LibreMesh: Decentralized File Storage Specification

Version: Alpha 0.1 (Proof of Concept)

1. Introduction

LibreMesh is a decentralized, peer-to-peer file storage system designed for deployment on resource-constrained, easily available web hosting environments, specifically targeting standard shared hosting accounts with PHP, HTTP, and cron job capabilities. It aims to provide a resilient and distributed storage solution without relying on centralized servers or requiring complex daemon processes or root access.

The system achieves redundancy through data replication across multiple independent nodes. Communication and synchronization between nodes are performed asynchronously via HTTP requests, typically triggered by scheduled cron jobs, adhering to the limitations of the shared hosting environment.

2. Architecture

LibreMesh operates as a flat, decentralized network of independent nodes. Each node is a self-contained instance of the LibreMesh software running on a distinct web hosting account.

Components:

- **LibreNode / PeerHost:** The core software running on each shared hosting account. It is responsible for:
    - Storing file data (replicas or chunks).
    - Maintaining local metadata about stored data and known peers.
    - Participating in network discovery (gossip).
    - Synchronizing metadata with peers (gossip).
    - Performing background maintenance (cleanup, archiving).
    - Serving data and metadata to other nodes via a secured API.
    - Serving data to users via a public download API (or accessed via a Gateway).
- **Data Directory:** A local directory on each node's hosting account where file data, archives, and state files (`peers.json` , `metadata.json` , `analytics.json`) are stored. Must be writable by the web server process and ideally located outside the web root for security.
- **Cron Jobs:** Scheduled tasks on each hosting account that periodically trigger background operations within the LibreNode software (gossip, checks, cleanup, archiving).
- **Mesh Gateway (Optional, External):** A separate web-facing application (not part of the core LibreNode software) that acts as a single entry point for users. It discovers and monitors LibreNodes and routes user download requests to available nodes, potentially providing load balancing.

Interaction Model:

Nodes interact primarily via authenticated HTTP requests to each other's API endpoints. These requests are mostly initiated by cron jobs running on each node (pull-based gossip, checks) or by an entry node during a user upload (push-based data transfer).

3. Protocols

LibreMesh defines several simple HTTP-based protocols for inter-node communication. Authentication for these protocols is typically achieved using a shared network secret included in a custom HTTP header (`X-Network-Secret`).

3.1. Peer Discovery Protocol (Gossip)

- **Mechanism:** Pull-based, eventually consistent. Nodes periodically query a subset of known peers to exchange lists of known node URLs.
- **Endpoint:** `GET /api/peers.php`
- **Request:** Authenticated GET request.
- **Response:** JSON array of URLs of peers known to the queried node.
- **Process:**
    1. A node's `gossip_peers` cron job runs.
    2. It selects a small, random subset of peers from its local `peers.json` list (excluding itself).
    3. For each selected peer, it makes an authenticated `GET /api/peers.php` request.
    4. It receives the list of peers known by that peer.
    5. It merges the received list into its own local `peers.json` , adding any newly discovered URLs.
    6. The process is repeated by other nodes, leading to eventual discovery of all nodes in the network, starting from `SEED_NODES` .
- **Bootstrapping:** The initial `peers.json` on a new node is populated with a list of `SEED_NODES` defined in its `config.php` .

3.2. Metadata Synchronization Protocol (Gossip)

- **Mechanism:** Pull-based, eventually consistent. Nodes periodically exchange information about the data they hold and its state.
- **Endpoint:** `GET /api/metadata.php` (Pull) - *Note: A push or push-pull model could also be used, but pull is simpler for cron-based execution.*
- **Request:** Authenticated GET request. Parameters could be added for filtering (e.g., `?since=timestamp` for incremental updates, `?file_id=...` for specific file info). In the basic implementation, it might return metadata for a random subset of files.
- **Response:** JSON object containing metadata entries for files/chunks.
- **Data Exchanged:** File/Chunk Metadata (see Section 4.1).
- **Process:**
    1. A node's `gossip_metadata` cron job runs.
    2. It selects a small, random subset of *healthy* peers (determined by `check_peers`).
    3. For each selected peer, it makes an authenticated `GET /api/metadata.php` request.
    4. It receives metadata information from the peer.
    5. It merges the received metadata into its own local `metadata.json` .
- **Conflict Resolution (Simplified):** The basic implementation uses simple merging (e.g., adding entries if new). Robust conflict resolution (handling concurrent updates, divergent states like `active` vs. `deleted` for the same chunk on different nodes) is complex and relies on timestamps or versioning; this is a known area for potential inconsistencies in the basic version. Deletion markers typically propagate and override other states after a delay.

3.3. Data Transfer Protocol

- **Mechanism:** Push (Upload) and Pull (Download) via authenticated HTTP requests carrying raw file data or chunks.
- **Endpoint (Upload):** `POST /api/upload_chunk.php`
    - **Request:** Authenticated POST request with `multipart/form-data`. Includes file/chunk data, `file_id`, `chunk_id`, `checksum`, `source_node_id`.
    - **Response:** JSON indicating success or failure.
    - **Process (Upload Orchestration):**
        1. A user initiates an upload to an entry node (`/api/upload.php`).
        2. The entry node generates a `file_id`.
        3. (For simple replication, the entire file is treated as chunk `0`).
        4. The entry node calculates the checksum of the file/chunk.
        5. The entry node stores a local copy (`storeDataLocally`).
        6. The entry node selects `REPLICATION_FACTOR - 1` other suitable peers (based on known health and capabilities).
        7. For each selected peer, the entry node makes an authenticated `POST /api/upload_chunk.php` request, sending the file/chunk data and metadata.
        8. Receiving nodes validate the checksum and save the data locally, updating their `metadata.json`.
        9. The entry node reports the overall replication outcome to the user.
- **Endpoint (Download - Internal):** `GET /api/download_chunk.php`
    - **Request:** Authenticated GET request with `file_id`, `chunk_id`.
    - **Response:** Raw file/chunk data on success, 404 or error on failure.
    - **Process (Internal Fetch):** A node needing a chunk (e.g., for reconstruction, healing) makes an authenticated `GET /api/download_chunk.php` request to a peer it believes holds the data.

3.4. Control & Status Protocols

- **Capabilities:**
    - **Endpoint:** `GET /api/capabilities.php`
    - **Request:** Authenticated GET request.
    - **Response:** JSON object detailing the node's capabilities (PHP version, available extensions like `curl`, `zip`, hashing algorithms, `can_initiate_http`). Used by peers (`check_peers` cron) and the Gateway (`update_peer_status` cron) to understand what features a node supports.
- **Analytics:**
    - **Endpoint:** `GET /api/analytics.php`
    - **Request:** Authenticated GET request. Parameters allow requesting specific data (`?type=status`, `?type=peer_health`, `?type=downloads`).
    - **Response:** JSON object containing requested analytics data (local download counts, storage usage, status of peers known by this node, capabilities). Primarily used by the Mesh Gateway for monitoring and aggregating data.

4. Data Model

Node state is primarily stored in JSON files within the `DATA_PATH`.

4.1. File/Chunk Metadata (`metadata.json`)

Stores information about the file replicas or chunks held *locally* by this node, and potentially information learned about chunks held by *other* nodes via metadata gossip (though the basic implementation focuses on local data state).

Structure (per File ID):

```
{
  "file_id_ABC": {
    "chunks": {
      "chunk_id_0": { // For simple replication, chunk_id is typically '0'
        "local_path": "/path/to/data/fi/le/file_ABC_0.dat", // Local filesystem path (null if archived or remote)
        "archive_path": "/path/to/data/archives/archive_ABC_0.zip", // Path to archive file (if state is 'archived')
        "archive_entry_name": "file_ABC_0.dat", // Name within the archive
        "checksum": "sha256:...", // Checksum of the data (e.g., 'sha256:...')
        "size": 12345, // Original size in bytes
        "state": "active", // "active", "archived", "deleted"
        "last_accessed": 1678886400, // Unix timestamp of last access (downloaded or unarchived)
        "stored_at": 1678800000, // Unix timestamp when first stored locally
        "deleted_at": null, // Unix timestamp when marked for deletion
        "source_node": "node_XYZ" // ID of the node that sent this copy (or 'client')
      },
      "chunk_id_1": { ... }, // For future sharding
      ...
    },
    "overall_file_status": "active" // Derived status (e.g., "deleted" if all chunks deleted locally)
  },
  "file_id_XYZ": { ... },
  ...
}
```

4.2. Peer List (`peers.json`)

A simple JSON array of URLs of all known nodes in the network.

```
[
  "https://node-a.com/libremesh/",
  "https://node-b.com/libremesh/",
  "https://node-c.com/libremesh/"
]
```

4.3. Analytics Data (`analytics.json`)

Stores local statistics and the node's view of peer health.

```
{
  "download_counts": { // Count of downloads served *by this node*
    "file_id_ABC_0": 42,
    "file_id_XYZ_0": 15,
    ...
  },
  "peer_status": { // Health status and capabilities of peers *as seen by this node*
    "https://node-a.com/libremesh/": {
      "status": "ok", // "ok", "offline", "checksum_mismatch", "unknown"
      "timestamp": 1678887000, // When status was last checked
      "capabilities": { "curl": true, "archiving": false, "hashing": ["md5"], ... }
    },
    "https://node-b.com/libremesh/": { ... },
    ...
  },
  "storage_usage": { // Local disk usage
    "total": 5368709120, // Total bytes (e.g., 5GB)
    "free": 3221225472,   // Free bytes
    "used": 2147483648,   // Used bytes
    "percentage": 40.0,   // Used percentage
    "timestamp": 1678887000 // When usage was last checked
  },
  "last_check_in": 1678887000 // Unix timestamp of the last cron job execution
}
```

5. Redundancy Mechanism

Redundancy in LibreMesh is primarily achieved through **Replication**.

- **Replication Factor (R):** Defined in `config.php`. For each file uploaded, the entry node attempts to store `R` copies (replicas) across the network. In the basic implementation, this is `R-1` copies sent to peers plus the original copy stored locally by the entry node.
- **Data Distribution:** When a user uploads a file to a node (`/api/upload.php`), that node acts as the entry point. It stores the initial copy locally and then selects `R-1` suitable peers from its known `ok` peers (based on peer health and capabilities) to send additional copies using the `POST /api/upload_chunk.php` protocol.
- **Download Redundancy:** The Mesh Gateway (or a node acting as a download entry point) knows about multiple nodes that might hold a copy of a file (inferred from the peer list and potentially metadata gossip). If a download attempt from the first selected node fails (e.g., node offline, reports 404), the Gateway/node can attempt to download the file from another healthy peer known to potentially have a copy.
- **Healing (Conceptual):** The current proof-of-concept does not include an automated data healing protocol. A future version would involve nodes periodically checking the replication level of files they know about (via metadata gossip) and, if a file is under-replicated (fewer than R live copies detected across the network), initiating a repair process by fetching a copy from a healthy node and replicating it to another available healthy node.

6. File Lifecycle

1. **Upload:** User sends file to `/api/upload.php` on an entry node. Entry node assigns `file_id`, stores locally, and replicates to `R-1` peers via `POST /api/upload_chunk.php`. Metadata updated to `state: active`.
2. **Replication:** Receiving peers validate, store locally, and update their `metadata.json`. Metadata changes propagate via `gossip_metadata` cron.
3. **Active State:** File/chunk is stored on `R` nodes and accessible. `last_accessed` timestamp is updated on download (`GET /api/download_chunk.php`).
4. **Inactivity:** If `last_accessed` is older than `ARCHIVE_THRESHOLD_DAYS`, the `archive_old_files` cron task identifies the file/chunk.
5. **Archiving:** If the node has the `zip` extension (`archiving: true` capability) and space allows, the file is compressed, stored in the archive directory, the original is deleted, and metadata is updated to `state: archived`, storing `archive_path`, `archive_entry_name`, and `original_size`. Metadata change propagates via gossip.
6. **Accessing Archived:** A download request for an archived file (`GET /api/download_chunk.php`) triggers `unarchiveDataLocally`. The file is extracted to a temporary location, served, and the temporary copy is cleaned up. `last_accessed` is updated. (Metadata state *could* be set back to `active` here, but adds complexity; basic version updates timestamp and relies on re-archiving check).

7. **Deletion Request:** User/system marks a file for deletion (e.g., via a future `/api/delete.php`). Metadata on the receiving node is updated to `state: deleted` with a `deleted_at` timestamp. This change propagates via gossip.
8. **Cleanup:** The `cleanup_data` cron task runs. It identifies local files/archives where the metadata is marked `state: deleted` and the `deleted_at` timestamp is older than `DELETE_PROPAGATION_DELAY_HOURS`. These files are physically deleted (`unlink()`). (Removing the metadata entry after physical deletion is complex in an eventually consistent system and not fully implemented in the basic code). Orphaned files (on disk but not in metadata) are also deleted.

7. Fault Tolerance & Limitations

- **Node Failure (Offline):** If a node goes offline, the `check_peers` cron on other nodes will detect it (`status: offline`). Data stored *only* on that node becomes temporarily inaccessible from that node. Redundancy ensures copies exist on other nodes (if `R > 1`). The Gateway/download logic can try other healthy nodes.
- **Network Partitioning:** If a group of nodes cannot communicate with another group (e.g., due to hosting provider issues), they form temporary partitions. Gossip and metadata sync will only occur within partitions. Data consistency can diverge. Consistency is restored once communication is re-established and gossip merges state.
- **Data Loss:** Loss of a node results in loss of the replicas stored there. If the number of lost replicas for a file exceeds `R-1`, the file becomes inaccessible until replaced (healing).
- **Consistency:** LibreMesh is an *eventually consistent* system. There is no guarantee that all nodes have the exact same view of the network or metadata at any given moment. Updates (like new files or deletions) take time to propagate via gossip.
- **Shared Hosting Limitations:** The reliance on cron jobs for background tasks means updates and network reactions are not real-time. Resource limits (CPU, memory, disk I/O, network transfer) on shared hosting can impact performance and reliability, especially for large files or frequent operations. Network restrictions (like anti-bot JS challenges) can prevent inter-node communication entirely.
- **No Strong Integrity Checks:** While checksums verify data integrity *during transfer* and *locally*, there's no Merkle tree or similar structure to verify the integrity of the *entire file* across distributed chunks/replicas in a robust, decentralized way in the basic version.

8. Security Considerations (Basic)

- **Authentication:** Node-to-node API calls and Gateway-to-node analytics calls are protected by a shared `NETWORK_SECRET`. This prevents arbitrary parties from triggering internal operations or accessing analytics.
- **Data Access:** User-facing download APIs (`/api/download.php` or via Gateway) are typically not authenticated in the basic model, making stored files public to anyone with the File ID.
- **Data Confidentiality:** LibreMesh itself does **not** provide encryption of data at rest or in transit (beyond HTTPS if configured on the hosting). **Client-side encryption of sensitive data BEFORE uploading is strongly recommended.**
- **Hosting Security:** Security is heavily dependent on the shared hosting provider's security measures. The `DATA_PATH` should ideally be outside the web root. PHP `open_basedir` and disabled dangerous functions are important protections provided by the host.

9. Future Work

- Implementation of M-of-N Sharding.
- Sophisticated Metadata Conflict Resolution (e.g., using Vector Clocks or CRDTs).
- Automated Data Healing and Repair Protocol.
- More advanced Peer Selection and Load Balancing (considering load, latency, geography).
- Secure Deletion Protocol (ensuring R copies are marked deleted before physical removal).
- User Management and Access Control (Authentication/Authorization for uploads/downloads).
- End-to-end Data Integrity Verification (e.g., Merkle Trees).
- Incentive Layer (for contributing storage/bandwidth).
