Distributed Metadata Management System

Distributed Systems Final Project

Călin Anda-Maria
Babeș-Bolyai University
Faculty of Economics and Business Administration

January 30, 2026

Contents

1. Introduction

Distributed systems are widely used to support applications that must scale, remain available under failures, and handle concurrent access to shared resources. In such systems, metadata management plays a critical role, as it enables clients to locate, access, and coordinate data efficiently. As systems grow in size and complexity, centralized metadata solutions become increasingly difficult to scale and maintain.

This project explores the design and implementation of a simplified distributed metadata management system. The focus is on demonstrating key distributed systems concepts, including metadata partitioning, service discovery, fault tolerance, concurrency control, and caching. Rather than targeting production-level performance, the system is designed as an educational prototype that emphasizes clarity, modularity, and correctness.

The report presents the problem context, related work, system architecture, functional and technical requirements, implementation details, and evaluation results. Together, these sections illustrate how core distributed systems principles can be applied to build a fault-tolerant and scalable metadata service.

2. Problem Description and Motivation

Distributed systems are widely used to support applications that require scalability, high availability, and resilience to partial failures. In such systems, data management is commonly divided into two responsibilities: storing the actual data and managing its metadata. Metadata includes descriptive information such as file identifiers, ownership, size, timestamps, and versioning, and is required by almost every client operation before accessing data.

In distributed file systems, metadata management becomes a critical challenge as the system scales [1]. Centralized metadata services can easily turn into performance bottlenecks and single points of failure [2], limiting throughput and reducing fault tolerance. For this reason, modern systems increasingly adopt distributed metadata management, where metadata is spread across multiple nodes to improve scalability and availability.

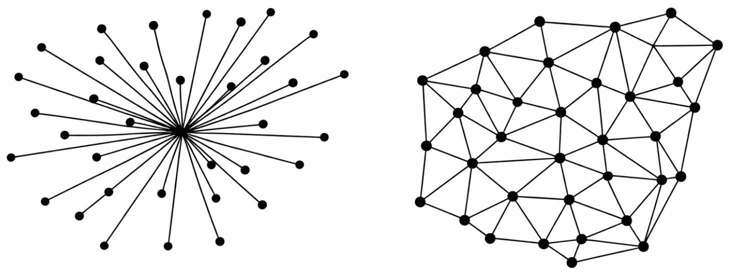
Figure 1: Centralized metadata management vs Distributed metadata management [9]

As illustrated in Figure 1, adapted from existing system architecture comparisons [9], distributing metadata across multiple nodes removes the single point of failure and allows the system to scale horizontally. However, this approach introduces new challenges, particularly related to fault tolerance, coordination, and consistency.

In real distributed environments, node failures are expected rather than exceptional. Metadata systems must therefore detect unavailable nodes and continue operating without manual intervention. This requires service discovery mechanisms, health monitoring, and dynamic reconfiguration. Furthermore, concurrent access to metadata by multiple clients can lead to race conditions if not properly controlled, making concurrency control a fundamental requirement.

To reduce access latency and limit load on metadata nodes, caching is often employed, especially since metadata workloads are typically read-heavy. At the same time, caching introduces trade-offs between performance and freshness, which must be managed through mechanisms such as cache invalidation and time-to-live policies.

To address these challenges, distributed systems typically introduce coordinating components and discovery mechanisms that decouple clients from individual storage nodes and enable dynamic reconfiguration.

The motivation of this project is to design and implement a simplified distributed metadata management system that captures these core challenges while remaining easy to understand and evaluate. By focusing exclusively on metadata rather than full data storage, the project highlights essential distributed systems concepts such as partitioning, service discovery, fault tolerance, concurrency control, and caching. The resulting system serves as an educational prototype that demonstrates how architectural decisions impact scalability, availability, and robustness.

3. Related Work & Existing Solutions

Metadata management is a fundamental component of distributed file systems, as it directly impacts system scalability, availability, and performance. Existing approaches generally fall into two broad categories: centralized and distributed metadata management.

Centralized metadata architectures are commonly used in early and widely adopted distributed file systems, such as the Hadoop Distributed File System (HDFS) [2]. In this model, a single metadata server is responsible for managing the file system namespace and handling all metadata operations. While this design simplifies coordination and consistency, it introduces scalability limitations and creates a single point of failure as system size and workload increase.

To address these limitations, distributed metadata management has been proposed and adopted in more scalable systems. In distributed approaches, metadata is partitioned across multiple nodes, allowing the system to scale horizontally and tolerate node failures. The overview provided by GeeksforGeeks highlights common challenges in such systems, including metadata partitioning, fault tolerance, concurrency control, and consistency management [1].

Practical implementations of distributed metadata services can be found in parallel and distributed file systems such as BeeGFS, which distributes metadata across multiple metadata servers to improve performance and availability. These systems demonstrate that decentralizing metadata can effectively reduce bottlenecks while increasing resilience.

In addition to production systems, simplified open-source prototypes explore distributed metadata concepts for educational and experimental purposes. The *pydfs* project illustrates the separation of metadata management from data storage and exposes file-system-like metadata operations. Similarly, the *py-dist-fs* project focuses on node coordination and liveness detection through heartbeat mechanisms. These projects provide practical insights into metadata distribution and failure handling, which informed the design choices of this project [4, 5].

4. Proposed Solution and System Architecture

The proposed solution introduces a distributed metadata management system designed to address the scalability and fault-tolerance limitations of centralized approaches [1, 2]. The system is structured around the idea of separating coordination, discovery, and metadata storage responsibilities into distinct components, each with a well-defined role. This modular design improves scalability, resilience, and clarity, while remaining suitable for educational and experimental purposes.

At a conceptual level, the system consists of three main components: a service registry, a gateway and multiple metadata nodes. Together, these components form a lightweight distributed architecture that enables metadata partitioning, dynamic service discovery, and continued operation in the presence of partial failures. The interaction between the main components of the proposed system is illustrated in Figure 2.

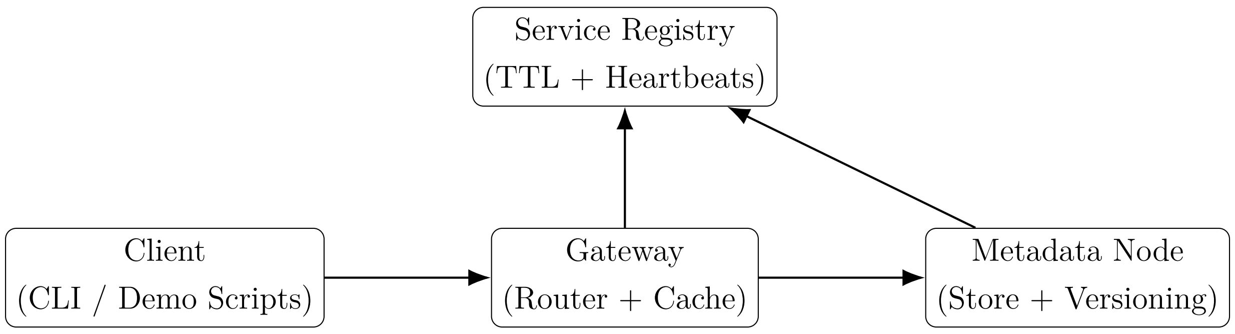

Figure 2: High-level metadata access flow and component interaction in the proposed system

The service registry is responsible for maintaining information about active metadata nodes, a common approach for service discovery in distributed systems [5]. Rather than acting as a coordinator for client requests, the registry functions as a discovery mechanism that tracks node availability. Metadata nodes periodically announce their presence to the registry, allowing the system to dynamically adapt to node failures or restarts without manual intervention.

The gateway acts as the single entry point for clients interacting with the metadata system. Clients are intentionally decoupled from the internal structure of the system and are unaware of individual metadata nodes. The gateway is responsible for routing metadata requests to the appropriate metadata node, based on a deterministic strategy. By centralizing request routing and coordination at the gateway level, the system simplifies client interaction while preserving flexibility in how metadata is distributed internally. This design choice avoids exposing internal metadata nodes to clients and simplifies system evolution and scaling.

Metadata nodes are responsible for storing and managing metadata entries. Each metadata node maintains a subset of the global metadata space, determined by a partitioning strategy. Metadata entries include information such as file identifiers, ownership, size, timestamps, and versioning information. Distributing metadata across multiple nodes enables horizontal scalability and prevents any single node from becoming a performance bottleneck. The distribution of metadata across nodes is conceptually illustrated in Figure 3, where the global metadata space is divided into disjoint partitions. For simplicity, the proposed system does not implement metadata replication, focusing instead on partitioning and fault detection.

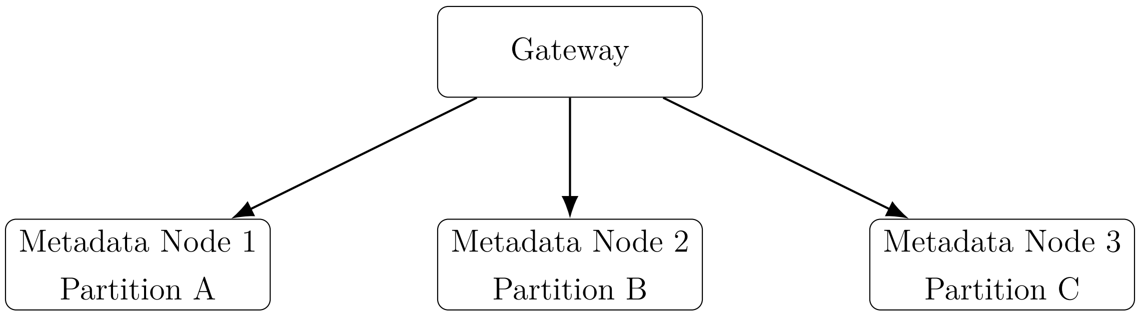

Figure 3: Conceptual metadata partitioning across multiple metadata nodes.

Figure 2 illustrates the high-level interaction between these components. Client requests are first handled by the gateway, which consults the service registry to identify active metadata nodes. Requests are then forwarded to the appropriate metadata node, which processes the operation and returns the result through the gateway. This interaction pattern ensures that the system remains operational as long as at least one metadata node is available.

Overall, the proposed architecture emphasizes simplicity and conceptual clarity while incorporating key distributed systems principles. By combining metadata partitioning, service discovery, and fault-tolerant coordination, the system demonstrates how distributed metadata management can be achieved without relying on complex infrastructure or tightly coupled components.

5. Functional and Technical Requirements

This section describes the functional and technical requirements that enable the proposed distributed metadata management system to operate correctly and efficiently [1]. The requirements define the system's core components, their responsibilities, and the mechanisms through which they interact in a distributed environment.

5.1. Functional Requirements

The system must support a set of core metadata operations that are representative of a distributed file system environment. These operations are exposed to clients through a unified entry point and are executed transparently across distributed metadata nodes.

- **Metadata creation and update:** The system must allow clients to create and update metadata entries associated with file identifiers. Metadata includes attributes such as ownership, size, timestamps, and versioning information.
- **Metadata retrieval:** Clients must be able to retrieve metadata entries using a file identifier, without needing to know which metadata node stores the corresponding information.
- **Metadata removal:** The system must support the removal of metadata entries and correctly handle requests for non-existing entries.
- **Metadata listing:** The system should provide directory-like listing functionality, allowing clients to retrieve metadata entries that share a common prefix.
- **Transparency to clients:** Clients must interact with the system through a single access point and remain unaware of internal data distribution and node membership.

5.2. Inter-Process Communication

Inter-process communication between distributed components is achieved using a REST-based communication model over HTTP. This choice provides loose coupling between components, language independence, and ease of debugging.

The gateway, service registry, and metadata nodes communicate exclusively through well-defined HTTP endpoints. This approach avoids tight coupling and allows individual components to be restarted or replaced without affecting the rest of the system. Alternative communication mechanisms, such as message queues or RPC frameworks, were considered but rejected due to their increased complexity and configuration overhead for an educational prototype.

5.3. Concurrency Control

Concurrency control is required to ensure correctness when multiple clients or services simultaneously access shared metadata. Without proper synchronization, concurrent updates may lead to inconsistent states or lost updates.

The system enforces concurrency control at the metadata node level. Each metadata node ensures that operations affecting the same metadata entry are executed atomically. By localizing synchronization to individual metadata nodes rather than enforcing global locks, the system avoids unnecessary contention and preserves scalability.

For example, concurrent update requests for the same file identifier are serialized at the responsible metadata node, ensuring that versioning and timestamps remain consistent.

5.4. Service Discovery

Service discovery is a critical requirement in dynamic distributed environments, where nodes may join or leave the system at runtime. The system employs a registry-based service discovery mechanism to track active metadata nodes, following common patterns used in distributed systems [5].

Metadata nodes periodically register their availability with the service registry. The registry maintains a view of currently active nodes and removes entries that have not been refreshed within a predefined time window. The gateway queries the registry to obtain the list of available metadata nodes and routes requests accordingly.

This design allows the system to adapt dynamically to node failures or restarts without manual reconfiguration. Alternative approaches, such as static configuration files, were avoided due to their lack of flexibility and fault tolerance.

5.5. Data Partitioning Strategy

To improve scalability and performance, metadata is partitioned across multiple metadata nodes. Each metadata node is responsible for a subset of the global metadata space, determined by a deterministic partitioning strategy [1].

Partitioning ensures that metadata operations are distributed across nodes, preventing any single node from becoming a bottleneck. Deterministic routing guarantees that requests for the same metadata entry are consistently routed to the same node, simplifying concurrency control and avoiding coordination overhead between nodes.

For instance, when multiple metadata nodes are active, different file identifiers are routed to different nodes, resulting in balanced load distribution. Compared to centralized metadata storage, this approach significantly improves scalability and resilience.

5.6. Design Choices and Justification

Several design decisions were made to balance correctness, simplicity, and scalability. The use of a gateway simplifies client interaction and decouples clients from internal system structure. Registry-based service discovery enables dynamic membership and fault tolerance. REST-based communication offers simplicity and transparency, while deterministic partitioning provides efficient and predictable metadata placement.

More complex alternatives, such as metadata replication, distributed consensus protocols, or advanced coordination services, were intentionally excluded to keep the system focused on core distributed systems concepts. This allows the system to remain understandable while still demonstrating key principles such as partitioning, fault tolerance, concurrency control, and service discovery.

6. Implementation Details

This section presents the implementation of the proposed distributed metadata management system, focusing on the structure of the system and the interactions among its main components. Code excerpts are included selectively to illustrate key mechanisms, while complete implementations and auxiliary scripts are provided in the appendices.

6.1. Service Registry

The service registry is implemented as an independent service that tracks active metadata nodes using a heartbeat-based mechanism. Metadata nodes periodically send registration messages (heartbeat) containing their identifier and network address. Upon receiving such a message, the registry updates the node's last-seen timestamp. Rather than storing an explicit countdown timer, the TTL is enforced implicitly by comparing the current time with the timestamp of the last received heartbeat.

Listing 1: Service registry: node registration with TTL

```
NODES: dict[str, dict] = {}

NODES[node_id] = {"base_url": base_url, "last_seen": time.time()}
```

Nodes that fail to refresh their registration within the configured time window are considered unavailable and automatically removed. During registry queries, nodes whose last-seen timestamp exceeds the TTL threshold are automatically removed from the registry's active node list. As a result, nodes that crash, become unreachable, or stop sending heartbeats are excluded without requiring explicit failure notifications.

This approach enables dynamic service discovery and fault tolerance without requiring manual reconfiguration. The registry implementation is inspired by lightweight liveness tracking approaches used in the *py-dist-fs* project [5].

6.2. Gateway

The gateway acts as the central coordination component and the single entry point for all client interactions. Its main responsibilities are deterministic request routing, metadata caching, request forwarding, and service discovery integration. By design, clients are decoupled from the internal structure of the system and remain unaware of individual metadata nodes.

The gateway is implemented as an HTTP service using the **FastAPI framework**. FastAPI was chosen due to its support for asynchronous request handling, clear API definition, and low overhead, making it suitable for implementing lightweight distributed services. Its native integration with asynchronous Python constructs enables efficient handling of concurrent client requests without introducing unnecessary complexity.

Deterministic routing is used to ensure consistent metadata placement. The gateway computes a hash of the file identifier and uses it to select a target metadata node from the set of currently active nodes. This guarantees that all operations for the same metadata entry are routed to the same node as long as the node set remains unchanged, simplifying concurrency control and avoiding cross-node coordination.

Listing 2: Deterministic routing based on file identifier

```
h = int(hashlib.sha256(file_id.encode()).hexdigest(), 16)
target_index = h \% len(node_ids)
target_node = node_ids[target_index]
```

To improve performance for read-heavy workloads, the gateway implements a lightweight **metadata cache**. Cached entries are associated with a time-to-live value and are automatically invalidated upon expiration or after update and delete operations. This approach reduces latency and minimizes unnecessary requests to metadata nodes while preserving correctness.

Listing 3: Gateway cache entry with TTL

```
CACHE_TTL_SECONDS = int(os.getenv("CACHE_TTL_SECONDS", "20"))
_CACHE: dict[str, tuple[float, dict[str, Any]]] = {}

_CACHE[file_id] = (time.time() + CACHE_TTL_SECONDS, payload)
```

All inter-component communication initiated by the gateway is performed using the asynchronous `httpx` client over RESTful HTTP APIs. An asynchronous communication model allows the gateway to handle multiple concurrent client requests efficiently without blocking while waiting for network responses. Compared to synchronous HTTP clients, this approach improves responsiveness in distributed environments where network latency is unavoidable.

Request forwarding and fault tolerance are handled through a dedicated function: `forward_with_fallback`. If the initially selected metadata node is unavailable, the gateway retries the request on alternative nodes in a deterministic order. The complete implementation of this function is provided in Appendix Listing 5.

The gateway exposes a small set of REST endpoints corresponding to metadata creation, retrieval, deletion, listing, and cache invalidation. Clients interact only with the gateway, while the gateway coordinates metadata access and hides distribution details from clients.

6.3. Metadata node

Each metadata node stores and manages a subset of the global metadata space. To ensure correctness under concurrent access, metadata nodes enforce fine-grained concurrency control at the level of individual metadata entries.

Listing 4: Fine-grained concurrency control at metadata node

```
lock = PATH_LOCKS[file_id]
async with lock:
    existing = STORE.get(file_id)
    meta.version = 1 if existing is None else existing.version +
       1
    STORE[file_id] = meta
```

This approach guarantees atomic updates for concurrent operations targeting the same file identifier while avoiding global locks that could limit scalability. The metadata node design is influenced by simplified distributed file system prototypes such as *pydfs* [4].

7. Evaluation and Testing

This section describes the goals and methodology used to evaluate the proposed distributed metadata management system. The evaluation focuses on validating the correct interaction between system components and demonstrating key distributed system properties rather than on exhaustive performance benchmarking. Detailed execution outputs and scripts are provided in the appendices.

7.1. Testing Goals

The primary goal of testing is to verify that the system behaves correctly under realistic usage scenarios and that the architectural decisions described earlier are reflected in practice. Specifically, testing aims to validate correct metadata operations, deterministic routing, metadata distribution across nodes, fault tolerance through service discovery, and the effectiveness of caching mechanisms.

Rather than measuring raw performance metrics, the evaluation emphasizes functional correctness and architectural behavior, which are more appropriate for an educational prototype of a distributed system.

7.2. Testing Methodology

Testing is conducted through a combination of a command-line client and a set of predefined demo scripts. All interactions with the system occur through the gateway, ensuring that the internal distribution of metadata nodes remains transparent to clients.

The command-line interface, implemented in `cli.py`, provides a convenient way to issue metadata operations such as creation, retrieval, deletion, listing, cache invalidation, and cluster inspection. This interface mimics the behavior of a real client and allows manual experimentation with the system without exposing internal implementation details.

In addition to manual interaction through the CLI, automated demo scripts are used to execute predefined scenarios. Each demo script targets a specific aspect of the system's behavior and produces observable outputs that can be inspected to verify correctness.

7.3. Functional Validation Scenarios

Basic Functionality and Caching (Demo 01) The first demo focuses on validating core metadata operations, including metadata creation, retrieval, update, deletion, and prefix-based listing. This scenario also demonstrates the behavior of the gateway cache by issuing repeated read requests for the same metadata entry. Cache hits and misses confirm that cached metadata is reused when valid and refreshed when necessary.

Metadata Partitioning and Distribution (Demo 02) The second demo evaluates metadata distribution across multiple metadata nodes. A larger number of metadata entries with different identifiers are created, and the resulting distribution is inspected using cluster statistics. This scenario demonstrates deterministic routing and shows that metadata entries are partitioned across nodes rather than concentrated on a single metadata server.

Fault Tolerance and Failover (Demo 04) The third demo validates fault tolerance and service discovery behavior. After creating and retrieving metadata entries, one metadata node is manually stopped. The demo verifies that the registry removes the failed node after the TTL expires and that the gateway continues to serve requests using the remaining active node. This scenario confirms that the system tolerates partial failures without manual reconfiguration.

Together, these testing scenarios demonstrate that the system satisfies its functional and architectural goals. Metadata operations behave correctly, metadata is distributed across nodes, caching improves read behavior, and the system remains operational in the presence of node failures. Detailed execution logs and example outputs are included in the appendices to support these observations.

8. Conclusion

This project presented the design and implementation of a distributed metadata management system that addresses key challenges associated with centralized metadata architectures. The primary outcome of the project is a functional prototype that demonstrates correct metadata operations, deterministic distribution across nodes, and fault-tolerant behavior in the presence of node failures.

The main contributions include a modular system architecture combining a gateway, service registry, and distributed metadata nodes, as well as the integration of deterministic routing, heartbeat-based service discovery, fine-grained concurrency control, and gateway-level caching. Through targeted evaluation scenarios, the system was shown to satisfy its functional and architectural goals.

Overall, the project provides a concise and practical illustration of core distributed systems concepts and serves as an educational foundation for further experimentation and extension.

References

[1] GeeksforGeeks, *Metadata Management in Distributed File Systems.*
Available at: https://www.geeksforgeeks.org/system-design/metadata-management-in-distributed-file-systems/

[2] Apache Hadoop, *HDFS Architecture.*
Available at: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[3] BeeGFS, *BeeGFS Architecture Overview.*
Available at: https://www.beegfs.io/docs/

[4] Dayyass, *pydfs – Python Distributed File System.*
GitHub repository: https://github.com/dayyass/pydfs

[5] Kunalx86, *py-dist-fs.*
GitHub repository: https://github.com/kunalx86/py-dist-fs

[6] FastAPI Documentation
Available at: https://fastapi.tiangolo.com/

[7] Python Software Foundation, "asyncio – Asynchronous I/O,"
Available at: https://docs.python.org/3/library/asyncio.html

[8] HTTPX Documentation
Available at: https://www.python-httpx.org/

[9] S. Nakamoto et al., *Centralized vs Distributed Ledger Architecture*, ResearchGate, 2018.
Available at: https://www.researchgate.net/figure/Centralized-vs-Distributed-Ledger_fig2_328408283

A. Gateway Service: `forward_with_fallback`

Listing 5: Gateway forwarding with deterministic fallback

```python
async def forward_with_fallback(method: str, file_id: str, nodes:
    dict[str, str], payload: dict[str, Any] | None = None):

    order = ordered_nodes(file_id, nodes)
    if not order:
        raise HTTPException(status_code=503, detail="No metadata
           nodes available")
    last_err = None
    for nid in order:
        base = nodes[nid]
        url = f"{base}/metadata/{file_id}"
        try:
            async with httpx.AsyncClient(timeout=5) as client:
                if method == "PUT":
                    r = await client.put(url, json=payload)
                elif method == "GET":
                    r = await client.get(url)
                elif method == "DELETE":
                    r = await client.delete(url)
                else:
                    raise ValueError("Unsupported method")
            # file doesn't exist
            if r.status_code == 404:
                return {"_status_code": 404, "detail": "Not found
                   ", "routed_to": nid}
            # success
            if r.status_code < 400:
                data = r.json()
                if isinstance(data, dict):
                    data["routed_to"] = nid
                return data
            last_err = f"{r.status_code}: {r.text}"
        except Exception as e:
            last_err = str(e)
            continue
    raise HTTPException(status_code=503, detail=f"All metadata
       nodes unreachable. Last error: {last_err}")
```