Docker-based Network Monitoring System (Mini NOC)

1. Introduction

This project implements a lightweight network monitoring system using Docker containerization. The application simulates a simplified **Network Operations Center (NOC)**, where network availability and latency of selected targets are continuously monitored and displayed through a web interface. The system is designed using a multi-container architecture, where each service runs in an isolated Docker container and communicates with other services through a Docker-managed network. The deployment is fully automated using Docker Compose and shell scripts, ensuring reproducibility and ease of use.

2. Requirements

To deploy and run this application, the following software must be installed on a Linux system:

- Docker
- Docker Compose (Docker CLI plugin)
- Bash shell

The project was developed and tested on a Fedora Linux environment, but it is compatible with any modern Linux distribution with Docker support.

3. Application Description

The application continuously monitors network targets such as public IP addresses by sending ICMP ping requests and collecting latency and availability data. This data is processed and stored, and then presented in a web-based dashboard accessible through a browser.

The monitoring process is automatic and runs in the background. Every few seconds, new measurements are taken and added to the system, allowing the user to observe real-time network behavior. The dashboard displays the target address, measured latency, status (UP or DOWN), and timestamp of each measurement.

4. System Architecture

The application is based on a multi-container architecture consisting of three main services:

- A probe service that performs network checks
- A backend service that processes and stores data
- A frontend service that displays the data

Each service runs in its own Docker container and is built from a custom Docker image. These containers are connected through a shared Docker network, allowing seamless communication between them.

5. Docker Networking

The application uses a custom Docker bridge network to enable communication between containers. Each container is assigned a virtual network interface and can communicate with other containers using their service names as hostnames.

For example, the frontend container communicates with the backend service using the hostname backend, rather than using an IP address or localhost. This is possible because Docker provides an internal DNS service that dynamically resolves container names to their respective IP addresses.

This approach ensures flexibility and avoids hardcoding network configurations, making the system more portable and scalable.

6. Persistent Storage

To ensure data persistence, the application uses a named Docker volume. The backend service stores monitoring data in a SQLite database file located inside the container, which is mapped to the Docker volume.

This means that even if the backend container is stopped or removed, the data remains intact in the volume. This is important because containers are ephemeral by default, and without volumes, all stored data would be lost when the container is recreated.

7. Container Configuration

Each service is defined and configured in the docker-compose.yml file. The configuration includes:

- Building images from Dockerfiles
- Connecting services to a shared network
- Mounting volumes for persistent storage
- Defining dependencies between services
- Exposing ports for external access

The frontend service exposes its internal port to the host system, allowing users to access the application through a web browser. Other services communicate internally and do not need to expose ports to the host.

8. List of Containers

The application consists of the following containers:

- **frontend**: Provides the web-based dashboard and retrieves monitoring data from the backend service.
- **backend**: Handles incoming data from the probe service and stores it in a persistent database.
- **probe**: Continuously performs network checks and sends results to the backend service.

Each container runs independently but works together as part of the overall system.

9. Application Lifecycle Management

The application is managed using four shell scripts:

prepare-app.sh

This script prepares the environment by creating the required Docker network and volume and building all container images.

start-app.sh

This script starts all services using Docker Compose in detached mode. After starting, it prints the URL where the application can be accessed.

stop-app.sh

This script stops all running containers without deleting the associated volumes, ensuring that stored data is preserved.

remove-app.sh

This script removes all containers, networks, and volumes associated with the application, completely cleaning up the environment.

10. How to Run the Application

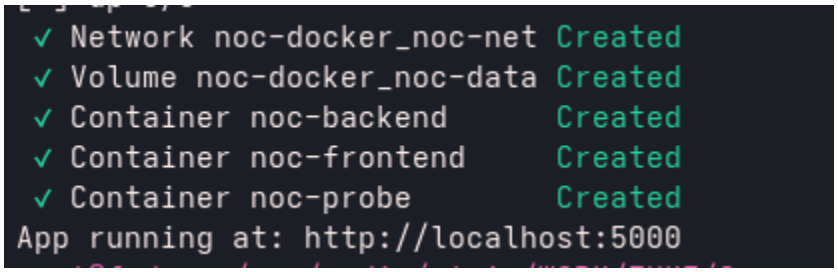
To deploy and run the application, execute the following commands in the project directory:

./prepare-app.sh
./start-app.sh

After starting the application, open a web browser and navigate to:

http://localhost:5000

The dashboard will display real-time monitoring data.



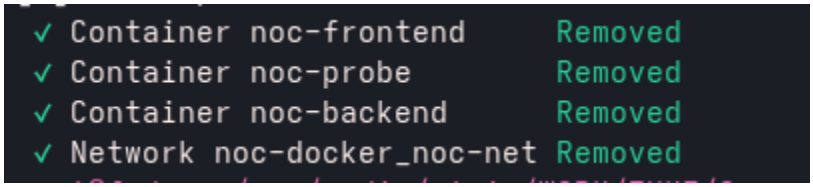
11. Stopping and Removing the Application

To stop the application:

./stop-app.sh

To remove all associated resources:

./remove-app.sh



12. Example Usage

After starting the application, the system begins monitoring predefined network targets automatically. The dashboard updates periodically with new entries showing latency and availability status.

The user can observe how network performance changes over time, making it useful for basic monitoring and analysis.

13. Resources Used

- Docker official documentation
- Docker Compose documentation
- Python Flask documentation

14. Use of Artificial Intelligence

Artificial intelligence tools were used to assist in understanding Docker concepts, designing the system architecture, and generating parts of the implementation and documentation. All outputs were reviewed, modified, and tested to ensure correctness and originality.