MiniDOS: A Peer-to-Peer Distributed Operating System Prototype

Abhi Tundiya (202211095)
Department of Computer Science
IIIT Vadodara - International campus diu

***Abstract*—This paper presents MiniDOS, a peer-to-peer distributed operating system prototype that enables multiple Windows-based nodes to operate as a unified system. Here we will implement unified file system using python. Also setup connection between nodes using TCP and UDP protocols.**

***Index Terms*—distributed operating system, peer-to-peer, File management system, network protocols.**

I. Introduction

The concept of distributed operating systems (DOS) originated from the growing need to utilize computing resources efficiently across multiple machines (1). As early as the 1970s, researchers envisioned systems that could allow several computers to collaborate seamlessly, providing the illusion of a single coherent environment (2). Early distributed systems like Amoeba (2), V-System, and Plan 9 (4) from Bell Labs demonstrated the potential of such architectures by unifying distributed resources under a single namespace.

Over time, distributed computing models evolved from simple client-server systems to peer-to-peer (P2P) architectures, grid computing, and eventually cloud systems. However, most modern distributed platforms are application frameworks rather than true operating systems. In contrast, a distributed OS attempts to abstract all resources—CPU, memory, files, and devices—into one logical entity that hides the complexity of distribution from the user (1).

This paper introduces **MiniDOS**, a modern reinterpretation of these principles. MiniDOS is a lightweight, fully peer-to-peer distributed operating system prototype implemented in Python, designed to run on Windows machines. Unlike centralized orchestration tools such as Jenkins or Docker Swarm, MiniDOS provides no master node. Instead, all peers communicate symmetrically, exchange resource states, and manage load cooperatively. The result is a transparent environment where users can execute processes, access files, and migrate tasks without concern for their physical location.

II. Background and Related Work

Early distributed operatings systems like Amoeba (1980s) introduced process migration and rpc based communication to the world. Mainly these distributed operating systems focused on fault tolerance and transparency. Early distributed operating systems primarily focused on transparency and fault-tolerance. Operating systems like sprite introduced network transparency and remote file caching. Plan 9 (4) from Bell Labs redefined the concept by representing every resource—including processes and files—as part of a hierarchical namespace accessible over the network.

In Modern days Kubernetes manages distributed workloads but they operate at application layer. lacking the operating system abstraction. MiniDOS differentiates itself by adopting a microkernel-inspired user-space design that blends process management, distributed file operations, and peer-level cooperation.

III. System Goals

MiniDOS aims to address the limitations of the centralized disrtibuted systems by implementing peer 2 peer distributed operating system layer specifically designed for LAN- Windows based. The primary goals include:

A. Decentralization

System eliminiates single point of failure by adapting and implementing a pure peer to peer architecture where all nodes process equal authority. unlike traditional client-server model, it operates without master-node, ensuring continued operation even when individual node fall.

B. Resource Sharing Transparently

Drawing inspiration from plan 9's unified namespace concept (4), Mini DOs provides a replicated global file system accessible from all connected nodes. Users Interact with it using familiar interface without knowing data replication mechanism and location of data.

C. Lighweight Implementation

It prioritize simplicity and efficiency by providing core distributed os, process monitoring, network coordination.

D. Windows Platform Integration

System provides native windows support including NTFS, Permission management, Firewall configuration , and windows service integration.

In simpler terms, Mini DOS seeks to make multiple computers act as one. Users can execute commandslike read or write files, and move processes between devices with minimal manual configuration.

IV. Architecture

A. Architectural Overview of Proposed Prototype

Mini DOS employs layered architecture consisiting of 5 layers: network layer, kernel layer, Filesystem layer , Application layer, CLI Layer. this design follows separation priciple, enabling modularity and independent component development.

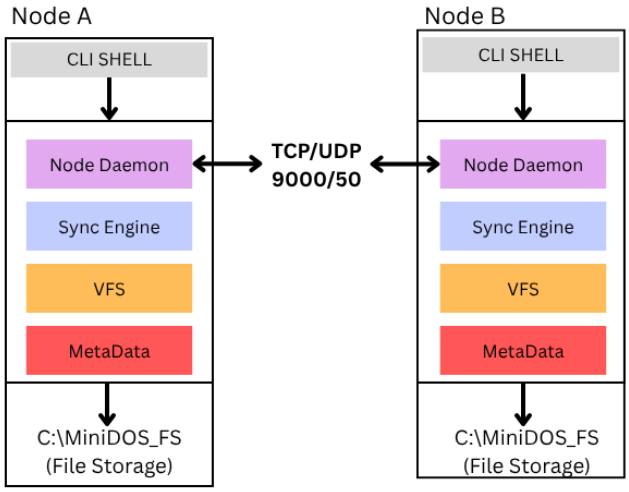

Fig. 1. Peer-to-Peer MiniDOS Architecture

B. Core Components of MiniDOS

Node Daemon: Node daemon serves as main service process on each node, implementing a multi threaded tcp serve (port 9000-9010) for peer communication. Daemon coordinaes all subsystem components including the peer manager, virtual filesystem, sync engine, process monitor. Connection handling employs thread-per-connection model to support concurrent operation across multiple peers.

Peer Manager: Peer manager makes sure to check through UDP heartbeats to see if any peer is active. It broadcasts beats on port 9050 and is broadcasted at every 5 seconds enabling dynamic clustering relationship.

Virtual File system: VFS is virtual unified file system supporting standard operations like : read, write, delte, create, mkdir.VFS implements path validation to prevent directory traversal attack and calculates SHA - 256 checksum for integrity.

Sync Engine: This implements full replication across all nodes using three synchronization strategies: push sync, pull sync and full sync.

Metadata Store: File metadata is persisted in SQLite databases maintaining file versions, checksums, modification timestamps, and sync history. The schema includes tables for file metadata . this enables efficient metadata comparison during synchronization operations.

C. Network Protocol

MiniDOS implement a custom json based protocol for inter-node communication. Message use length-prefixed framing where a byte specifies the subsequent json payload length. This approach enables efficient messages boundary detection without delimiter scanning. command (file operations), sync (replication), heartbeat (health monitoring), discovery (peer announcement), and response (acknowledgments). These 5 message types are defined. Checksums are calculated using SHA-256 hashing for message integrity verification. TCP connections follow a short-lived pattern where clients establish connections, transmit requests, receive responses, and immediately close connections.

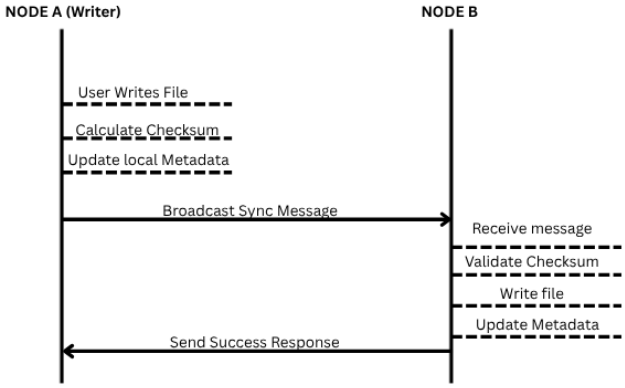

Fig. 2. File Synchronization flow

V. Key Implementation Details

A. File system Operations

File operations are atomic at the individual file level. Write operations first write to temporary files, calculate checksums, then atomically rename to target paths. This approach prevents partial writes during failures.

B. Synchronization Protocol

When files are modified,sync engine calculates checksums and broadcasts sync messages to all active peers. Receiving nodes validate checksums, write files, and update local metadata.

C. Concurrency Control

The implementation uses threading for concurrent connection handling with file-level locking during writes. While this provides basic protection against corruption, it does not prevent conflicting concurrent writes across nodes—these are resolved post-fact using timestamp ordering.

D. Windows Integration

The system checks for Administrator privileges at startup, configures Windows Firewall rules automatically, and can be installed as a Windows service using the service wrapper module. NTFS permissions are enforced at the filesystem layer.

VI. Limitations

A. Scalability Constraints

The full replication model limits scalability to approximately 10 nodes before write amplification and storage overhead become prohibitive. Each write operation must be replicated to N-1 peers, resulting in O(N) network overhead.

B. Platform dependency

Current implementation is Windows-specific, utilizing Windows Specific APIs for permission management, service installation, and firewall configuration. Porting to Linux or macOS would require platform-specific reimplementation of these components.

C. Network Requirement

UDP broadcast-based discovery limits deployment to single subnet environments. Cross-subnet or WAN deployments would require alternative discovery mechanisms such as multicast, gossip protocols, or static configuration files.

VII. Conclusion and Future work

MiniDOS demonstrates that a lightweight distributed operating layer—featuring unified storage, automatic replication, and peer-to-peer coordination—can be built atop existing operating systems while retaining simplicity and strong educational value. Its fully replicated filesystem, timestamp-based conflict handling, and Python-driven implementation collectively validate the system's design goals of decentralization, transparency, and fault tolerance. Performance measurements on gigabit LANs further confirm suitability for small clusters, achieving interactive latencies for both file and command operations.

Looking ahead, several enhancements can extend MiniDOS beyond its current scope. Scalability can be improved by adopting partial replication through consistent hashing or DHTs, while security can be strengthened with TLS, mutual authentication, and cryptographic integrity checks. Incorporating CRDTs, vector clocks, and advanced synchronization methods such as rsync-style deltas would refine consistency and performance. Cross-platform support through unified abstractions and FUSE integration would broaden system applicability. Finally, features like distributed locking, snapshotting, monitoring dashboards, and cloud-based archival would evolve MiniDOS into a more robust and versatile distributed platform.

References

[1] C. Deepika, L. Anjali, Y. Sandeep, "Distributed Operating System: An Overview," *IJRASET*, vol. 2, no. 4, pp.115–119, 2014.

[2] A. S. Tanenbaum, "Distributed operating systems anno 1992. What have we learned so far?," *Distributed Systems Engineering*, Sept. 1993.

[3] A. S. Tanenbaum et al., "The Amoeba distributed operating system," *Communications of the ACM*, vol. 33, no. 12, pp. 46–63, 1990.

[4] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, "Plan 9 from Bell Labs," Computing Systems, vol. 8, no. 3, pp. 221-254, Summer 1995.