

MEENTO SHARE

A Federated, Content-Addressable Distributed File Store with Proof-of-Storage Incentives

Technical Overview & Architecture Paper

2025

meento.lovable.app | sourceforge.net/projects/meento | github.com/rhizomai/meentoshare

ABSTRACT

Meento Share is a single-file, HTTP-native, federated file replication system in which any web server can become a fully autonomous storage node. Files are identified by their SHA-256 content hash, replicated automatically to a configurable set of peer nodes on upload, and verified through a dual-confirmation protocol. An account system tracks which servers hold each user's files and awards a balance point for each newly confirmed, deduplicated storage location — forming the foundation of a lightweight proof-of-storage incentive mechanism. This paper describes the system architecture, the replication and verification protocol, the server deduplication strategy, the current incentive model, and a roadmap for evolving the incentive layer into a modular, cryptographically secure, and fully decentralised proof-of-storage scheme.

Keywords: distributed storage, content-addressable storage, federated systems, proof of storage, file replication, peer-to-peer, HTTP overlay network

1 INTRODUCTION

The dominant model for file hosting centralises storage behind a handful of large providers. This concentration creates single points of failure, gives providers unilateral control over data availability, and forces users to trust that a service will remain online and honest. Decentralised alternatives such as IPFS and the Dat protocol address these concerns but introduce significant operational complexity: dedicated daemon processes, non-standard transports, and steep configuration curves that place them out of reach for many self-hosters.

Meento Share takes a different position: a storage node should be deployable by anyone who can host a PHP file on a web server, with zero additional software dependencies. The entire system — account management, file storage, inter-node replication, verification, and incentive accounting — lives in a single *index_full.php* file and communicates exclusively over standard HTTP/HTTPS POST requests.

This paper is organised as follows. Section 2 surveys related work. Section 3 describes the system architecture. Section 4 covers the replication and verification protocol. Section 5 explains the server deduplication strategy. Section 6 details the current proof-of-storage incentive mechanism and Section 7 proposes a roadmap for hardening it into a modular, cryptographically sound, and fully decentralised scheme. Section 8 discusses security considerations, and Section 9 concludes.

2 RELATED WORK

Content-addressable storage is a well-studied paradigm. Venti [1] introduced the idea of immutable, hash-identified blocks for archival storage. Git [2] popularised content addressing at the file-tree level. IPFS [3] extended these ideas to a global distributed network with DHT-based peer discovery and merkle-DAG data structures.

Proof-of-storage schemes have been studied extensively in the context of cloud auditing. Ateniese et al. introduced Provable Data Possession (PDP) [4], allowing a client to verify that a server holds a file without downloading it. Juels and Kaliski proposed Proofs of Retrievability (PoR) [5], which additionally verify that the file can be reconstructed. Filecoin [6] combines proof-of-replication and proof-of-spacetime in a blockchain-based marketplace.

Meento Share occupies a pragmatic position in this design space: it sacrifices cryptographic proof strength in favour of near-zero deployment friction, while providing a clear upgrade path toward stronger guarantees. Its federation model is closest to ActivityPub [7] — nodes interoperate via a shared HTTP protocol without a central registry — though the domain is file storage rather than social messaging.

Ref.	Work	Relevance
[1]	Quinlan & Dorward — Venti (2002)	Content-addressable archival blocks
[2]	Torvalds — Git (2005)	Hash-identified immutable objects
[3]	Benet — IPFS (2014)	DHT-routed CAS network
[4]	Ateniese et al. — PDP (2007)	Remote data possession proofs
[5]	Juels & Kaliski — PoR (2007)	Retrievability proofs
[6]	Protocol Labs — Filecoin (2017)	Blockchain proof-of-storage market
[7]	W3C — ActivityPub (2018)	Federated HTTP protocol

Table 1. Key related works.

3 SYSTEM ARCHITECTURE

Meento Share is structured as a single PHP script that simultaneously acts as an HTTP server endpoint (receiving uploads and inter-node pushes) and as an HTTP client (pushing files to peers, verifying remote storage, and synchronising account state). Figure 1 illustrates the high-level architecture.

3.1 Node Components

Each node maintains the following directory structure on disk:

Path	Contents	Access
files/	Stored files named <sha256>[.ext]	Web-accessible (served directly)
info/	Per-file JSON metadata	Web-accessible (for verification)
accounts/	User account JSON files	Should be restricted
transactions/	Per-user point ledger logs	Should be restricted

Path	Contents	Access
servers.txt	Known peer URLs (one per line)	Private config
node_info.txt	Human-readable node description	Optional
public_key.txt	Node's own public key	Optional
files.txt	Flat index of stored filenames	Private
data.json	Aggregate metadata of all files	Private

Table 2. On-disk directory structure of a Meento Share node.

3.2 Account Model

Accounts are keyed by a user-supplied *public key* string. The key is sanitised — characters outside the base64url alphabet are replaced with underscores and the result is truncated to 200 characters — to produce a safe filesystem filename. The password is stored as a SHA-512 hash. Each account record carries:

- **files** — list of file records, each containing the content hash, original filename, size, extension, description, and the list of servers that have confirmed storage.
- **servers** — map of *url_key* → *raw_url* for every server that has confirmed storage of at least one file for this user.
- **dates** — map of *url_key* → *ISO-8601 datetime* recording when each server was last confirmed.
- **server_ip_hashes** — list of SHA-256 hashes of resolved server IP addresses, used to prevent the same physical machine from being counted twice under different URLs.
- **balance** — integer counter incremented once per newly confirmed, deduplicated server.

3.3 Session Model

Authentication uses a daily rotating nonce derived as **sha256(pubkeyFilename + UTC-date)**. The nonce is computed independently by both the client and the server; it is never transmitted at login. Upload requests must carry both the public key and the expected nonce. The nonce rotates at midnight UTC, limiting the validity window of a captured nonce to at most 24 hours.

Security note: The daily nonce provides replay protection within a 24-hour window but does not provide forward secrecy. Section 8 discusses the upgrade path to a cryptographically random session token.

4 REPLICATION AND VERIFICATION PROTOCOL

When an authenticated user uploads a file, the node executes the following sequence:

Step	Action	Detail
1	Store locally	The file is SHA-256 hashed. It is saved as <code>files/<hash>[.ext]</code> and a JSON metadata record is written to <code>info/<hash>.json</code> . If the file was already present the node returns immediately without pushing to peers.

Step	Action	Detail
2	Collect push targets	The union of servers.txt peers and the user's declared user_server (if set and not already listed) forms the set of push targets. Deduplication by URL key is applied before the push loop.
3	Embed account snapshot	The current account JSON is serialised and embedded in each push request. This allows the receiving peer to update its local accounts/ copy without a separate round-trip.
4	Push to peers	Each target receives a multipart/form-data POST containing the file binary, metadata fields, the peer_push=1 flag, and the embedded account JSON. cURL is preferred; file_get_contents is used as a fallback.
5	Dual-confirmation	A peer is counted as confirmed if either (a) its push response JSON contains the correct hash, or (b) a subsequent HTTP HEAD/GET against files/<hash>[.ext] on that peer returns a 2xx status. The dual rule handles firewalled peers that accept pushes but block public read access.
6	Update account	Confirmed servers are written to the file record and to the user's server map. Each new server increments the balance by one.
7	Post-upload broadcast	The fully updated account JSON (with new balance and server list) is broadcast to every confirmed peer via a dedicated account_sync POST, ensuring eventual consistency across all copies.

Table 3. Upload and replication sequence.

4.1 Peer-Push Receiver

When a node receives a peer_push=1 POST it stores the file locally, appends it to data.json, and — if a valid account_json field is present — atomically overwrites the corresponding accounts/ file. The account_sync endpoint performs the same overwrite operation independently and can be called at any time by any peer that holds a more recent copy of the account.

4.2 Verification Strategy

verifyOnServer() attempts three progressively more expensive checks in order:

- **cURL HEAD on files/<hash>[.ext]** — cheapest; most servers respond correctly.
- **cURL GET on info/<hash>.json** — fallback for servers that reject HEAD.
- **cURL GET with Range: 0-0** — downloads one byte as a last resort; HTTP 206 Partial Content counts as confirmation.

If cURL is unavailable, get_headers() is used for both checks. The \$http_response_header global-variable scoping bug (where the variable is always null inside a function unless declared with global) is avoided entirely by using cURL or get_headers(), both of which return status information as function return values.

5 SERVER DEDUPLICATION

A naive system keyed by raw URL allows the same physical machine to accumulate balance points by registering under multiple hostnames or URLs. Meento Share applies two independent deduplication

checks.

5.1 URL-Key Deduplication

A canonical URL key is derived by lowercasing the URL, stripping the scheme prefix (`http://` or `https://`), removing trailing slashes, and removing the script filename if present. The result is SHA-256 hashed. This means the following three entries in `servers.txt` all produce the same key and are treated as a single peer:

```
http://peer.example.com/node/  
https://peer.example.com/node  
http://peer.example.com/node/index_full.php
```

5.2 IP-Hash Deduplication

When a new server is reported, the hostname is resolved via `gethostbyname()` and the resulting IP address is SHA-256 hashed. The hash is stored in `server_ip_hashes` on the account. Before accepting any new server, the system checks whether its IP hash is already in that list. If it is, the server is silently rejected — the balance is not incremented and the URL is not added to the servers map.

Storing the hash rather than the raw IP address is a deliberate privacy choice: the actual IP is never persisted, making the account JSON safe to share or replicate without exposing network topology.

Scenario	URL key match?	IP hash match?	Outcome
Same URL, different scheme	YES	—	Rejected (URL dup)
Different URL, same machine	NO	YES	Rejected (IP dup)
Genuinely different server	NO	NO	Accepted, balance +1
Known server, re-confirmed	YES	—	Date updated, no balance change

Table 4. Server deduplication outcomes.

6 PROOF-OF-STORAGE INCENTIVE MECHANISM

6.1 Current Model

The current incentive mechanism is intentionally lightweight. Each time an upload confirms a server that the user has not previously recorded — and whose resolved IP does not duplicate an existing entry — the user's balance is incremented by one point and a timestamped entry is appended to the user's transaction log. The balance accumulates indefinitely; no points are deducted.

This model provides a *social proof of distribution*: the balance reflects, approximately, the number of distinct physical machines that have accepted and confirmed storage of the user's files. It is visible to anyone who reads the user's profile and can inform trust decisions — a user with a high balance has files replicated across many geographically and topologically diverse nodes.

6.2 Limitations of the Current Model

- **No challenge-response:** A peer can claim to hold a file, return `ok:true` on the push, and then immediately delete it. The system has no mechanism to challenge the peer at a later time to prove

continued possession.

- **No re-verification:** `last_verified` is updated at upload time but there is no scheduled re-check. A server that goes offline or deletes files silently retains its balance entry.
- **Trust hierarchy:** The confirming node is also the node that increments the balance. A compromised or colluding node can grant balance points for storage that never happened.
- **No economic stake:** Peers have no cost for accepting a push and no reward for continued storage, so there is no game-theoretic incentive to remain honest long-term.

7 ROADMAP: TOWARDS MODULAR, SECURE, AND DECENTRALISED PROOF OF ST

The current balance system can be evolved in discrete, independently deployable modules. Each module addresses a specific weakness identified in Section 6.2 without requiring a breaking change to the core protocol. The modules are ordered by implementation complexity from lowest to highest.

Module 1 — Periodic Re-Verification (Availability Proof)

A lightweight cron job or CLI command re-runs `verifyOnServer()` against every server in every user's account on a configurable schedule (e.g. weekly). If a server fails three consecutive re-verification attempts, its balance contribution is frozen and the user is notified. If it passes again, the contribution is restored. This requires no cryptography and no changes to the peer protocol — it is a purely local bookkeeping improvement.

- Implementation: a standalone PHP/Java CLI script reading accounts/ and calling `verifyOnServer()`.
- Cost: negligible — one HEAD request per server per file per schedule interval.
- Addresses: silent deletion, server going offline.

Module 2 — Spot-Challenge Protocol (Possession Proof)

To prove a peer still holds a file without downloading it, the verifying node can request a *spot check*: select a random byte range of the file and ask the peer to return its SHA-256 hash. The peer must read the file from disk to respond correctly. This is a simplified version of the Provable Data Possession (PDP) scheme.

The challenge is issued as a GET request with a custom header or query parameter specifying the byte range. The peer endpoint reads that range from the stored file and returns its hash. The verifying node recomputes the expected hash from its own local copy and compares.

```
GET /index_full.php?challenge=1
```

```
&hash;=c33c4f0db1d7a3fc...
```

```
&offset;=14928
```

```
&length;=512
```

```
Response: { "ok": true, "range_hash": "a8f3c2..." }
```

- A new `?challenge` endpoint on the peer, returning `sha256(file[offset:offset+length])`.
- Challenge parameters selected uniformly at random by the verifying node.
- Passes only if the peer has the complete file on disk.
- False positive probability: $1/2^{256}$ — negligible.

Module 3 — Cryptographic Account Signatures (Integrity Proof)

The `account_sync` endpoint currently accepts any JSON that claims to belong to a given public key. A malicious peer can send a modified account that inflates the balance or removes file records. The fix is to sign the account JSON with the user's private key before broadcasting, and to verify the signature on receipt.

The user holds an Ed25519 or RSA private key client-side (in the browser or Java client). After every upload, the updated account JSON is signed and the signature is embedded in the broadcast. Peers reject any `account_sync` request whose signature does not verify against the public key stored in the account.

- `public_key` field in the account JSON becomes an Ed25519 or RSA public key.
- A new `account_sig` field carries the detached signature over the canonical JSON.
- Peers verify before writing: invalid signature → 403, discard.
- The user's private key never leaves the client — zero server-side secret material.

Module 4 — Third-Party Auditors (Decentralised Verification)

In the current model the uploading node both performs and records verification — a conflict of interest. Module 4 introduces *auditor nodes*: independent Meento Share instances that do not store files but periodically issue spot challenges to storage nodes and record the results in a tamper-evident append-only log.

Any node can register as an auditor by advertising `?node_status&role=auditor`. Storage nodes that want their confirmations to carry auditor-backed weight add auditor URLs to their configuration. After each upload, the uploading node requests a challenge from a randomly selected auditor, which issues the spot check independently and co-signs the confirmation.

- Auditor list is public and can be maintained as a community-curated `servers.txt`.
- Co-signed confirmations carry more weight in balance calculations (e.g. 2 points instead of 1 for auditor-backed confirmations).
- No single auditor is trusted absolutely; requiring M-of-N auditor signatures provides Byzantine fault tolerance.

Module 5 — Erasure Coding and Distributed Redundancy

As an optional advanced module, files can be split into $k+m$ erasure-coded shards before distribution, where any k shards are sufficient to reconstruct the original. This reduces per-node storage cost (each peer stores $1/k$ of the file) while tolerating up to m node failures. The content hash of the original file remains the canonical identifier; shard hashes are recorded in the info JSON.

- A Reed-Solomon or similar erasure code is applied client-side before upload.
- Each shard is pushed to a different peer as if it were an independent file.
- The info JSON records the erasure parameters and shard hash list.
- Recovery requires contacting any k peers and reassembling the shards.

Module	What it proves	Complexity	Protocol change?
1 — Re-verification	Continued availability	Low	No
2 — Spot challenge	Current possession	Low-medium	New endpoint
3 — Account signatures	Account integrity	Medium	New field
4 — Auditor nodes	Third-party confirmation	Medium-high	New role/endpoint

Module	What it proves	Complexity	Protocol change?
5 — Erasure coding	Efficient redundancy	High	Client-side

Table 5. Incentive hardening modules and their properties.

Design principle: Each module is independently deployable and backwards compatible. A node that has not implemented Module 3 will still interoperate with nodes that have — it simply does not verify signatures on `account_sync`. Adopters can implement modules incrementally without forking the network.

8 SECURITY CONSIDERATIONS

Concern	Current mitigation	Recommended hardening
Brute-force login	None	Lockout after N failures; argon2id hashing
Nonce replay	24-hour window	Server-generated random session tokens
Password storage	SHA-512, no salt	<code>password_hash(ARGON2ID)</code>
Unauthenticated push	None	Pre-shared node secret + HMAC
Account sync forgery	Public key match check	Ed25519 account signatures (Module 3)
Directory traversal	<code>basename()</code> on filenames	Web server deny rules on accounts/
Large payload DoS	PHP <code>upload_max_filesize</code>	Application-level size cap on <code>account_json</code>
SSL stripping	<code>verify_peer=false</code> (hardcoded)	Config flag; default true for <code>https://</code> peers
Silent file deletion	None	Periodic re-verification job (Module 1)
PHP file upload	Extension blocklist	Blocklist + MIME type check

Table 6. Security concerns and mitigations.

9 JAVA COMMAND-LINE CLIENT

Alongside the PHP server component, Meento Share ships a zero-dependency Java 8+ command-line client (`MeshNode.java`) that can push files from a local `files/` directory to peer nodes without requiring a web browser or running a local PHP server.

9.1 Features

- Interactive menu: push all files, select specific files by number, list local files with metadata, list peer servers, and view account information.
- Non-interactive CLI flags: `--all` pushes every file; `--file <name> ...` pushes specific files.
- Mirrors the PHP dual-confirmation logic: push hash match or HTTP HEAD/GET.
- Updates the local account JSON after each confirmed push using the same deduplication rules as the PHP node.

- No external dependencies: JSON parsing, multipart encoding, SSL, and pretty-printing are all implemented from scratch in a single .java file.

9.2 Compile and Run

```
# Compile
javac MeshNode.java

# Interactive
java MeshNode

# Push all files non-interactively
java MeshNode --all

# Push specific files
java MeshNode --file c33c4f0db1d7.jpg abc123.png
```

10 DEPLOYMENT

A minimal Meento Share node requires only a PHP 7.4+ web server and write permission on the working directory. No database, no daemon, no build step.

```
# 1. Place index_full.php in your web root
cp index_full.php /var/www/html/node/

# 2. Create a peers file
echo 'http://peer.example.com/node' > /var/www/html/node/servers.txt

# 3. Optional: identify this node
echo 'Primary node - Amsterdam, NL' > /var/www/html/node/node_info.txt

# 4. Protect sensitive directories (Apache example)
# Add to .htaccess or virtual host:
# Deny from all
# Deny from all
```

The node creates files/, info/, accounts/, and transactions/ on the first request. No further configuration is required to begin accepting uploads and replicating to peers.

Endpoint	Method	Purpose
index_full.php	POST (JSON)	Register, login, get profile
index_full.php	POST (multipart, no peer_push)	Authenticated file upload
index_full.php	POST (multipart, peer_push=1)	Peer-to-peer file push
index_full.php	POST (account_sync=1)	Account JSON synchronisation

Endpoint	Method	Purpose
index_full.php?node_status	GET	Node health check
index_full.php?debug_verify	GET	Diagnostic verification (localhost only)
files/<hash>[.ext]	GET	Direct file download
info/<hash>.json	GET	File metadata (used by verifyOnServer)

Table 7. HTTP endpoints exposed by a Meento Share node.

11 CONCLUSION

Meento Share demonstrates that a functional federated content-addressable storage network can be built with no infrastructure beyond a shared-hosting PHP account. Files are identified by their content hash, automatically replicated to a configurable set of peers on upload, and verified through a dual-confirmation protocol that is robust to both firewalled peers and the PHP `$http_response_header` scoping bug. An account system provides user identity, file provenance tracking, and a lightweight proof-of-storage incentive in the form of a balance counter.

The incentive mechanism is honest about its current limitations — it provides social proof of distribution rather than cryptographic proof of possession — but the modular roadmap in Section 7 shows a clear, backwards-compatible path from the current model to progressively stronger guarantees: availability proofs via re-verification, possession proofs via spot challenges, integrity proofs via account signatures, and decentralised confirmation via auditor nodes.

The system is available as open-source software and can be explored at the following locations:

- <https://meento.lovable.app> — Project home and web interface
- <https://sourceforge.net/projects/meento> — SourceForge project page
- <https://github.com/rhyzomai/meentoshare> — Source code repository

This paper was prepared as a technical overview of the Meento Share project. Contributions, issue reports, and peer node registrations are welcome via the GitHub repository.