REPORT

The Raft3D project is a distributed system designed to manage 3D printing resources (printers, filaments, and print jobs) using a simplified mock Raft consensus algorithm. Implemented as a Dockerized cluster with three nodes (raft3d-node1, raft3d-node2, raft3d-node3), it demonstrates fault tolerance, leader election, RESTful API functionality, and monitoring through Prometheus metrics. Below is a comprehensive description of what was implemented and developed, focusing on the system's features, components, and capabilities.

Project Overview

Purpose: Raft3D provides a fault-tolerant system for managing 3D printing resources in a distributed environment. It uses a mock Raft algorithm (MockRaft) to ensure a single leader coordinates operations, exposes APIs for resource management, and provides metrics for monitoring system health.

Key Objectives:

1. Deploy a three-node cluster using Docker Compose.
2. Implement leader election to maintain a single leader at a time.
3. Develop RESTful APIs for managing printers, filaments, and print jobs.
4. Enable fault tolerance by electing a new leader when a node fails.
5. Expose Prometheus metrics to monitor leader status, requests, and snapshots.
6. Support periodic state snapshots for persistence.

Core Components:

- **Dockerized Cluster**: Three nodes running as Docker containers with shared storage.
- **Mock Raft Algorithm**: Simplified Raft implementation for leader election using a shared file.
- **FastAPI Application**: RESTful APIs for managing printing resources.
- **Prometheus Metrics**: Metrics for leader status, HTTP requests, and snapshots.

- **Snapshot Manager**: Periodic state snapshots stored in a shared directory.

Implemented Features and Components

1. Dockerized Cluster

- **Docker Compose Configuration**:
  - Defined three services (raft3d-node1, raft3d-node2, raft3d-node3) in docker-compose.yml.
  - Configured environment variables: NODE_ID (node identifier), RAFT_PORT (Raft communication), HTTP_PORT (API endpoint), RAFT_DIR (data directory), and CLUSTER (peer list).
  - Mapped external ports 8080-8082 to internal HTTP port 8080 and 9090-9092 to internal Raft port 9090.
  - Used a single shared volume (raft-data) mapped to /raft/data for all nodes to store leader.txt and snapshots.
- **Deployment**:
  - Built and started the cluster with sudo docker-compose up --build -d.
  - Verified container status with sudo docker ps, confirming container names (e.g., raft3d_raft3d-node1_1) and port mappings.
- **Capabilities**:
  - Runs three independent nodes communicating over a bridge network (raft3d-net).
  - Ensures shared storage for consistent leader coordination and state persistence.

2. Mock Raft Implementation

- **MockRaft Class** (app/raft/mock_raft.py):
  - Implemented a simplified Raft algorithm to manage leader election and state application.
  - Used a shared leader.txt file in /raft/data to track the current leader’s ID.
  - Provided methods:
    - start: Initiates a background thread for leader election.
    - stop: Stops the node and resigns leadership if applicable.

        - become_leader: Writes the node's ID to leader.txt after verifying no other leader exists.
        - resign_leader: Removes leader.txt if the node is the leader and updates metrics.
        - run: Periodically checks for a leader, removes stale leader.txt (>10 seconds old), and attempts election with 50% probability.
        - apply: Applies log entries to a finite state machine (FSM) for state updates.
    - Integrated with app/raft/store.py (RaftNode) to manage state transitions.
- **Leader Election**:
    - Nodes check leader.txt to detect the current leader.
    - If no leader exists or leader.txt is stale, nodes attempt election by writing their ID to leader.txt.
    - Elections occur with a 50% probability to balance frequency and stability.
- **Fault Tolerance**:
    - Detects stale leader.txt files (older than 10 seconds) to recover from leader crashes.
    - Ensures only one leader exists at a time through shared file coordination.
- **Capabilities**:
    - Maintains a single leader to coordinate operations.
    - Supports recovery from node failures by electing a new leader.
    - Applies log entries to ensure consistent state updates.

3. RESTful APIs

- **FastAPI Application** (app/main.py, app/api/handlers.py):
    - Developed a FastAPI server for each node to handle API requests.
    - Implemented endpoints under /api/v1:
        - **Printers**:
            - POST /printers: Create a printer (e.g., {"id": "p1", "company": "Creality", "model": "Ender 3"}).
            - GET /printers: Retrieve all printers.
        - **Filaments**:
            - POST /filaments: Create a filament (e.g., {"id": "f1", "type": "PLA", "color": "Blue", "total_weight_in_grams": 1000}).

        - GET /filaments: Retrieve all filaments.
    - **Print Jobs**:
        - POST /print_jobs: Create a print job (e.g., {"id": "j1", "printer_id": "p1", "filament_id": "f1", "filepath": "prints/sword/hilt.gcode"}).
        - POST /print_jobs/{id}/status: Update print job status (e.g., Queued, Running, Done).
  - Integrated with RaftNode to apply API requests to the Raft log for consistency.
- **Capabilities**:
  - Enables creation, retrieval, and updating of 3D printing resources.
  - Supports state updates (e.g., filament weight reduction after print jobs).
  - Operates on any node, with the leader handling state changes.

4. Prometheus Metrics

- **Metrics Setup** (app/monitoring/metrics.py):
  - Defined Prometheus metrics:
    - raft3d_is_leader: Gauge indicating leader status (1 for leader, 0 for follower) per node.
    - raft3d_requests_total: Counter for HTTP requests, labeled by endpoint and method.
    - raft3d_snapshots_total: Counter for snapshots taken.
    - Python process metrics (e.g., process_cpu_seconds_total, process_virtual_memory_bytes).
  - Exposed metrics via /metrics endpoint in FastAPI.
- **Capabilities**:
  - Monitors leader status across nodes.
  - Tracks HTTP request volume and snapshot activity.
  - Provides system-level insights (CPU, memory, file descriptors) for performance monitoring.

5. Fault Tolerance

- **Implementation**:
  - Designed the system to handle node failures by electing a new leader.
  - Used stale leader.txt detection to identify crashed leaders and trigger elections.
  - Provided commands to test fault tolerance:

    - Stop the leader node (e.g., sudo docker stop raft3d_raft3d-node1_1).
    - Verify new leader election via logs and metrics.
    - Perform API operations on a follower node to confirm continued functionality.
    - Restart the stopped node to rejoin the cluster.
- **Capabilities**:
  - Ensures system availability by electing a new leader after a failure.
  - Maintains API functionality through the new leader or followers.
  - Supports node recovery by allowing stopped nodes to rejoin.

6. Snapshots

- **Snapshot Manager**:
  - Implemented a SnapshotManager to periodically save state snapshots in /raft/data.
  - Integrated with raft3d_snapshots_total metric to track snapshot counts.
  - Provided command to verify snapshots:
    bash

    ```bash
    sudo docker exec raft3d_raft3d-node2_1 ls /raft/data
    ```

- **Capabilities**:
  - Persists system state for recovery or inspection.
  - Stores snapshots in the shared raft-data volume for consistency.

The Raft3D project successfully implements a distributed 3D printing management system with a mock Raft algorithm, Dockerized deployment, RESTful APIs, Prometheus metrics, fault tolerance, and snapshot persistence. The system supports leader election, resource management, and monitoring, with a complete set of commands to demonstrate its functionality. It serves as a practical example of distributed systems concepts, with robust features for operation, recovery, and debugging, ready to showcase fault-tolerant behavior and API-driven resource management.

Demonstration Commands

The following commands were developed to showcase the system's functionality:

1. Cluster Setup:

```
sudo docker-compose up --build -d

sudo docker ps

sudo docker-compose logs | grep "elected leader"
```

2. Metrics Verification:

```
curl http://localhost:8080/metrics | grep raft3d_is_leader

curl http://localhost:8081/metrics | grep raft3d_is_leader

curl http://localhost:8082/metrics | grep raft3d_is_leader
```

3. API Interactions:

```
curl -X POST http://localhost:8080/api/v1/printers -H "Content-Type: application/json" -d '{"id": "p1", "company": "Creality", "model": "Ender 3"}'

curl -X POST http://localhost:8080/api/v1/filaments -H "Content-Type: application/json" -d '{"id": "f1", "type": "PLA", "color": "Blue", "total_weight_in_grams": 1000, "remaining_weight_in_grams": 1000}'

curl -X POST http://localhost:8080/api/v1/print_jobs -H "Content-Type: application/json" -d '{"id": "j1", "printer_id": "p1", "filament_id": "f1", "filepath": "prints/sword/hilt.gcode", "print_weight_in_grams": 100, "status": "Queued"}'

curl -X POST http://localhost:8080/api/v1/print_jobs/j1/status?status=Running

curl -X POST http://localhost:8080/api/v1/print_jobs/j1/status?status=Done
```

curl http://localhost:8080/api/v1/filaments

4. **Fault Tolerance**:

Check Current Leader:

curl http://localhost:8080/metrics | grep raft3d_is_leader

curl http://localhost:8081/metrics | grep raft3d_is_leader

curl http://localhost:8082/metrics | grep raft3d_is_leader

Stop Leader (assume node1 is leader, check node1 docker container id):

sudo docker stop raft3d_raft3d-node1_1

Verify Leader Election:

curl http://localhost:8081/metrics | grep raft3d_is_leader

curl http://localhost:8082/metrics | grep raft3d_is_leader

5. Snapshots:

sudo docker exec raft3d_raft3d-node2_1 ls /raft/data

Capabilities Summary

The Raft3D system provides:

- **Distributed Cluster**: Three-node setup with shared storage for consistent state management.
- **Leader Election**: Mock Raft ensures a single leader using leader.txt coordination.
- **Fault Tolerance**: Recovers from node failures by electing a new leader and continuing operations.

- **RESTful APIs**: Manages 3D printing resources with create, retrieve, and update operations.
- **Monitoring**: Exposes Prometheus metrics for leader status, requests, snapshots, and system health.
- **Persistence**: Saves state snapshots for recovery and inspection.