A DISTRIBUTED SYSTEMS CLUSTER SIMULATION

PES2UG22CS180

PES2UG22CS163

PES2UG22CS183

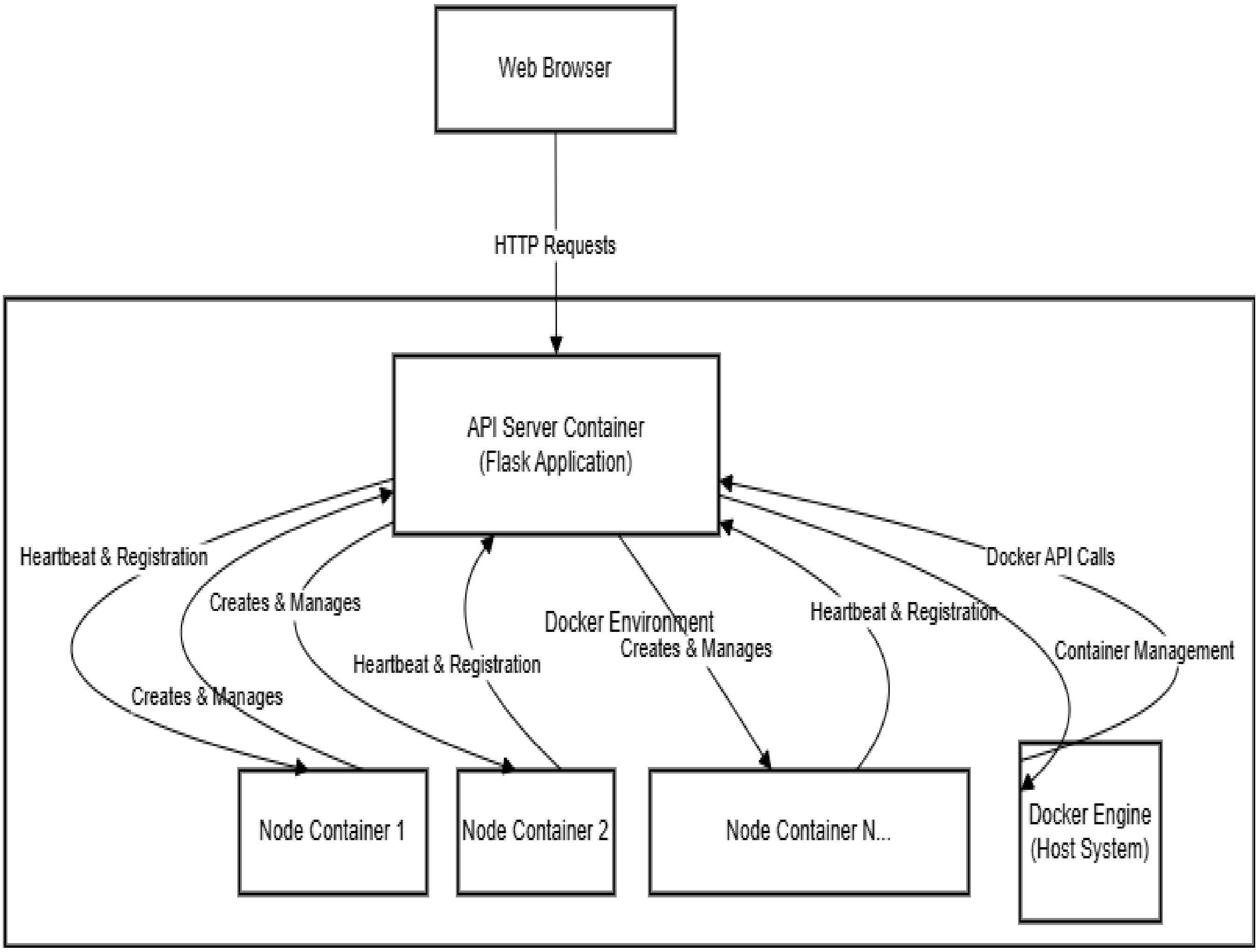
PES2UG22CS167

Architecture (Technical Stack)

System Overview

The Distributed Cluster Simulator is a containerized application that mimics the behavior of a distributed computing cluster with nodes and pods, similar to a simplified Kubernetes-like orchestration system. The system demonstrates concepts of resource scheduling, container orchestration, and health monitoring.

Architecture Diagram



Technical Stack

A) **Backend Services:**

- **Python 3.9**: Primary programming language
- **Flask 2.3.2**: Web framework for the API server
- **Docker 6.1.3 SDK**: For container management and orchestration
- **Requests 2.31.0**: HTTP client for node-to-server communication

B) **Containerization:**

- **Docker**: Container runtime
- **Docker Compose**: Multi-container application definition and deployment

C) **Network Infrastructure:**

- Custom Docker bridge network ("cluster_network") for container communication

D) **Frontend:**

- **HTML/CSS/JavaScript**: Browser-based UI
- **Fetch API**: For asynchronous client-server communication
- **JSON**: Data exchange format

Core Components

1. **API Server (app.py):**
    - Flask application that serves as the central control plane
    - Manages node and pod lifecycle
    - Implements scheduling logic for pods
    - Monitors node health through heartbeats
    - Provides REST API endpoints for system management
    - Serves the web-based user interface
2. **Node Services (node.py):**
    - Simulates compute nodes in the cluster
    - Registers with the API server on startup
    - Sends periodic heartbeats to signal health
    - Reports available compute resources (CPU cores)
3. **Web Interface (index.html, style.css):**

- User-friendly dashboard for cluster management
- Provides forms for adding nodes and launching pods
- Displays real-time status of nodes and their allocated pods
- Auto-refreshes to reflect the current system state

Data Flow

1. User interacts with the web interface to add nodes or launch pods
2. Browser sends HTTP requests to the API server
3. API server processes requests:
   - For node creation: launches a new Docker container running the node service
   - For pod deployment: schedules pod on an available node using first-fit algorithm
4. Nodes register with the API server on startup
5. Nodes send periodic heartbeats to maintain health status
6. API server monitors node health and reschedules pods if nodes fail
7. Web UI periodically polls the API server for cluster status updates

Health Monitoring Mechanism

The system implements a health monitoring thread that:

- Continuously checks the last heartbeat time of each node
- Marks nodes as failed if no heartbeat is received within 10 seconds
- Reschedules pods from failed nodes to healthy nodes when possible
- Cleans up failed node containers from the Docker environment

Steps for Testing and Validation

Prerequisites Verification

1. **Docker Installation Check:**
   - Verify Docker Desktop is properly installed on Windows
   - Ensure Docker service is running
   - Confirm Docker commands work in PowerShell or Command Prompt
2. **Port Availability Check:**
   - Verify port 5000 is not already in use on the host system

- Use netstat -ano | findstr :5000 to check port status

Project Setup

1. **Project Structure Creation:**
   - Create project directory
   - Create all required files with proper content:
     - app.py
     - node.py
     - docker-compose.yml
     - Dockerfile.app
     - Dockerfile.node
     - requirements.txt
     - templates/index.html
     - static/style.css
2. **Docker Network Check:**
   - Run docker network ls to verify no conflicts with existing networks
   - Review network configuration in docker-compose.yml

Build and Deployment Testing

1. **Node Image Build:**
   - Run docker build -t cluster-node -f Dockerfile.node .
   - Verify build completes without errors
   - Confirm image exists with docker images | findstr cluster-node
2. **Application Deployment:**
   - Run docker-compose up --build
   - Verify logs show successful startup
   - Confirm API server messages about listening on port 5000
   - Check for network creation messages
3. **Container Status Verification:**
   - In a separate terminal, run docker ps
   - Verify API server container is running

- Note container names and IDs for later reference

Functional Testing

1. **Web Interface Accessibility:**
    - Open web browser and navigate to http://localhost:5000
    - Verify the Distributed Cluster Simulator interface loads
    - Confirm all UI elements are properly rendered
2. **Node Creation Testing:**
    - Enter a value (e.g., 4) in the "CPU Cores" field
    - Click "Add Node" button
    - Verify success message appears
    - Check node appears in Node Status section
    - Run docker ps to verify a new container was created
3. **Pod Deployment Testing:**
    - Enter a value (e.g., 2) in the "CPU Required" field
    - Click "Launch Pod" button
    - Verify success message appears
    - Check pod is assigned to a node in Node Status section
    - Verify available cores on the node decreased by the requested amount
4. **Resource Limit Testing:**
    - Try to deploy a pod requesting more CPU than available on any node
    - Verify appropriate error message is displayed
    - Confirm no pod is created
5. **Health Monitoring Testing:**
    - Identify a node container ID from the Node Status section
    - Manually stop a node container using docker stop <container_id>
    - Wait 10-15 seconds
    - Verify node disappears from the Node Status list
    - Check if pods from the failed node are rescheduled to other nodes

Performance and Reliability Testing

1. **Scale Testing:**
   - Add multiple nodes with varying CPU cores (e.g., 2, 4, 8)
   - Deploy multiple pods with different CPU requirements
   - Verify scheduling works correctly across all nodes
   - Check system remains responsive
2. **Stress Testing:**
   - Rapidly add and remove nodes
   - Launch pods with boundary values of CPU requirements
   - Monitor Docker statistics with docker stats
   - Check for memory leaks or performance degradation
3. **Network Resilience:**
   - Test system response when network is temporarily disrupted
   - Verify heartbeat mechanism correctly identifies node failures
   - Confirm system recovers when network connectivity is restored

Cleanup and Shutdown

1. **Application Termination:**
   - Press Ctrl+C in terminal running docker-compose
   - Alternatively, run docker-compose down in project directory
   - Verify all containers are stopped
2. **Resource Cleanup:**
   - Run docker ps -a to check for any remaining containers
   - Remove any leftover containers with docker rm <container_id>
   - Run docker images to check for created images
   - Remove test images if desired with docker rmi cluster-node