Technical University of Ilmenau
Faculty of Computer Science and Automation
Institute for Technical Informatics and Engineering Informatics
Department of Software Architectures and Product Lines

Program: Master of Research in Computer and System Engineering

- **Title of project:** Docker for Embedded Modular Software
- **Author:** Shawal Ibrahim Khan
- **Date:** 01.09.2024
- **Supervisor:** Dr. Detlef Streitferdt
- **Keywords:** Docker, Embedded Systems, Benchmarking, Modular Software, Real-Time Systems

Abstract

Embedded systems are more complex nowadays which need more flexible, scalable software architectures and it is possible with modular software architectures. In the past, the monolithic approach was used in embedded systems which shows the problem when we need to be scalable and adaptable and these things are very important in this era of embedded systems. To keep this thing in mind, our research is exploring the Docker-based modular software for embedded systems. It shows that modular-based dockerized software is easy to scalable and flexible to enhance the performance of the system. It can maintain consistency in different hardware environments.

The purpose of this project is to make a benchmark software for embedded systems. This benchmark software is divided into modules and each module has a separate docker container to perform the task in an isolated environment. To check the performance of the system, we make a benchmark table that has the results of all modules. In this benchmark, we have metrics like CPU load, Memory load, network throughput, execution time, latency, and inter-container communication time.

The results show the importance of this docker-based modular software in meeting the maintainability and scalability of embedded systems. The research also points out the areas where improvement is needed such as optimization, enhance the benchmark tool, and security. More research in the future will explore the new gateways for integrating this framework with new technologies e.g. Artificial Intelligence, IoT, and edge computing to increase the applicability.

Table of Contents

1- Introduction

1.1- Background and Motivation

Background

Embedded systems are special computer systems that perform specific tasks or functions in electric and mechanical machines. These systems are used in a large number of devices like household appliances, medical machinery, and industrial or automotive machinery.

The use of embedded systems in the modern era is becoming more complex, so these systems should be designed by following specific architecture that is easy to maintain and does not need much effort to make it scalable. In the past, the architecture followed for embedded systems was monolithic architecture and it was not easy to scale it and add new requirements to the system.

Challenges

The performance of embedded systems is challenging because these systems use limited or defined specific resources e.g. CPU load, memory throughput, and network bandwidth. Making sure to use these resources or constraints to perform ideally is a big challenge.

Scalability

Embedded systems are growing functionalities in the modern era, so it is also very important to make them scalable and integrate modern features ea. On the other hand, traditionally embedded systems had less scalability because of their monolithic architecture and tightly coupled modules.

Purpose of Using Docker

- **Isolation:** By using docker containers, it is easy to manage the dependencies and different modules can work separately without any interference with other modules. It is beneficial for maintaining and scalability of embedded systems.

- **Consistency Across Environments:** Docker ensures that the software has to be run in the same way on other different environments. As embedded systems have different types of hardware, maintaining consistency after deploying is a very important feature of the docker.

- **Modularity:** For making scalable and maintainable software, it is important to use docker. Because docker provides a modular approach. Every module has to perform a particular task in isolation without affecting other modules. This is the recommendable approach for embedded systems.

- **Scalability:** Running many containers on one device or distributing containers on other hardware makes it more scalable.

1.2- Research Objectives

Primary Goal

The purpose of this research is to create a Docker-based modular software for the embedded system. By using docker containers technology, the software can address the scalability and efficiency of the embedded systems. It is modular based which makes it also easy to maintainable.

Specific Objectives

- Establish a Modular Docker-Based Framework: Designing and implementing many containers and each container can perform a particular task which belongs to the measurement of the system performance. This is also can be reusable to different other embedded environments.

- Make Comprehensive Benchmarking Tools: Evaluate the performance of the system by providing metrics like CPU load, network load, inter-container communication time, latency, execution time, and memory usage. It helps to get the performance of the system under different circumstances.

- Ensure Adaptability and Scalability: This framework can be deployed on different ARM cores e.g. A53 or A72. It shows that the system can work on different hardware.

- Implement Customization through Parameterization: For each container, there is a dynamic configuration for the CPU, memory, and network. With this dynamic functionality, it can test for different test cases.

1.3- Scope of the Project

In Scope

- The project will design, implement, and test the docker containers which are specifically designed for the embedded system. It has multiple containers and each container has to do a particular task and contribute to the system [1].

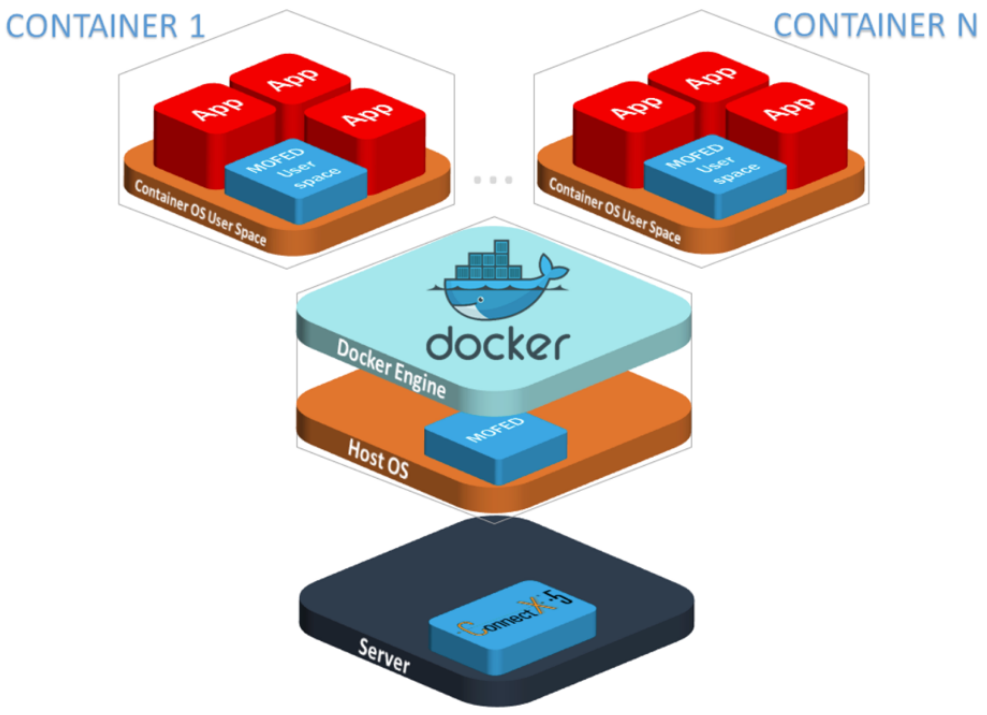


Figure: (i) Deployment of Docker containers on embedded hardware. Source [2]

- The prominent part of the project is to make a tool for benchmarking the performance of the system and to make an analysis of the data that we collected as a result.

Out of Scope

- The project does not involve hardware-related modification or development. The main purpose will be to make benchmark software and to deploy it on provided hardware.

- This research will not go into the depth of security-related aspects because docker is providing an isolated environment. The main focus is to measure the performance of the embedded system.

2- Literature Review

2.1- Docker in Embedded Systems

Introduction

Overview: Docker is introduced as a containerized technology [3]. Docker is responsible for providing lightweight containers; these containers have to run applications in isolated and different environments. The docker was used before only in the cloud and large-scale data centers and now it is also useable in the embedded systems.

Existing Literature

- **Application in embedded systems:** It provides a summary of the studies that investigated the technology of docker in embedded systems. And also explains that before the modular-based approach in the docker, it used the monolithic approach.

- **Isolation and consistency:** The literature review discussed that docker ensures consistency in different environments because it isolates the processes. It is a very important aspect of docker according to the embedded system because changing the hardware can cause the failure of the application but with docker, this problem has been resolved.

- **Ease of Deployment:** Deployment is very easy by using docker in embedded systems. It encapsulates all dependencies and as a result, it takes less time for setup and minimizes the chances of errors.

2.2- Benchamarking Techniques

Importance of benchmarking

- The purpose of benchmarking is to get the performance of the system under different conditions, which is a necessary thing to do to check the efficiency and reliability of embedded systems.

- These performances are measured in the real-time scenario, benchmarking is reporting the performance of the system. It helps to identify any bottlenecks and performance issues.

Existing Techniques

- **Performance Metrics:** Performance metrics are usually used to benchmark embedded systems [4].
    - CPU Load: It is very important to measure the CPU load, it provides the processing power and any potential overload information.
    - Memory Utilization: It provides information on memory usage. In embedded systems, memory has limitations.
    - Network Throughput: It is also an important metric in embedded systems, specifically it affects the overall performance because the transmission of data can impact the performance.
    - Latency: It is an essential metric, specifically in real-time systems if there is a delay then it can fail the system.
- **Tools and Methods:**
    - Stress-ng: It is a very powerful tool. It is used for benchmarking and testing the stress in embedded systems. By using this, it is possible to identify the issues in the early stages. It can be used in the development phase and testing phase.
    - Iperf: It is used to measure the performance of the network. In embedded systems, the network has a significant role. It helps to assess the system's stability in particular scenarios where the stability of the system is a very important aspect of the system.

2.3- Modular Software Architecture

Modularity in Embedded Systems

- It is an approach where multiple modules work independently and these modules can work simultaneously and produce a combined result. In the past, the approach was monolithic architecture where modules were connected tightly, reducing the system's flexibility.

- Modularity is essential in embedded systems because resources and real-time requirements are necessary. In the past monolithic architectures were used that are not easily scalable and maintainable and nowadays microservice architecture is very suitable to update, scale up, and replace a particular module in the embedded system without changing the whole architecture. [5].

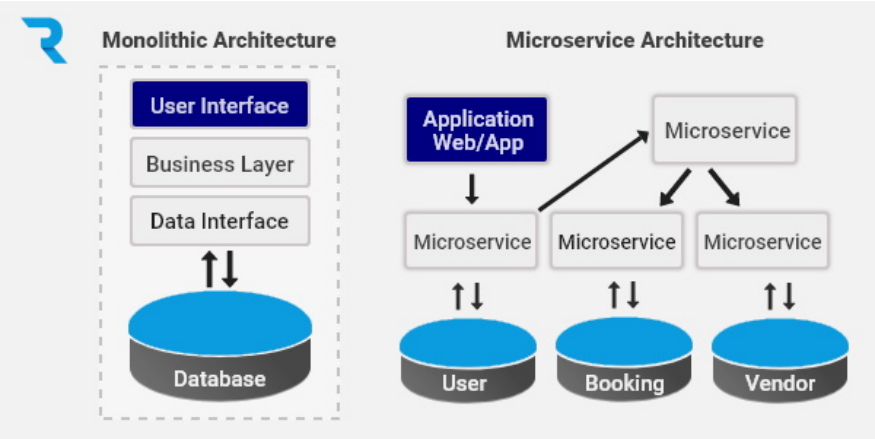


Figure: (ii) Comparison of monolithic and microservice architecture. Source: [5]

Advantages of Modularity

- **Maintainability:** To choose the modular architecture is essential to enhance the ability of maintainability in the embedded system. Without affecting all modules or the whole system, developers can make changes to a particular module. It is worthwhile in embedded systems because it usually needs to make any update after deployment. Studies show that the modular approach is less complex, reduces the time, and also increases the lifespan of the system.

- **Scalability:** Modular architecture has another advantage of scaling. Over time embedded systems need more features and increased functionalities, in this scenario modular design helps a lot. This approach is worthwhile because making the entire

system again is difficult but adding new modules to the existing system is easy. This is making embedded systems more flexible, where hardware and software can be upgraded gradually.

- **Reusability:** It is another important aspect of modular architecture, this approach is enabling to use the code again in the same project or somewhere else. It helps to reduce the time and cost of the development because to develop all the time new modules need extra resources. It prevents the resources of the company by reusing reliable modules.

3- System Design

3.1- Overview of System Architecture

Introduction

For the embedded systems, it is essential to make an architecture of the system, which has a modular, scalable framework by using containers. The purpose of this architecture is to make an isolated environment for each component or module of the system by making separate containers. It will enhance the capability of the system e.g. maintainability, longevity, and performance as well.

Components of the Architecture

- **Docker Containers:** There are several docker containers and each container has to perform a particular task.

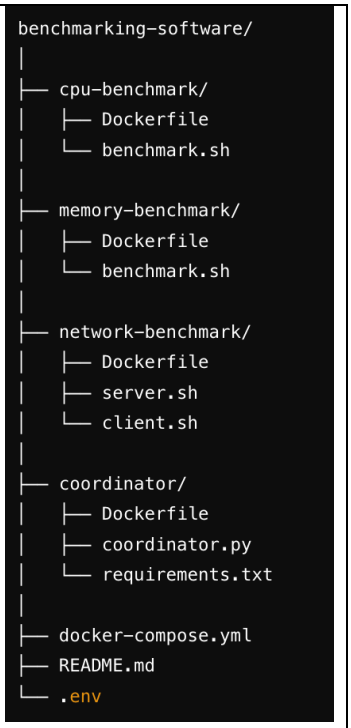
○ **CPU Benchmarking:** It uses the stress-ng and gives the CPU performance. ○ **Memory Benchmark:** It also uses the stress-ng but this time it is used to test the memory. ○ **Network Benchmark:** It has server and client configuration and it uses the iperf3 for measuring the performance of the network. ○ **Coordinate Benchmark:** It receives the result from CPU, network and memory benchmark containers and show them in benchmark table.	

Table: (i)

- **Communication Channels:** The communication of containers using TCP/IP network protocols. It is making ensure that communication should be in an easy way and make sure the isolation of the container will not disturbed.

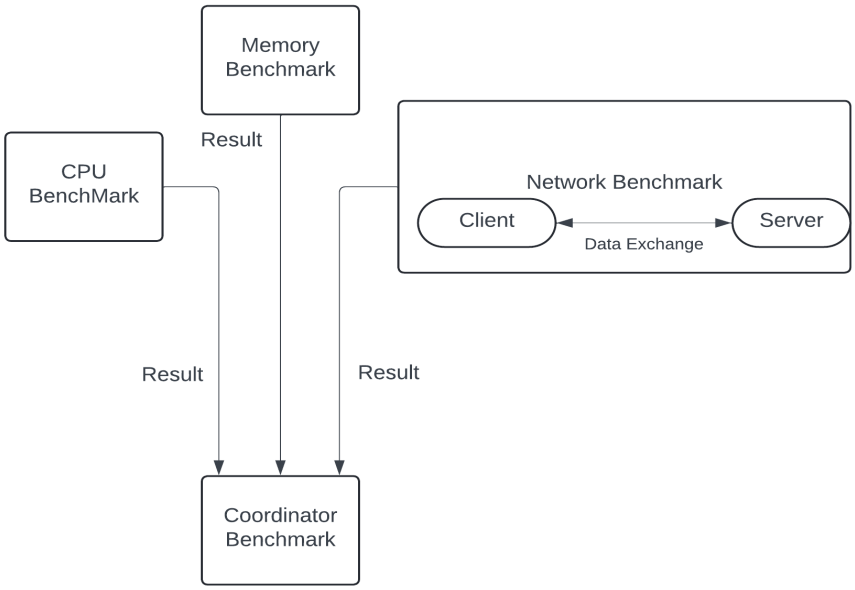
- **Host Environment:** The docker containers run on hardware that has ARM-based architecture. It uses the Debian or Yocto operating systems that are specifically configured for the embedded systems.

3.2- UML Diagram

To visualize the architecture of the system, we usually use the UML diagrams. These diagrams provide a clear view of the system architecture and show the relationship between the components of the system [6].

Component Diagram

Here we used a UML component diagram in which we have CPU, Memory, and Network benchmark docker containers and the coordinator benchmark container is receiving the results from other containers.



Figure(iii)

As the figure shows the interaction between the containers and network benchmark container has configured for the setup of data sharing between server and client that is responsible for checking the network latency.

3.3- Selection of Tools and Technologies

- **Criteria for Selection:** The selection of the technology and tools for the project is an essential part. While the selection of tools and technology is based on the objective or goal of the project [7].

- **Chosen Tools and Technologies:**

    - **Docker:** It can provide an isolated environment for each component and it has portable functionality for other environments. To deploy a docker container is easy which is very important in embedded systems.

    - **Benchmarking Tools:** stress-ng, iperf, and ping were chosen for measuring the CPU, memory, and network performances.

- **Integration and Testing:** To evaluate the functionality of the system, it is necessary to do unit testing and performance testing.

4- Implementation

4.1- Development of Docker Modules

In this, we developed multiple numbers of docker modules, and each docker module is responsible for performing a specific task where each module is isolated because each module has a separate docker container. Each module works separately and can be deployed and tested in an isolated environment. This type of approach is significant because it makes us able to monitor each module and check load generation and data aggregation separately in each module.

Module 1: Monitoring Module

- **Function:** This module is very common and usually discusses this type of system where we have to check or measure the performance of the system. We are using it here to monitor the performance of CPU usage, memory load, and network bandwidth because these things are essential for understanding the behavior of the system.

- **Development Process:** For building this module, it is also important to keep in mind that this module will not affect the optimization of the resources of the system where we are using this module. So, we need a tool which is light in weight like top. By using this module we are enable to get the performance-related information and store this in a log file.

Module 2: Load Generation Module

- **Function:** This module is used to measure loads of the system like CPU stress, memory usage, and networking traffic within the system. This is an essential module because it provides the real-time performance of the system.

- **Development process:** This module is used for testing purposes and supporting tools like stress-ng which is used for CPU load testing and memory usage it uses iperf. We pass the environmental variable, the variable has the value of CPU load and memory usage and we can also pass the network bandwidth value. The main purpose of this type of module is to test the different scenarios of the system with different conditions.

Module 3. Benchmarking Module

- **Function:** This module is also separate and is isolated in a docker container. This module gathers the results of other modules and displays the final results of all modules. We can say that all modules send their final results to this module and this module works like a central module which is communicating with other modules.

- **Development Process:** For the development of this module, we need the data receiving HTTP requests, for this purpose, we use the Python programming language which has the scripts and by using this language we can communicate with other containers by HTTP Post and Get requests. All other containers perform their tasks and send the data as a result to the benchmark container. In our project, we name this module a "coordinator Benchmark".

4.2- Integration of Modules

Introduction

All containers are created separately and work in isolation but we need to combine all containers into a system and make a detailed benchmarking. The purpose is to make communication between containers of different modules.

Inter-Container Communication:

- Networking: Secure communication between containers is a very important thing and networking is set up in this way that it decreases the latency and makes sure the data transfer is reliable between docker containers.

- Data Sharing: Inside the docker, we have docker volumes that are responsible for sharing the data between the docker containers. The purpose of this approach is that when other module will generate their data as results the coordinator benchmark module will access these results. The other very important thing is data integrity and synchronization during the whole process of data sharing.

Orchestration:

- Docker Compose: This is used for managing the containers and deployment of containers. By using a single command whole system can be controlled. The docker-compose command can start and stop the system and it is defined in the docker file as starting the service in sequence and checking or installing the dependencies [8].

- Service Coordination: It is giving the health of other containers working. So, we need to put a health check on whether other containers complete their task or not. For example in our project, all containers are sending their results to the coordinator container so we added the health check on all containers. If any container is not sending the data then the health check will generate a prompt that health is not good. So we can check that specific container and we have to find the problem with that container. So, a health check can tell us which container has the problem.

Testing Integrated System:

- Functional Testing: This is the testing phase, in this we check all functionalities of the components of the system whether all components are working fine or not. It also checks the communication of containers with other containers and checks the variable values and the results are according to whether these values are nearly correct or not.

- Performance Testing: To check the overall system performance we do performance testing. In our case to check that all modules are working efficiently like stress tests. It will help us by giving the idea that our system needs any improvement or optimization.

5- Benchmarking and Metrics Collection

5.1- Benchmarking toolkit

Introduction

To measure the performance of our system, we need a customized benchmarking toolkit. All the results will be shown in that benchmarking toolkit or table. In our case we have a table, this benchmark table has CPU Load, Memory Usage, Network throughput, duration, execution of time, and communication time. Between containers, memory usage over time, and Latency. In this CPU Load, memory usage and network throughput are provided by the user to calculate the duration, execution of time, and communication time. Between containers, memory usage over time, and Latency of the system.

5.2 Metrics Collection

Introduction

In our system, we have customized metrics to analyze the performance of the embedded system. These metrics are very important to understand the final result of the system, these results are quantitative and by analyzing these results, we can figure out whether the system is fulfilling the requirements or it needs further improvements.

Key Metrics Collected

- **CPU Load:** In embedded systems, CPU load is most valuable because in embedded systems as we know we have limited resources and the CPU load metric tells us the usage of CPU by using the stress-ng tool [9]. This tool will tell us the usage of the CPU when we provide the different levels of CPU load or stress. It is very important to understand whether the CPU will work or not when it has the specific load and also we can check that what is the maximum load the system can bear and will work efficiently.

- **Memory Utilization:** This metric shows the memory usage over time, and shows the memory usage when the system is performing different tasks. This metric is crucial in embedded systems because memory resource is usually limited.

- **Network Load:** This metric deals with the networking load, and measures the performance of the network under different load tests. To find the network latency we used the iperf tool which gives us the latency of client-to-server communication in different tests or loads [10].

Latency and Communication Times

This metric shows the communication time between the containers. When a container sends the data to another container then what time our system will take for this communication? It is usually in milliseconds but this is very important to measure this because in the real world, we need the fastest communication. This metric is useful to give us an idea of the ability of the system's responsiveness.

5.3 Initial Results

For checking the performance of the system, we are applying one test case under stress. This test will show the working of the system and tell us that if we have to work on any improvement or not.

- **CPU Load Test:** We apply a specific load on the CPU and as a result, we get that the CPU can handle the specific load easily. So, later we will identify on maximum load of how our system will work. Because, when we put maximum load then it can cause any degradation in the performance.

- **Memory Utilization Test:** The result of memory shows that the system is managing the memory in a good way because we provide a typical load and on this load, it is working smoothly. But as we know when we increase the load on memory then we need optimization to handle this load.

- **Network Performance Test:** When we are sending or receiving data between client and server then the performance of network throughput is adequate for more cases but we can also maintain less latency when we are dealing with the big traffic of data by increasing the capability of the system.

- **Inter-Container Latency Test:** The results show that the communication time between containers is reasonable when we are dealing with the normal stress/load but it increases when the load is heavy.

- Network Latency: The results show the time of the transfer of the data packet from the server to the client and back. It measures the time of transfer in milliseconds.

- Execution Time: This is giving the completion time of the task. In each module, we have to perform a task or a set of tasks, and the time required for that task is execution time.

- Memory Usage: The result shows the memory usage during different tasks execution and performs the system under various loads.

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

Table: (ii)

Test Case: 2

Types of Benchmark	Resources Allocation	Duration (s)	Execution Time(s)	Inter-container Communiaction Time (ms)	Memory Usage Over Time	Latency	Notes
CPU Benchmark	75% (CPU Load)	60s	62.44s	63ms	N/A	N/A	CPU showed a slight increase in execution time under higher load
Memory Benchmark	3072 (Memory Usage in MB)	60s	60.17s	30ms	Slightly Increase	N/A	Memory usage increased slightly
Network Benchmark	20Mbps (Network Throughput)	60s	60.04s	203ms	N/A	1.98ms	Network latency increased with higher throughput

Table: (iii)

6- Evaluation

6.1- Evaluation of the system

This is a very important and critical part to understand that the system we created fulfills all requirements and how well it can meet all objectives for the docker for embedded modular software.

6.3- Potential Improvements

In the future, we can this type of system more user-friendly and efficient by making a few improvements. Here we will discuss those improvements that we feel are challenging when we are working on this project to increase the system capability. Now we are discussing the areas that should be improved.

Performance Optimization

- **Improvement:** To make the system more efficient or optimized by reducing the latency and making sure it uses the resource in the best manner possible. For this, we can work on the inter-container communication code and we can make it more refined, it can help to reduce latency and the system will be more capable.

- **Justification:** If we can reduce the latency and resource efficiency then it will affect the overall behavior of the system and our system will be more efficient in real-time environments.

Enhanced Benchmarking Tools

- **Improvement:** We added metrics in this project that were the requirements of this project but we can add more metrics according to our requirements as our all containers are working in isolation and our application is modular based, so we can add particular new metrics in that particular module and it will not affect the other module.

- **Justification:** By adding more benchmarking metrics, we can get more details or an overview of our system in more depth. It can also be a requirement of any real-world scenario and we can make our system more optimized.

Security improvements

- **Improvements:** security is very essential thing in any system. By regular security audits and more secure communication between containers, we can enhance the security of the system.
- **Justification:** In the real world where data sovereignty is a very important thing. There is a lot of personal data and sensitive data, so good security can make able to the system more trustworthy.

Conclusion of Improvement Suggestions

By applying these improvements, we can make our docker for embedded modular software more efficient, robust, and reliable as well. The current limitations and these limitations by using the improvements would make the system competitive to future challenges in the field of embedded systems.

7- Conclusion and Future Work

7.1- Summary of Findings

The summary tells about the overall success of the project, emphasizing the results and their importance in this project. Here we are telling the overall impact without going into the details because we discussed all the results details in the evaluation part.

The objective of this project was to create a Docker-based modular software for embedded systems. The system is successfully able to achieve all the objectives like our software is modular based and all modules are in isolated docker containers, and these containers can make inter-container communication.

All benchmarking was implemented successfully like execution time, memory usage, latency, and inter-container communication and this benchmarking was a critical part of our project. This framework is working successfully and giving the importance of modularity and scalability in embedded systems. As we discussed before it is also opening the door for further research and development. The results of this research show that it is still expected to make more future innovations.

7.2- Future Research Directions

As we have completed our preliminary goals now we are discussing future aspects of our research and we will discuss how and where can we increase the capability and applicability of embedded systems. In this section, we will explore the area of integration with emerging technologies of embedded systems.

Emerging Technologies Integration

As technology is evolving very fast over time, we can integrate the Docker-based framework with new technologies like AI, IoT, and edge computing are very exciting opportunities. In

these areas, we can explore the other new use cases in these technologies where docker will be able to increase the coordination and performance of these technologies by focusing on decentralizing them. With these emerging technologies, it is possible to increase the capability and ensure relevance in the modern era of embedded systems.

Glossary

CPU: Central Processing Unit

Docker: A platform for developing, shipping, and running applications inside containers.

ARM Core: A type of processor architecture widely used in embedded systems.

UML (Unified Modeling Language): A standardized modeling language used to specify, visualize, construct, and document the artifacts of a system.

References

1. D. Merkel, "lightweight Linux containers for consistent development and deployment," *Linux Journal,* no. 239, 2014. Available:https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment

2. www.takethenotes.com, "Deployment of Docker containers on embedded hardware," Aug. 27, 2024.Available: https://takethenotes.com/docker-storage-in-embedded-systems/

3. Docker, "What is Docker?" Docker Documentation. Available: https://docs.docker.com/get-started/overview/.

4. Embedded.com, "Benchmarking Embedded Systems." Available: https://www.embedded.com/benchmarking-embedded-systems/

5. Revaalo Labs, "Monolithic vs. microservices" Jul. 28, 2021. Available: https://revaalolabs.com/post/monolithic-vs-microservices-which-architecture-suits-best-for-your-project

6. Visual Paradigm, "UML Diagram" Available: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-diagram-types-overview/

7. ARM Holdings, "Introduction to the ARM Architecture." Available: https://www.arm.com/architecture

8. Docker Documentation, "Overview of Docker Compose" Available: https://docs.docker.com/compose/

9. Ubuntu Documentation, "Stress-ng." Available: https://wiki.ubuntu.com/Kernel/Reference/stress-ng

10. Iperf.fr, "Iperf - The ultimate speed test tool for TCP, UDP and SCTP." [Online]. Available: https://iperf.fr/

Declaration of Independence

I hereby declare that this research report is written independently, only with the help of the the sources and aids listed.

Location, Date	Signature
Ilmenau, 01.09.2024	Shawalkhan