Docker for Embedded Modular Software

Research Project Presentation

Presented by: Shawal Ibrahim Khan

Date: September 30, 2024

Supervisor: Dr. Detlef Streitferdt

Keywords: Docker, Embedded Systems, Benchmarking, Real-Time System

Introduction

Background

- Embedded systems are used in a wide range of devices (e.g., appliances, medical machines, industrial automation) and are becoming more complex.

Challenges

- Traditional monolithic architectures are difficult to scale and adapt.
- Embedded systems face resource constraints (CPU load, memory, network bandwidth), which makes performance optimization critical.

Solution

- Docker-based modular software, which enhances scalability and adaptability by isolating tasks within containers.

Research Objectives

Primary Goal

- Create Docker-based modular software for embedded systems to address scalability and efficiency.

Specific Objectives

Modular Docker Framework: Develop independent modules for tasks like CPU, memory, and network performance measurements, ensuring modularity and reusability.

- Benchmarking Tools: Evaluate system performance with key metrics (CPU load, memory usage, network throughput, etc.).
- Adaptability: Ensure the software framework can be deployed across different hardware platforms (e.g., ARM cores A53, A72).
- Parameterization:Enable dynamic CPU, memory, and network configuration for different test cases.

Scope of the Project

In Scope

- Design, implement, and test Docker containers for embedded systems, dividing tasks into separate modules to improve scalability.Develop tools to benchmark the system’s performance (CPU, memory, network performance, latency, etc.).

Out of Scope

- No hardware modification or development.Focus is on benchmarking performance rather than addressing security in depth.

Literature Review

Docker in Embedded Systems

- Docker provides lightweight containers for running applications in isolated environments. It was traditionally used in cloud and large data centers but is now applicable to embedded systems.

Benchmarking Techniques

- Stress-ng: A tool to simulate CPU and memory stress, crucial for testing system resilience under load.
- Iperf: Measures network throughput, a critical performance factor in embedded systems.

Modular Software Architecture

- Modular approaches replace monolithic architectures, allowing flexibility, maintainability, and scalability in embedded systems. Independent modules can operate without affecting the entire system.

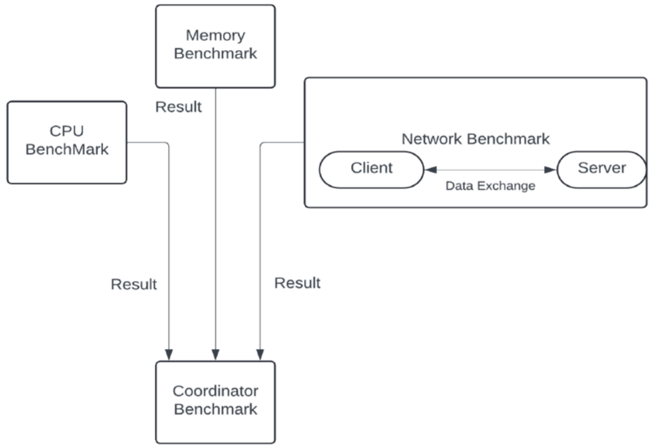
System Design

Modular Architecture:

- The system is divided into Docker containers, each handling a specific task like CPU, memory, network and Coordinator benchmarking.
- Communication between containers is handled using TCP/IP network protocols to maintain isolation and consistency.Containers are deployed on ARM-based hardware using Debian or Yocto OS.

Components

1. CPU Benchmarking: Measures processing performance.
2. Memory Benchmarking: Tests memory usage.
3. Network Benchmarking: Measures data transfer efficiency between client and server.
4. Coordinator Benchmark: aggregates results from the performance modules.



Implementation

Developed Docker Modules

1.Monitoring Module: Monitors system performance (CPU usage, memory load, network bandwidth) without affecting resource optimization.

2.Load Generation Module: Tests the system under stress (CPU, memory, and network stress using tools like stress-ng and Iperf).

3.Benchmarking Module: Aggregates results from other modules and communicates with them via HTTP requests (implemented using Python scripts).

Integration

- Containers operate independently but communicate through Docker Compose. Docker volumes ensure data integrity and synchronization.
- A health check system monitors the status of all containers, ensuring they complete tasks correctly.

```
benchmarking-software/
│
├── cpu-benchmark/
│   ├── Dockerfile
│   └── benchmark.sh
│
├── memory-benchmark/
│   ├── Dockerfile
│   └── benchmark.sh
│
├── network-benchmark/
│   ├── Dockerfile
│   ├── server.sh
│   └── client.sh
│
├── coordinator/
│   ├── Dockerfile
│   ├── coordinator.py
│   └── requirements.txt
│
├── docker-compose.yml
├── README.md
└── .env
```

Benchmarking and Metrics Collection

Benchmarking Toolkit

- Custom benchmarks are used to evaluate the performance of the embedded system.A benchmark table collects results for CPU load, memory usage, network throughput, and more.

Metrics Collected

- CPU Load: Measures CPU performance under varying loads (using stress-ng).
- Memory Utilization: Tracks memory usage over time, particularly important in embedded systems with limited memory.
- Network Performance: Measures data transfer performance using Iperf, particularly useful for identifying network latency.
- Latency: Measures communication time between containers, crucial for real-time applications.

Benchmark Table

Test Case: 1

Types of Benchmark	Resources Allocation	Duration (s)	Execution Time(s)	Inter-container Communiaction Time (ms)	Memory Usage Over Time	Latency	Notes
CPU Benchmark	50% (CPU Load)	60s	61.19s	25ms	N/A	N/A	CPU handled load efficiently
Memory Benchmark	2048 (Memory Usage in MB)	60s	60.02s	42ms	Stable Usage	N/A	Memory remained stable throughout the test
Network Benchmark	10Mbps (Network Throughput)	60s	60.04	73ms	N/A	0.63ms	Network performance was consisten

Evaluation

System Evaluation

- The Docker-based modular architecture successfully meets scalability and adaptability objectives.
- The system efficiently handles stress tests for CPU, memory, and network performance under moderate loads.

Potential Improvements

- Performance Optimization: Refine inter-container communication to reduce latency and improve resource management.
- Enhanced Benchmarking Tools: Add more benchmarking metrics for deeper analysis.
- Security Enhancements: Strengthen security for inter-container communication and conduct regular security audits.

Challenges and Limitations

Challenges

- Latency Issues: Under heavy loads, communication times between containers increase, affecting real-time performance.
- Inter-Container Communication: Optimizing data sharing and communication between containers is necessary for better efficiency.

Opportunities

- There's potential for system optimization through refined communication code and better resource handling.

Conclusion

Achievements

- Successfully developed Docker-based modular software for embedded systems.
- Achieved key objectives of scalability, modularity, and adaptability.T
- he system effectively demonstrates the use of Docker containers for real-time performance measurement in embedded environments.

Future Work

- Further integration with emerging technologies such as AI, IoT, and edge computing.
- Expand research into optimizing Docker containers for real-time embedded applications.

Thank You!