer` retains all levels.

Verification proceeds by iteratively reconstructing the root hash from the leaf and its authentication path, comparing the result with the expected root. The implementation in Rust is reported in Listing 7.

```rust
impl<H> Proofer for DefaultProofer<H>
where
    H: Hasher,
{
    fn generate(&self, index: usize) -> Option<MerkleProof> {
        if index >= self.levels[0].len() { return None; }
        let mut path = Vec::new();
        let mut current_index = index;
        for level in &self.levels[..self.levels.len() - 1] {
            let sibling_index = (current_index ^ 1).min(level.len() - 1);
            let sibling = &level[sibling_index];
            let child_type = if sibling_index < current_index {
                NodeChildType::Left
            } else {
                NodeChildType::Right
            };
            path.push(ProofNode { hash: sibling.hash().to_string(), child_type });
            current_index >>= 1;
        }
        Some(MerkleProof { path, leaf_index: index })
    }

    fn verify<T>(&self, proof: &MerkleProof, data: T, root_hash: &str) -> bool
    where
        T: AsRef<[u8]>,
    {
        let mut current_hash = self.hasher.hash(data.as_ref());
        for proof_node in &proof.path {
            let combined: String = match proof_node.child_type {
                NodeChildType::Left =>
                    format!("{}{}", proof_node.hash, current_hash),
                NodeChildType::Right =>
                    format!("{}{}", current_hash, proof_node.hash),
            };
            current_hash = self.hasher.hash(combined.as_bytes());
        }
        current_hash == root_hash
    }
}
```

Listing 7: Implementation of the `Proofer` trait for `DefaultProofer`. The proof is built by traversing the tree levels and collecting sibling hashes along the path.

Example and Conclusion The complete workflow of the library is illustrated in Listing 8. In this test, a Merkle tree is constructed from the folder *pics*, which contains three files. The program verifies the basic properties of the tree, such as its height, and root hash, before generating a Merkle proof for the first leaf. Finally, the proof is successfully verified against the computed root hash, confirming the correctness of both tree construction and proofing. This example brings together the key components of the `mt-rs` library (hashers, tree building, and proof generation) and demonstrates their integration in practice, concluding the presentation of the Merkle tree library by showing how the system operates end-to-end on real data.

```rust
let hasher = Blake3Hasher::new();
let folders = vec![String::from("pics/")];

let tree =
    MerkleTree::from_paths(hasher.clone(), folders);

assert_eq!(tree.height(), 3);
assert_eq!(
    tree.root().hash(),
    "a08c44656fb3f561619b8747a0d1dabe97126d9ed6e0cafbd7ce08ebe12d55ca"
);

let proofer = DefaultProofer::new(
    hasher.clone(),
    // Recursively hashes the contents of files and directories.
    hash_dir(hasher.clone(), folders),
);

let proof = proofer.generate(0).expect("proof generation failed");

assert!(proofer.verify(
    &proof,
    // Read the content of the first leaf read by the folder pics/
    &fs::read("pics/photo0.png").expect("file not found"),
    "a08c44656fb3f561619b8747a0d1dabe97126d9ed6e0cafbd7ce08ebe12d55ca"
));
```

Listing 8: End-to-end test of the `mt-rs` library: building a Merkle tree from the folder *pics* (with three files), checking its properties, and verifying that a proof for the first leaf matches the expected root hash.

3.2 File Organisation

This section describes how files are structured and managed within the proposed corruption detection system, highlighting the integration of Merkle trees with Cubbit's existing Reed-Solomon-based infrastructure. As discussed in Section 2.4.2, each file is split into $n + k$ shards, with each shard distributed to a different agent (i.e., node).

To uniquely identify each file and its shards, the original filename is converted into a random lowercase hexadecimal string (e.g., `ff4c4b3`). Each shard is then appended with the identifier of the agent storing it (e.g., `ff4c4b3.1` for the shard on Agent 1), facilitating tracking and reconstruction.

When a user downloads a file, the system retrieves the shards from the respective agents and reconstructs the file using the Reed-Solomon algorithm. The reconstructed file is then returned to the user. In reality, the complete flow also involves encryption and decryption steps, but these are not relevant to understanding how files are organised for the purposes of this discussion.

As explained in Section 3.1, the developed Merkle tree library can operate directly on a list of folders. For this reason, the file organisation within the proposed architecture can easily adopt a two-level folder hierarchy. For example, given a file named `ff4c4b3`, the file is stored under the path `ff/4c/4b3`. Another file, such as `ff4c61a`, is stored under the path `ff/4c/61a`, meaning that the folder `ff/4c` contains both files. This hierarchical organisation is illustrated in Figure 3.2.

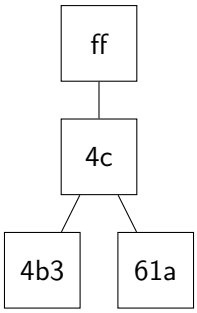

Figure 3.2: Organisation of files under the folder `ff`, represented as a tree structure.

From this figure, the reader can observe that `4c` can be regarded as an internal node with two leaves. For a larger example, Figure 3.3 shows how the same organisation scales when more files are stored in the folder `ff`. At this scale, the overall structure resembles a larger tree with `ff` as the root.

It is important to clarify, however, that the diagrams so far represent *directory trees*, not Merkle trees. In a Merkle tree, internal nodes are not simply folders but cryptographic hashes computed from their children (the leaves or subtrees).

For this reason, the second-level folders in the figure cannot directly be considered internal nodes of a single Merkle tree. Instead, the entire filesystem should be viewed as a *Merkle tree forest*: each second-level folder forms an independent Merkle tree, while each top-level folder can itself be treated as a Merkle tree whose leaves are the root hashes of its subfolders.

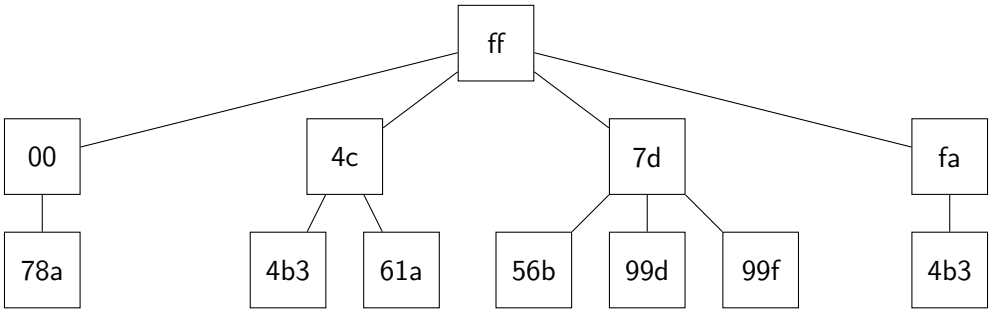

Figure 3.3: Extended example of file organisation under `ff`, represented as a tree structure with multiple files.

If the leaves of a Merkle tree correspond to individual file blocks or shards, then a single root hash can represent the entire `ff` folder. Alternatively, smaller Merkle trees can be built independently for each subfolder. For instance, in Figure 3.3, one could compute four Merkle root hashes for the second-level folders and one root hash for the top-level folder `ff`.

This arrangement, referred to here as a *Merkle tree forest*, allows up to 256 top-level folders. Since only lowercase hexadecimal strings are used, ensuring compatibility with both case-sensitive and case-insensitive filesystems, the range spans from `00/` to `ff/`. Each top-level folder can contain up to 256 second-level folders, as the namespace is determined by four hexadecimal characters. Dividing the data into smaller trees is crucial for scalable and efficient verification, as will be detailed in the following section.

3.2.1 Storing different Merkle trees for each agent

Because each file is split into $n + k$ shards and distributed across different nodes, each agent maintains its own local file organisation. As a result, there are effectively $n + k$ distinct filesystems, one for each agent. It should be noted that the sets of files stored by different agents may not be identical, since some agents may be offline during an upload. Nevertheless, the file is still successfully stored thanks to the Reed-Solomon requirements.

To better understand this, we can consider the entire filesystem of an agent as a tree structured as follows:

- **Root.** The global root, representing all files stored by the agent. In practice, this is rarely used because computing a Merkle tree for this element would require hashing the entire filesystem at once.
- **Second level.** The top-level folders (e.g., `ff`), each of which corresponds to the root of a Merkle tree for that folder.
- **Third level.** The second-level (or *sub*) folders (e.g., `ff/4c`), which are smaller Merkle trees containing only a subset of files.
- **Leaves.** The actual file data blocks.

The global root of the entire filesystem is avoided because it is too expensive to compute, especially across multiple agents that may be offline at any given time. At the other extreme, while each file could theoretically be validated individually, storing and checking a hash for every file would reduce the system to a simple checksum-based corruption detection scheme, which lacks scalability.

Instead, the system leverages intermediate Merkle trees at the *folder levels*. These allow integrity verification to be performed at different granularities: either locally within a subfolder or globally within a top-level folder, without the overhead of recalculating the root hash for the entire agent's filesystem.

Figure 3.4 illustrates an example of this organisation for a Reed-Solomon configuration with $n = 2, k = 1$. In this example, the top-level and second-level folders are highlighted with lighter and darker colours, respectively.

For each top-level folder, every agent computes and stores the Merkle root hash obtained from a Merkle tree whose leaves correspond to the *terminal files*. The same procedure is applied to second-level folders. For example, in Figure 3.4, Agent 1 computes and stores the Merkle root hashes of `fe`, `ff`, `fe/2d`, `ff/4c`, and `ff/6d`. Agents 2 and 3 follow the same procedure, maintaining the root hashes corresponding to their respective local filesystems.

A key property of these Merkle root hashes is that they can themselves be used as input for higher-level Merkle trees. The system computes aggregated Merkle trees from the folder roots and stores only the resulting root hashes in *string format*. This recursive organisation is illustrated in Figures 3.5 and 3.6.

The distinction between top-level and second-level folder roots becomes largely irrelevant in storage terms: both are represented uniformly in a map, where the key is the folder identifier (two characters for a top-level folder, five for a second-level folder) and the value is the corresponding root hash. This raises a couple of questions: which component is responsible for storing and maintaining this map, and why does the system use two levels of folders instead of just one?

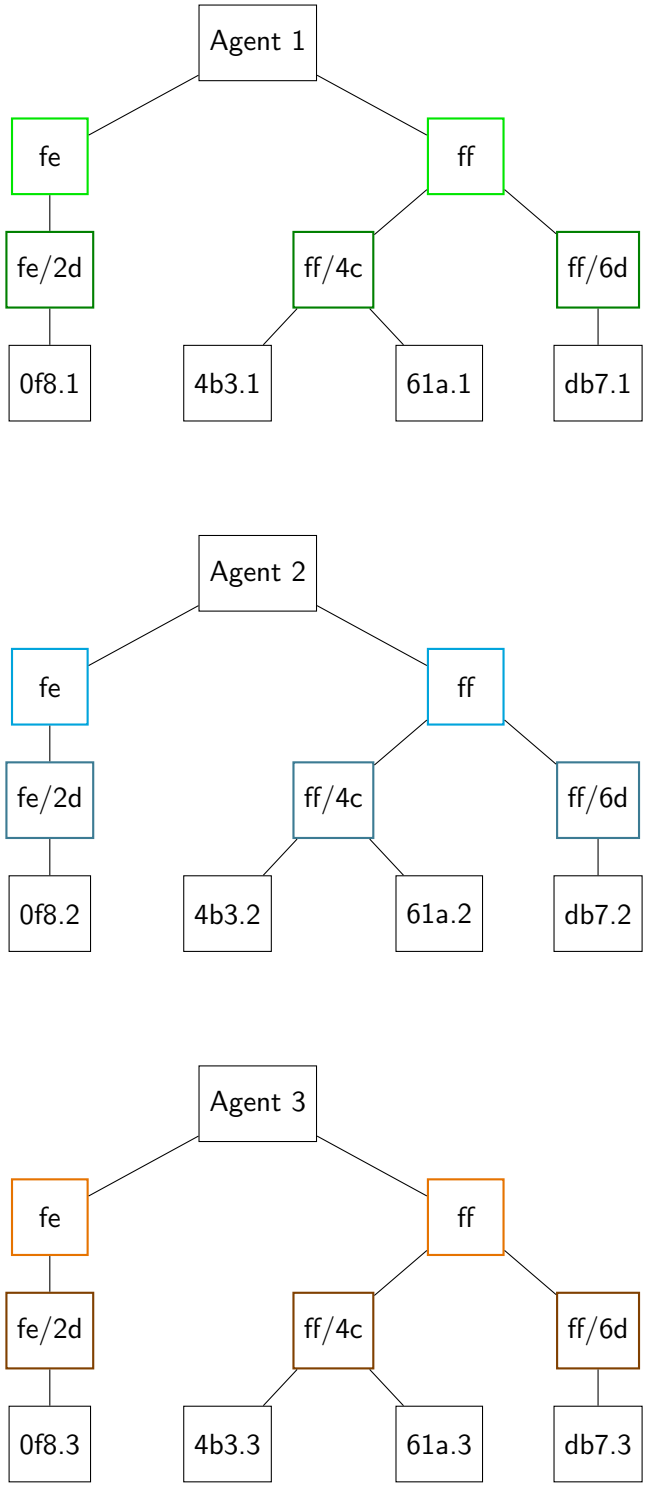

Figure 3.4: Example of filesystems for `Agent 1`, `Agent 2`, and `Agent 3`.

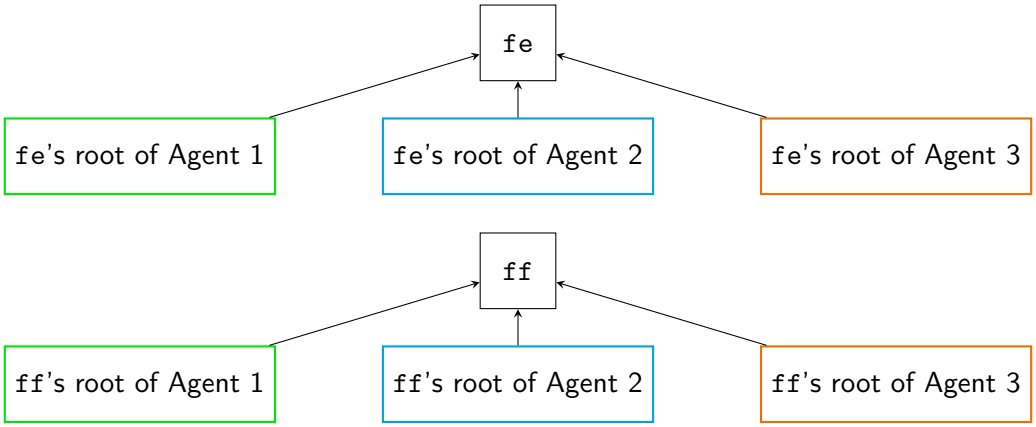

Figure 3.5: Merkle tree constructed from the root hashes of top-level folders. Each folder root acts as a leaf in this Merkle tree.

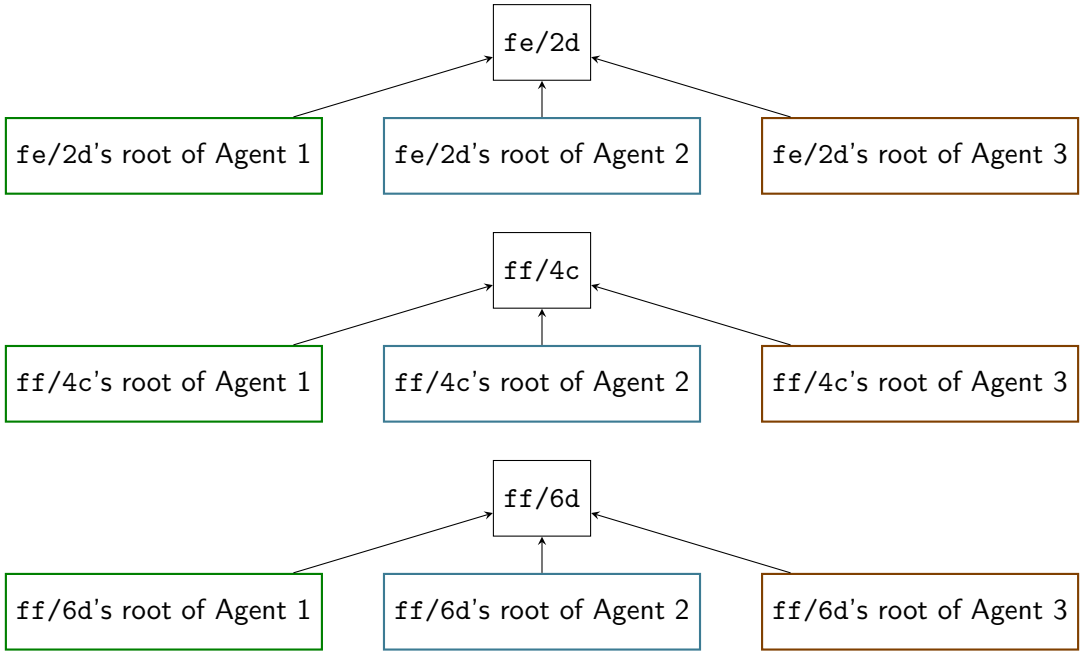

Figure 3.6: Merkle tree constructed from the root hashes of second-level folders. Each folder root acts as a leaf in this Merkle tree.

3.3 A Raft Cluster for File Uploads

In Section 2.3.1, the Raft consensus algorithm was introduced. This section explains its role within the proposed file corruption detection system, focusing on how Raft enables global agreement among agents on the Merkle root hashes associated with different folders.

As discussed in the previous section, there is no practical distinction between top-level and second-level folder roots: both are stored uniformly. The key requirement is that each agent must be aware of the Merkle root hash of every folder *globally*, not just for its own local filesystem. This consistency is ensured through a Raft cluster, in which all agents act as nodes. Each node maintains a replicated log that functions as a command list of operations for a distributed key/value store, where folder identifiers serve as keys and the corresponding Merkle root hashes serve as values. Ideally, every correct node maintains an identical log, guaranteeing global agreement.

In practice, the Raft log is a sequence of messages broadcast by the leader to all nodes. Since the Raft cluster includes all agents, the terms *node* and *agent* can be used interchangeably. If an agent is online, it is part of the Raft cluster as well.

The log allows each agent to reconstruct "two levels" of a map, corresponding to the two levels of folder roots. Whenever a node restarts or rejoins the cluster, the log is replayed to restore state consistency. For example, in the scenarios illustrated in Figures 3.5 and 3.6, each agent would maintain a single map such as:

```
roots = {
    # (1) Roots of top-level folders:
    "fe":   "root hash of fe",
    "ff":   "root hash of ff",

    # (2) Roots of second-level folders:
    "fe/2d":"root hash of fe/2d",
    "ff/4c":"root hash of ff/4c",
    "ff/6d":"root hash of ff/6d"
}
```

Listing 9: Example of a map of Merkle root hashes. Whether the key represents a top-level or second-level folder is irrelevant for now.

Only the Raft leader is allowed to append new messages to the log. In other words, one designated agent is responsible for broadcasting updates so that all other agents converge on the same global map.

Figure 3.7 shows the sequence of events when a user uploads a file via the

gateway (the service exposed for uploads and downloads). The gateway (GW) communicates directly with all agents, and the agents form a Raft cluster. Each action triggers local computations, and the figure illustrates how updates propagate through the system:

(1) The gateway receives a file, applies Reed-Solomon encoding, and splits it into $n + k$ shards. It generates a random salt (e.g., two bytes) and appends it to the filename (e.g., `photo.png24`). The salt prevents filename collisions and ensures randomized placement of the shards. Each shard is then sent to a different agent.

(2) Each agent uses a deterministic algorithm to transform the filename+salt into a unique hexadecimal identifier (e.g., `ff4c4b3`). This ensures the file is placed in a folder not previously used or marked as corrupted. The file is then stored under a two-level folder hierarchy (e.g., `ff/4c/4b3`). Each agent appends its identifier (i.e., `.1`, `.2`, `.3`) and saves the shard locally (e.g., `ff/4c/4b3.1` for Agent 1).

(3) Each agent computes the updated Merkle root hashes for the affected folders (e.g., `ff` and `ff/4c`) and sends them to the leader. Since any file modification propagates to the corresponding folder root, this step ensures consistency even when the top-level or second-level folders already contain files.

(4) The leader aggregates the roots received from all agents and computes global Merkle roots for each folder as illustrated in Figures 3.5 and 3.6 (e.g., one for `ff`, one for `ff/4c`). The leader appends these values to the Raft log, which is replicated across the cluster.

As a result, all agents maintain an up-to-date map from folder names to their latest root hashes, as illustrated in Listing 9.

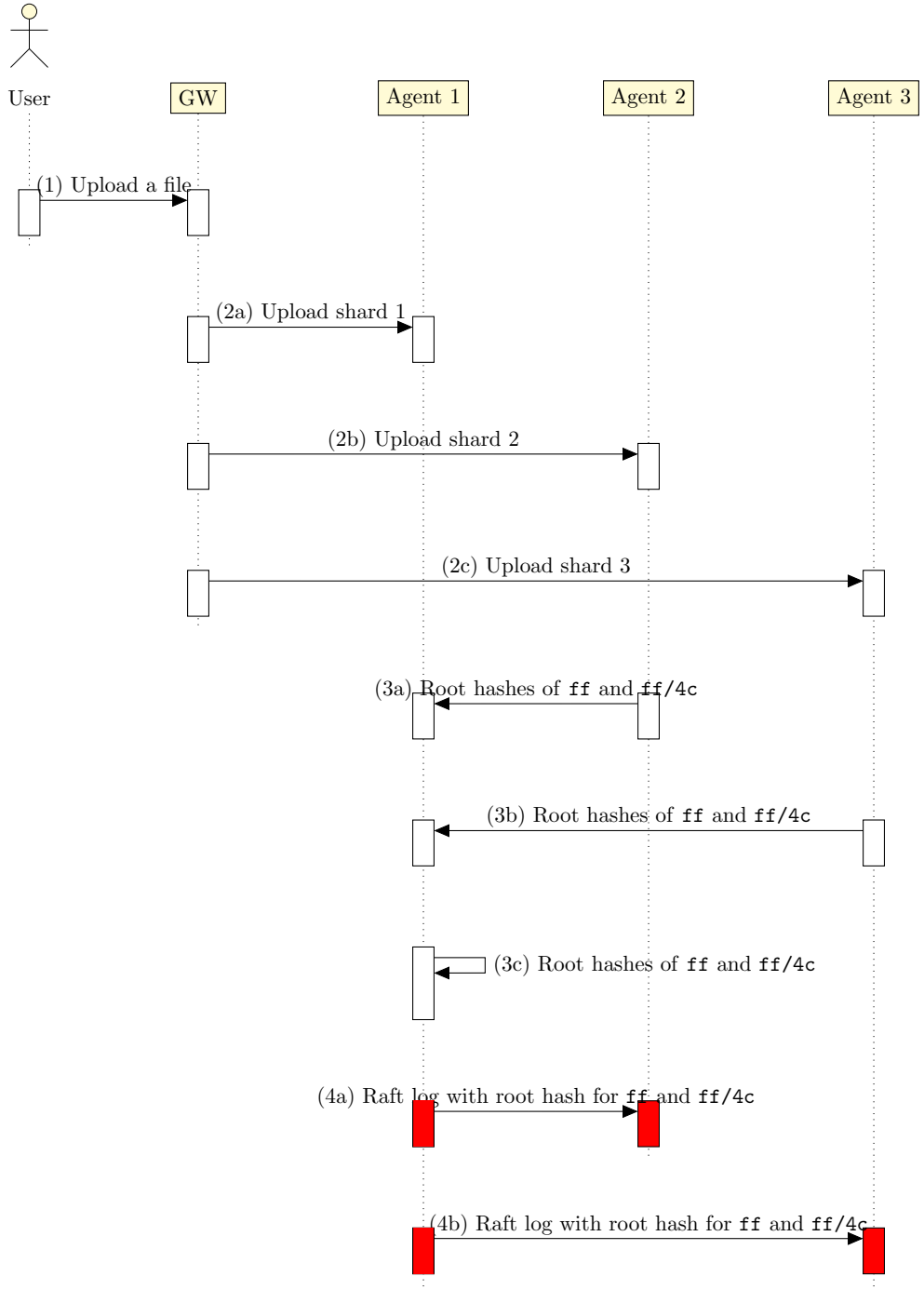

Figure 3.7: Sequence diagram for uploading a file. In this example, Agent 1 is the Raft leader and the uploaded file has a path starting with `ff/4c`.

3.3.1 Corruption Check

While Figure 3.7 illustrates how uploads are handled, an equally important question remains: how can the system detect corrupted files? This is the purpose of the *Corruption Check* process.

The process, executed at regular intervals for all top-level folders, is initiated by the leader. This responsibility lies with the leader because it is the only agent authorised to append new entries to the log. If corruption is detected, the leader records the result in the Raft log, thereby propagating the information consistently to all other agents.

Although earlier sections emphasized that top-level and second-level folders are stored uniformly, a practical distinction emerges during corruption checks. If a top-level folder is verified as intact, there is no need to check its subfolders. This follows from the Merkle tree property: any modification to a leaf propagates upward, changing the root. Thus, an unchanged top-level root implies that all second-level folders beneath it are also intact. Conversely, if a top-level folder's root does not match, the leader must recursively verify all second-level folders belonging to it.

In the best case, only the top-level folders need to be examined (a maximum of 256 keys, 16^2). In the worst case, when every top-level folder fails verification, all second-level folders must also be checked (up to 65536 keys, 16^4). This scenario corresponds to widespread corruption, where each top-level folder contains at least one corrupted file.

Figure 3.8 illustrates the process for a top-level folder:

(1) The leader queries $n + k$ agents for the Merkle root hash of a given top-level folder (e.g., `ff`). It then applies the Merkle proof verification procedure, comparing the collected root hashes against the previously stored root (e.g. `roots["ff"]`).

(2) The leader appends the result of this verification to the Raft log, broadcasting whether the folder is corrupted. If the folder is marked as corrupted, the process is recursively repeated for all of its second-level subfolders (e.g. `ff/4c` and `ff/6d`).

As a result, all agents share a consistent view of which folders are marked as corrupted. A top-level folder is corrupted if at least one of its subfolders is corrupted.

3.3.2 Corruption Check for Partial Uploads

The mechanisms described so far assumed that all agents are continuously online. In practice, temporary unavailability of agents is unavoidable. This subsection

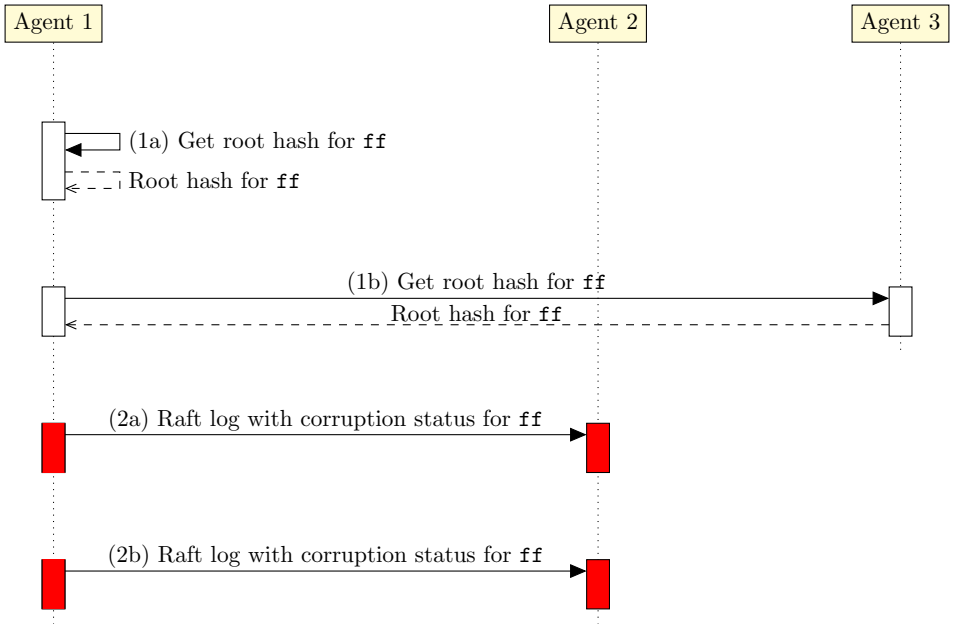

Figure 3.8: Sequence diagram illustrating the Corruption Check process for the top-level folder `ff`. Agent 1 is the Raft leader.

extends the model by considering two cases: (i) when an agent is offline during a file upload, and (ii) when one or more agents are offline during the corruption check itself.

To mitigate these cases, the data structures introduced in Listing 9 are extended. Instead of storing only a single Merkle root hash per folder (both top-level and second-level), the system also records Merkle root hashes for every agent. This results in two complementary maps:

- **roots**: a map from folder identifiers to global root hashes. Each value is the aggregated Merkle root computed over all agents' contributions for that folder.

- **agent_roots**: a map from folder identifiers to the set of per-agent root hashes. Each value explicitly tracks the current Merkle root hash for the corresponding folder, stored in an array indexed from 1 to $n + k$.

If an agent was offline during an upload, its entry in the array remains empty for a new entry or unedited for an older one. Thanks to the Raft log, these per-agent arrays are consistently replicated across all agents. This ensures that even if a leader crashes and a new leader is elected, the corruption check phase remains unaffected.

An example is shown in Listing 10, where Agent 3 was offline during certain uploads. In this case, some entries of the `roots` map are computed using only two leaves, since the third leaf in the corresponding `agent_roots` entries is `nil`.

```
roots = {
    "fe":   "root hash of fe",
    "ff":   "root hash of ff",
    "fe/2d":"root hash of fe/2d",
    "ff/4c":"root hash of ff/4c",
    "ff/6d":"root hash of ff/6d"
}

agent_roots = {
    "fe":[
        "Agent 1 root hash of fe",
        "Agent 2 root hash of fe",
        "Agent 3 root hash of fe"
    ],
    "ff":["Agent 1 root hash of ff",   "Agent 2 root hash of ff",   nil],
    "fe/2d":[
        "Agent 1 root hash of fe/2d",
        "Agent 2 root hash of fe/2d",
        "Agent 3 root hash of fe/2d"
    ],
    "ff/4c":["Agent 1 root hash of ff/4c","Agent 2 root hash of ff/4c",nil],
    "ff/6d":["Agent 1 root hash of ff/6d","Agent 2 root hash of ff/6d",nil]
}
```

Listing 10: Example of folder root hashes with $n = 2$, $k = 1$. Agent 3 was offline during the upload of files in `ff` folder.

It is important to note that root hashes are stored at the *folder* level, not per file. As a result, `agent_roots[<some_folder>]` may contain entries that are not perfectly aligned across agents. For example, Agents 1 and 2 may have already uploaded ten files into a folder, while Agent 3, having been offline, may only have one file in the same folder.

This apparent asymmetry does not compromise correctness. The reason is that `roots[<some_folder>]` is computed as a Merkle root over the entire array `agent_roots[<some_folder>]`. Even if one agent's contribution lags behind, the global root still represents a consistent snapshot of the folder state across all agents. During verification, the Merkle proof ensures that each agent's current hash (or stored historical hash, if offline) matches the position it contributed to in the global root.

Agent offline during upload If an agent is offline during an upload, the system can still guarantee correctness as long as the Reed-Solomon requirement is satisfied. In an $n + k$ configuration, at least n agents must be online, since n shards are sufficient to reconstruct the file on download.

If an agent was offline during an upload, its entry in the per-agent array remains empty (or unedited). Thanks to the Raft log, these arrays are consistently replicated across all agents. Because the system does not modify the entry corresponding to the offline agent, the Merkle root computed in `roots` correctly reflects the full folder state, including the filesystem status of the offline agent. Therefore, the global Merkle root remains consistent even if some agents are temporarily unavailable.

When the offline agent later rejoins, it synchronizes its metadata by replaying the Raft log. However, it does not automatically reconstruct the shards it missed during downtime. During a corruption check, the leader queries the folder root hashes from all agents. For an agent that was previously offline, the stored value in `agent_roots` is exactly what the leader expects. Thus, the check does not fail: the Merkle root returned by the previously offline agent is consistent with the snapshot that the leader uses for verification, assuming no corruption has occurred.

This reasoning naturally generalizes to the case where up to k agents are offline at the same time: as long as at least n agents remain available, the system continues to satisfy the Reed-Solomon requirement.

Agents offline during corruption check A more challenging scenario arises when some agents are offline during the corruption check itself. The set of agents currently online may differ from those that participated in the original upload. If the leader cannot collect the current Merkle root for a folder from all agents (even when some entries in `agent_roots` are `nil`) the corruption check for that folder must rely on the available data. Consequently, the check process is adapted to handle offline agents gracefully.

During a corruption check for a folder, the leader iterates over all $n+k$ positions:

- If agent i is online, it requests the current root hash from that agent.
- If agent i is offline, it retrieves the saved value from the corresponding slot in `agent_roots`.

Once all $n + k$ values are collected, the leader performs the Merkle proof verification against the global root hash stored in `roots`. This process is illustrated in Figure 3.9.

Returning to the example: suppose Agent 2 goes offline while Agent 3 comes online. During the corruption check of folder `ff`, the system uses the stored value of Agent 2's root from `agent_roots["ff"][2]`, while requesting a fresh root from

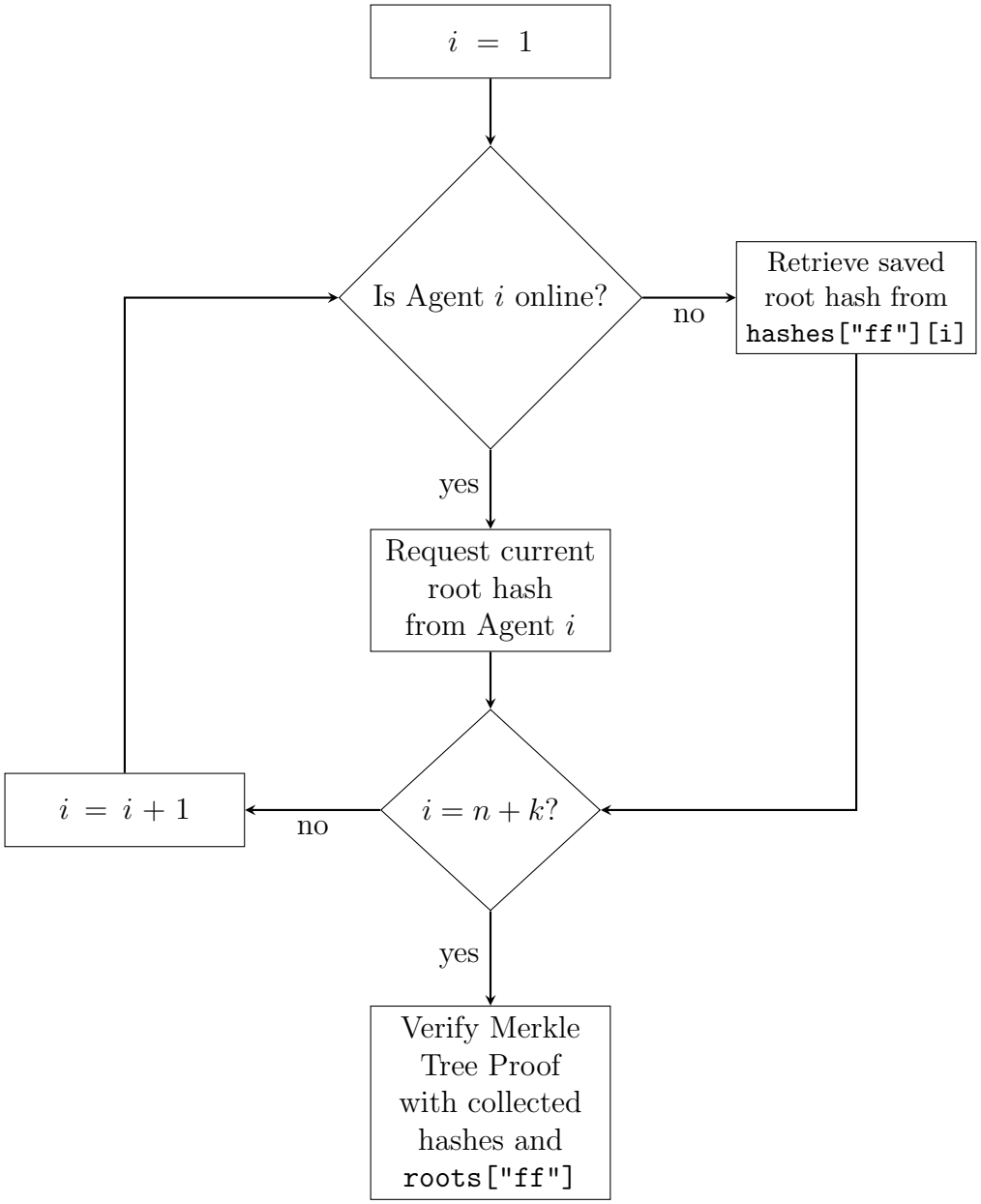

Figure 3.9: Corruption check process for the top-level folder `ff`, considering some agents may be offline.

Agent 3. If the Merkle proof verification succeeds, this guarantees that all currently online agents are consistent and uncorrupted. In this case, the status of offline agents is irrelevant:

- If the proof holds, online agents have uncorrupted data.
- If the proof fails, at least one online agent is corrupted, and the outcome is marked as a corruption regardless of the offline agents.

This highlights an important distinction: the Corruption Check 3.3.1 phase itself does not rely on the Reed-Solomon requirement of having n correct shards. Instead, it depends only on the fact that the Raft log ensures a consistent view of the per-agent root hashes across all agents. Offline agents may lag behind in state (or even remain offline for the entire process) without affecting the verification. In fact, it would be possible to perform the corruption check even if more than k agents are offline. The correctness of the two cases considered (Agents offline during upload and Agents offline during the corruption check) is ensured because the first case already satisfies the Reed-Solomon reconstruction requirement, while the verification itself relies solely on the consistency guaranteed by Raft. Therefore, the corruption check can safely verify the folder state based on the available information, independently of the offline agents.

3.3.3 Recovery of Missing Shards

In the previous subsection, the reader learned what happens when an agent is offline during a file upload. This subsection discusses how missing shards are reconstructed when a previously offline agent comes back online.

The sequence diagram illustrated in Figure 3.7 is simplified: every time a new shard is uploaded, the agent sends an acknowledgment to the leader. This allows the leader to maintain an up-to-date view of the cluster, tracking which agents are online and which have stored a shard for a given file, avoiding active pings or status checks. The acknowledgment is asynchronous, so the leader does not block waiting for responses. Meanwhile, the user receives confirmation through the gateway (GW), which handles shard uploads synchronously. In other words, shard storage and acknowledgment to the leader are independent processes.

Thanks to Raft, the leader keeps a log of all received acknowledgments. Recording this information in the Raft log ensures that every node in the cluster knows which agents were offline during the upload of a file.

During the download process, some previously online agents might now be offline, and vice versa. Even if the set of online agents at upload time satisfied the Reed-Solomon requirement, there is no guarantee that the currently online agents

hold enough shards (at least n) to reconstruct the file. When this happens, a recovery phase is necessary to restore the missing shards to ensure reconstruction.

Similar to the Corruption Check phase, a Recovery phase can be triggered to deliver missing shards to previously offline agents. The leader coordinates this process. To maintain consistency, the receiving agent recalculates the Merkle roots for both the top-level and second-level folders affected by the newly received shard. It then sends the updated roots back to the leader, which updates the corresponding i-th entries in `agent_roots` and recomputes the global `roots` for the folders. Once the missing shard is successfully restored, the leader records the update in the Raft log to notify all nodes that the shard has been sent and the new values for the relevant `roots` and `agent_roots` maps have been established.

Retaking the previous example shown in Listing 10, after a full recovery all entries in `agent_roots` are filled, as illustrated in Listing 11.

```
roots = {
    "fe":   "new root hash of fe",
    "ff":   "new root hash of ff",
    "fe/2d":"new root hash of fe/2d",
    "ff/4c":"new root hash of ff/4c",
    "ff/6d":"new root hash of ff/6d"
}

agent_roots = {
    "fe":   [
        "Agent 1 root hash of fe",
        "Agent 2 root hash of fe",
        "Agent 3 root hash of fe"
    ],
    # [...]
    "ff/6d":   [
        "Agent 1 root hash of ff/6d",
        "Agent 2 root hash of ff/6d",
        "Agent 3 root hash of ff/6d"
    ],
}
```

Listing 11: Example of folder root hashes with $n = 2$, $k = 1$ after every agent received all the shards.

This chapter described how the Merkle tree solution was designed and integrated with a Raft cluster, explaining step by step how the main issues were addressed. The next chapter focuses on the implementation, presenting the developed solution and evaluating it through system-wide testing.

4 Implementation and Tests

This chapter provides a practical overview of the proposed file corruption detection system by presenting a real implementation developed specifically for this work. The implementation consists of a standalone binary application built around the `mt-rs` library, employing the BLAKE3 cryptographic hash function, the Raft consensus algorithm, and the Reed-Solomon codes.

Rather than including extensive code listings, which would not significantly contribute to the goals of this thesis, the discussion highlights selected code snippets that illustrate how the individual components operate together in a coherent system. Finally, the chapter presents a series of benchmarks designed to evaluate the behaviour of the implementation under various scenarios, such as node failures, varying numbers of agents, and datasets of different sizes and file counts.

4.1 Files in a Raft Cluster

As discussed in Section 3.3, the system is built upon a Raft cluster integrated with a storage mechanism based on Reed-Solomon codes. To study this integration, a Go[1] application was implemented, reconstructing a Cubbit-like infrastructure by layering a Raft cluster on top of Reed–Solomon redundancy. Go was chosen for its gentle learning curve and widespread adoption, including by companies such as Cubbit. Following the common project layout recommended for Go[25] all the developed services are organized within a single module. This structure avoids redundancy, such as duplicating the service message definitions described later. Each service is maintained separately as a distinct binary under the `cmd/` folder, in accordance with Go best practices.

To investigate communication patterns between the system components, two service APIs were implemented: a REST API, widely used for web services, and gRPC[2], developed by Google and based on Protobuf, which allows precise definition of the message types transmitted over the network.

[1] `https://go.dev/`
[2] `https://grpc.io/`

```proto
service Agent {
    rpc SendShard(ShardRequest) returns (ShardResponse) {}
    rpc GetShard(ShardGetRequest) returns (ShardGetResponse) {}
    rpc AckShard(ShardAckRequest) returns (ShardAckResponse) {}
    rpc GetRootHash(RootHashRequest) returns (RootHashResponse) {}
    rpc JoinRaft(JoinRequest) returns (JoinResponse) {}
}

message ShardRequest {
    string filename = 1;
    int64 index = 2;
    bytes data = 3;
}

message ShardGetRequest {
    string filename = 1;
    int64 index = 2;
}

message ShardAckRequest {
    string filename = 1;
    int64 index = 2;
    bytes roots = 3;
}

message ShardResponse {
    string filename = 1;
    bytes salt = 2;
}

message ShardGetResponse { bytes data = 1; }

message ShardAckResponse { bool status = 1; }

message RootHashRequest { bytes folder = 1; }

message RootHashResponse { bytes hash = 1; }

// ...
```

Listing 12: Protobuf definitions for the `Agent` service, used for communication between Gateways and Agents as well as among Agents themselves.

Listing 12 illustrates the Protobuf definitions used by the agents, including messages for both gateway-agent and agent-agent interactions.

The particular feature of gRPC is that the Protobuf file can be directly com-

```
protoc --go_out=. \
        --go_opt=paths=source_relative \
        --go-grpc_out=. \
        --go-grpc_opt=paths=source_relative \
        <path>.proto
```

Listing 13: Protobuf compiler command that generates Go code from the service definition located at `<path>.proto`.

piled into Go code using a simple command line, as shown in Listing 13.

This command generates two Go files:

- `<path>.pb.go`, which contains the standard Protobuf message definitions and serialization code.
- `<path>_grpc.pb.go`, which contains the gRPC client and server stubs necessary to implement the service endpoints in Go. The methods defined in Listing 12 are represented as an interface in this file, and they must be implemented to enable communication between a Gateway and the Agent service, or among Agents themselves.

These generated files form the foundation for both Gateway-to-Agent and inter-agent communication in the system.

4.1.1 Gateway service

The Gateway serves as the interface available to the user, as illustrated in the step 1 on Figure 3.7. It is exposed via a REST API and manages file uploads and downloads by splitting files into $n + k$ shards and distributing them across the agents. The REST interface is intentionally minimal in this prototype, offering only two endpoints: one for uploading and one for downloading files. The upload endpoint requires the local file path and the desired filename, while the download endpoint requires only the filename.

4.1.2 Agent service

Each agent is responsible for storing its assigned shards locally and participates as a node in the Raft cluster. The total number of agents corresponds to the Reed–Solomon configuration $(n + k)$. Communication between the Gateway and agents is implemented using gRPC, which allows efficient binary data transfer and

a rich set of commands. Inter-agent communication also uses gRPC, supporting operations such as sending file acknowledgments, managing cluster membership requests, and retrieving the current Merkle root hash for a given folder.

Bibliography

[1] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[2] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.

[3] Ivan Bjerre Damgård. Collision free hash functions and public key signature schemes. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 203–216. Springer, 1987.

[4] Merkle tree in bitcoin. `https://bitcoinwiki.org/wiki/merkle-tree`. Accessed: 2025-09-02.

[5] Patricia merkle trie in ethereum. `https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/`. Accessed: 2025-09-02.

[6] Merkle directed acyclic graphs in ipfs. `https://docs.ipfs.tech/concepts/merkle-dag/`. Accessed: 2025-09-02.

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[8] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. Xmss-a practical forward secure signature scheme based on minimal security assumptions. In *International Workshop on Post-Quantum Cryptography*, pages 117–129. Springer, 2011.

[9] Johannes Buchmann, Erik Dahmen, and Michael Schneider. Merkle tree traversal revisited. In *International Workshop on Post-Quantum Cryptography*, pages 63–78. Springer, 2008.

[10] Wouter Penard and Tim Van Werkhoven. On the secure hash algorithm family. *Cryptography in context*, pages 1–18, 2008.

[11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 313–314. Springer, 2013.

[12] Nist selects winner of secure hash algorithm (sha-3) competition. `https://www.nist.gov/news-events/news/2012/10/nist-selects-winner-secure-hash-algorithm-sha-3-competition`. Accessed: 2025-09-07.

[13] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn. Blake3: one function, fast everywhere. *url: https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf*, 2021.

[14] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. Blake2: simpler, smaller, fast as md5. In *International Conference on Applied Cryptography and Network Security*, pages 119–135. Springer, 2013.

[15] Shay Gueron, Simon Johnson, and Jesse Walker. Sha-512/256. In *2011 Eighth International Conference on Information Technology: New Generations*, pages 354–358. IEEE, 2011.

[16] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. *url: https://raft.github.io/raft.pdf*, 2014.

[17] etcd performance. `https://etcd.io/docs/v3.6/op-guide/performance/`. Accessed: 2025-09-03.

[18] Raft use in tikv. `https://tikv.org/deep-dive/consensus-algorithm/raft/`. Accessed: 2025-09-03.

[19] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[20] Matteo Monti, Martina Camaioni, and Pierre-Louis Roman. Fast leaderless byzantine total order broadcast. *arXiv preprint arXiv:2412.14061*, 2024.

[21] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[22] Haoran Shi, Zehua Chen, Yongqiang Cheng, Xiaofeng Liu, and Qianqian Wang. Pb-raft: A byzantine fault tolerance consensus algorithm based on weighted pagerank and bls threshold signature. *Peer-to-Peer Networking and Applications*, 18(1):26, 2025.

[23] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[24] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[25] Organizing a go module. `https://go.dev/doc/modules/layout`. Accessed: 2025-10-01.

List of Figures

List of Tables