Distributed File System Design Report

---

1. Introduction

DistriFS is a distributed file system engineered to provide a high degree of availability, fault tolerance, and scalability across a cluster of storage nodes. The system is designed to decouple the control plane from the data plane, allowing the Master node to focus on metadata management, replication coordination, and client orchestration, while the Data Nodes handle the storage and retrieval of file chunks. This separation not only improves performance but also ensures that system failures can be isolated and handled efficiently.

The purpose of this document is to describe the architectural and design decisions underlying DistriFS, explain the rationale for key choices such as the custom TCP protocol, chunk replication, and write pipeline, and to explicitly address potential failure scenarios. This document serves as a reference for developers, maintainers, and stakeholders evaluating the system's resilience and correctness.

---

2. System Architecture

The architecture of DistriFS is built around the division between the control plane and the data plane. The **control plane**, represented by the Master node, is responsible for maintaining metadata such as file-to-chunk mappings, chunk placement across nodes, and replication counts. It also coordinates write and replication pipelines, ensuring that file writes are propagated safely and consistently across the cluster. All metadata changes are logged in a Write-Ahead Log (WAL), which guarantees that even in the event of a crash, the Master can restore a consistent state without data loss.

The **data plane** consists of the worker nodes, each of which stores chunks and responds to read and write requests from clients. These nodes periodically report their health and load statistics to the Master, enabling the system to make intelligent decisions about chunk placement and replication. Clients interact with the system by breaking files into fixed-size chunks, requesting placement guidance from the Master, and transferring the chunks directly to the designated Data Nodes using a lightweight, custom TCP protocol.

This separation of responsibilities ensures that the system can scale horizontally: additional nodes increase storage capacity and throughput without overloading the Master. At the same time, the control plane maintains a global view of the cluster, enabling efficient replication and fault recovery.

---

3. Design Choices and Protocols

The system communicates via a **custom TCP protocol**, chosen for its low overhead, flexibility, and direct control over messaging semantics. Each message contains a header specifying the type, length, and checksum, followed by a payload that may include metadata in JSON format or binary chunk data. This design allows the system to handle both small control messages and large data transfers uniformly while enabling straightforward debugging and logging.

Files are divided into fixed-size chunks, typically 64 MB each, and each chunk is replicated across multiple nodes to meet the configured replication factor, which defaults to three. The replication mechanism ensures that if any node fails, copies of the chunks remain available elsewhere in the cluster. The **Replication Manager** monitors chunk health and initiates re-replication whenever replicas are missing, ensuring durability without blocking client operations.

Metadata management relies on a combination of in-memory structures and the WAL. All metadata mutations are first appended to the WAL before updating in-memory tables. This ensures that even if the Master crashes during a write operation, all committed metadata can be restored by replaying the WAL. Concurrency is carefully managed using locks for metadata updates and replication state changes, preventing race conditions when multiple clients perform concurrent writes.

The **client write pipeline** is designed to maximize throughput and reliability. When a client wishes to write a file, it requests the Master to assign target nodes for each chunk. The client then uploads each chunk through a pipelined sequence, where the first node receives the data, which is subsequently replicated to the other target nodes. The Master only acknowledges the write after the WAL is persisted and replication is initiated, ensuring that system consistency is maintained even in the event of node or client failures.

---

4. Failure Scenarios

Failure handling is central to the design of DistriFS. Several scenarios have been considered to ensure both data integrity and operational continuity.

If the **Master crashes during a write pipeline operation**, in-progress metadata may reside only in memory. However, because all committed changes are first recorded in the WAL, the Master can replay the log upon restart, restoring a consistent metadata state. Any chunks partially written during the crash are identified as incomplete by the replication manager and re-replicated to maintain the desired replication factor. Clients may need to retry writes for chunks affected by the crash, but overall system consistency is preserved.

If a **client crashes after sending 99% of a chunk**, the system ensures that partial data does not corrupt stored files. The Master monitors replication counts and identifies incomplete chunks, marking them as invalid. The replication manager can then restore missing replicas from other nodes. When the client retries, it is able to resume writing without risk of duplication or corruption, preserving atomicity at the chunk level.

Node failures during write or replication are handled transparently. The Master marks the failed node as offline, and the replication manager immediately begins restoring lost replicas to healthy nodes. Any client operations in progress are rerouted to alternative nodes, ensuring that writes do not fail silently. WAL integrity ensures that all metadata changes remain recoverable despite node failures.

Even in the unlikely event of WAL corruption, the system mitigates data loss by maintaining periodic backups of the log and performing checksum verification on each entry. During recovery, the backup log can be applied to restore a consistent metadata state, ensuring that system integrity is never compromised.

---

5. Data Integrity and Security

DistriFS employs checksums for all chunks to guarantee data integrity during transfer and storage. Each chunk is verified upon receipt and periodically revalidated to detect silent corruption. While encryption is optional, the system design allows for secure transfer protocols between clients and nodes, and the Master signs all allocation messages to prevent unauthorized or rogue clients from affecting cluster metadata.

---

6. Observability and Monitoring

The system provides observability into cluster health and performance. Metrics such as the number of active nodes, total chunk counts, replication status, and namespace state are continuously monitored. The WAL serves as an audit trail, enabling administrators to track all metadata changes. Optional logging allows for detailed monitoring of client pipelines, node failures, and replication events, supporting operational troubleshooting and performance tuning.

---

7. Conclusion

DistriFS is designed to provide a highly available, consistent, and durable distributed file system. Its architecture separates control from data, enabling scalable and resilient operation.

The combination of a custom TCP protocol, pipelined client writes, WAL-based metadata persistence, and automated replication ensures that the system can tolerate failures at the client, node, or Master level without data loss. By considering and explicitly designing for failure scenarios, the system achieves a balance between performance, reliability, and operational simplicity, making it suitable for production deployment in environments that require robust distributed storage.