TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Fakultät efi

Studiengang Elektronische und Mechatronische Systeme

Development and Evaluation of a Lifecycle Management Interface for Improving Stability and Integration of Mobile Robots based on ROS2

Masterthesis von

Aayush Yadav
Matrikel-Nr.: 3549452

Winter Semester 2021
Abgabedatum: 03.01.2022

Betreuer:
Prof. Dr. Stefan May
Technische Hochschule Nürnberg

Michael Fiedler
SIEMENS

Schlagworte: Docker, Industrial Edge, ROS2

Contents

List of Acronyms

AGV	Automated guided vehicle
CBW	Conveyor Belt Window
CLI	Command Line Interface
CSS	Cascadimg Style Sheets
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IEA	Industrial Edge Application
IED	Industrial Edge Device
IEDK	Industrial Edge Device Kit
IEH	Industrial Edge Hub
IEM	Industrial Edge Management
MVC	Model View Controller
npm	Node Package Manager
REST	REpresentational State Transfer
RFC	Request for Comments
ROS2	Robot Operating System 2
RPM	RPM Package Manager, originally Red Hat Package Manager
SAT	SIMATIC Automation Tool
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtuelle Maschine
WWW	World Wide Web

1 Introduction

1.1 Motivation

Modern robotic Systems are made up of several individual and independent components. As systems grow in size and become more complex and sophisticated it becomes more and more difficult to have a proper overview of the complete system and at the same time the troubleshooting, debugging and streamlined operation becomes more challenging. To solve this problem there is a need for a solution that helps to improve the overall system satability and facilitates system Integration. In this thesis a ros2 system with managed lifecycle nodes which run inside a docker enviroment and a management interface for these lifecycle nodes is developed. Furthermore the docker based appliation is converted to a Siemens Industrial Edge Application which helps to streamline the delivery and deployment of the developed application within the Siemens industrial edge infrastructure.

1.2 Problem Statement

In a current ROS2 based robotic system (AGV) there are several subsystems running simultaneously in order to operate seamlessly. If any of these independent systems fail the whole system fails and a developer must work on the AGV to troubleshoot and fix the problem. This entire processs is very tedious, time consuming and not efficient. If these problems could be debugged and solved from a remote (web) Interface, that would offer huge benifits to the overall system. So each individual systems will be packaged as a ROS2 lifecycle nodes with custom lifecycle behaviours, which would allow a user to dynamically reconfigure individual nodes, restart or respawm the nodes remotely and get the debug and other operationl information throug a remote logger.

Another problem is the delivery and deployment, currently all the installation process needs to be done manually, i.e a developer has to login to a specific AGV install each library and dependencies necessary for a specific application. This process can be streamlined and made way more efficient by introducing a docker based appliation that can be installed using the Siemens industrial edge infrastructure. Another major task of this thesis is to utilize the DDS offerd by ROS2 and operate a Lifecycle managment Interface within a Local network. This amplyfies the efficiency of the system drasticly and offers new offortunities to control a swarm of robots through a single unified Web-interface.

1.3 Requirements

According to [1], the following questions must be answerable after the analysis phase of the software development process:

- What are the crucial requirements for the software to be created?
- What problem is to be solved with the help of the application system?
- What do your customers and users want from the system?

- Development enviroment
- Restrictions / Bounding conditions
- Minimum vaible Product
-

2 Background

This chapter gives an outline of the concepts and a detailed explanation of the various technologies that will be used later in this thesis.

2.1 Containers

The concept of container technology uses the same model as shipping containers in transportation. The idea is that before the invention of shipping containers, manufacturers had to ship goods in a variety of fashions which included ships, trains, airplanes, or trucks, all with different sized containers and packaging. With the standardization of shipping containers, products could be transported seamlessly without further preparation using different shipping methods. Before the arrival of this standard, shipping anything in volume was a complex, laborious process. The motivation behind software containers is the same. [2, P. 1]

Instead of shipping a complete operating system (OS) and the software (with necessary dependencies), we pack our code and dependencies into an image that can run anywhere. Furthermore, it enables the packaging of clusters of containers onto a single computer. In other words, a container consists of an entire runtime environment: an application, plus all the dependencies, libraries, and other binaries, and configuration files needed to run it, bundled into one package. The ability to have software code packaged in pre-built software containers means that code can be pushed to run on servers running different Linux kernels or be connected to run a distributed app in the cloud. This approach also has the advantage of speeding up the testing process and creating large, scalable cloud applications. This approach has been in software development communities for several years. It has recently gained in popularity with the growth of Linux and cloud computing. [2, P. 2]

2.1.1 Containerization vs Virtualization

Linux containers and virtual machines (VMs) are both package-based computing environments that combine several IT system components and keep them isolated from the rest of the system. Their main distinguishing features are scalability and portability. Containers are usually measured in megabytes, whereas VMs in gigabytes. [3]

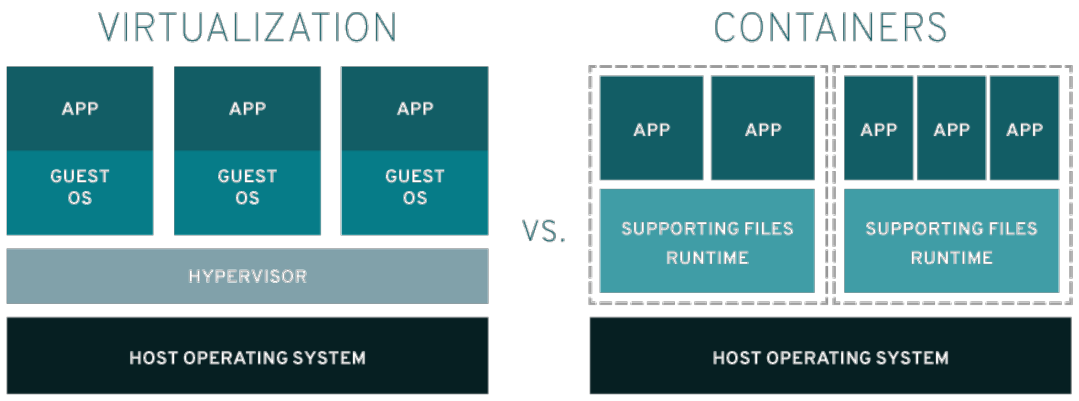


Figure 2.1: Differences between Virtualization and Containerization [3]

Containerization is an alternative to standard virtualization that encapsulates an application in a container with its executing environment. Containers hold an application and everything it needs to run. Everything within a container is maintained on an image—a code-based file that includes all libraries and dependencies. These files are similar to a Linux distribution installation. An image comes with RPM packages and configuration files. Containers are so small compared to VMs, there are usually hundreds of them loosely coupled together.[3]

Virtualization is a way of sharing a single physical instance of a resource or an application to multiple organizations and clients. It utilizes software called a hypervisor that separates resources from their physical devices. It enables the partitioning of the resources and assigned to individual VMs. When a user issues a VM instruction that requires additional resources from the physical environment, the hypervisor sends the request to the physical system and saves the changes. VMs look and act like physical servers, which can multiply the drawbacks of application dependencies and large OS footprints—a footprint that's often not required to run a single app or microservice.[3]

Table 2.1 illustrates the key differences between the above two approaches concerning package-based computing environments.

Parameters	Virtualization	Containerization
Isolation	Provides complete isolation from the host operating system and the other VMs	Provides lightweight isolation from the host and other containers, but doesn't provide a strict security boundary as a VM
Operating System	Runs a complete operating system including the kernel, thus requiring more system resources such as CPU, memory, and storage	Runs the user-mode portion of an operating system, and can be customized to include just the required services for your app utilizing fewer system resources
Compatibility	Runs just about any operating system inside the virtual machine	Runs on the same operating system version as the host
Deployment	Deploys individual VMs by using Hypervisor	Deploys single container by using Docker or deploy multiple containers by using an orchestrator such as Kubernetes
Persistent storage	Uses a Virtual Hard Disk (VHD) for local storage for a single VM or a Server Message Block (SMB) file share for storage shared by multiple servers	Uses local disks for local storage for a single node or SMB for storage shared by multiple nodes or servers
Networking	Uses virtual network adapters	Uses an isolated view of a virtual network adapter. Thus, providing a little less virtualization
Startup time	They take few minutes to boot up	They can boot up in few seconds

Table 2.1: Differences between Virtualization and Containerization [4]

The use of containers can decrease the required time for developing, testing, and deploying applications. It makes testing and fault detection less complex as there is no difference between running your application on a test environment and in production. It provides a cost-effective solution and can help reduce operational and development expenses. In most use-cases, container-based virtualization offers several advantages over traditional Virtual Machine based virtualization.

2.2 Docker

Docker is a person who works at a port whose job is to load goods onto and off container ships. [5]

Software Docker essentially does the same in the software context. Docker is a collection of open-source tools that quickly wraps up any application and all its unique dependencies in a lightweight, portable, self-sufficient container that can run virtually anywhere on any infrastructure.[6] Docker was launched as an open-source project by dotCloud, Inc. in 2013. it relies heavily on namespaces and cgroups to provide resource isolation and to package an application along with its dependencies. This bundling of dependencies into one package allows an application to run across different platforms and still support a level of portability. This provides flexibility to developers to develop in the desired language and platform. It has drawn a lot of interest in recent years.[2, P. 10]

Docker consists of several parts. The following section gives an overview of the main components of Docker.

2.2.1 The Docker Runtime and Orchestration Engine (Docker Engine)

The Docker engine is the software for the infrastructure that runs and orchestrates containers. All other Docker, Inc. and third-party products connect to and develop around the Docker Engine. It provides a workflow for building and managing the application stack. It builds and runs containers using other Docker components and services. It consists of the Docker daemon; a REST API that specifies the interfaces that programs can use to communicate with the daemon; and the CLI, the command-line interface that communicates with the Docker daemon via the API. Docker Engine creates and runs the Docker container from the Docker image file.

Following Diagramm illustrates the Docker System Architecture.

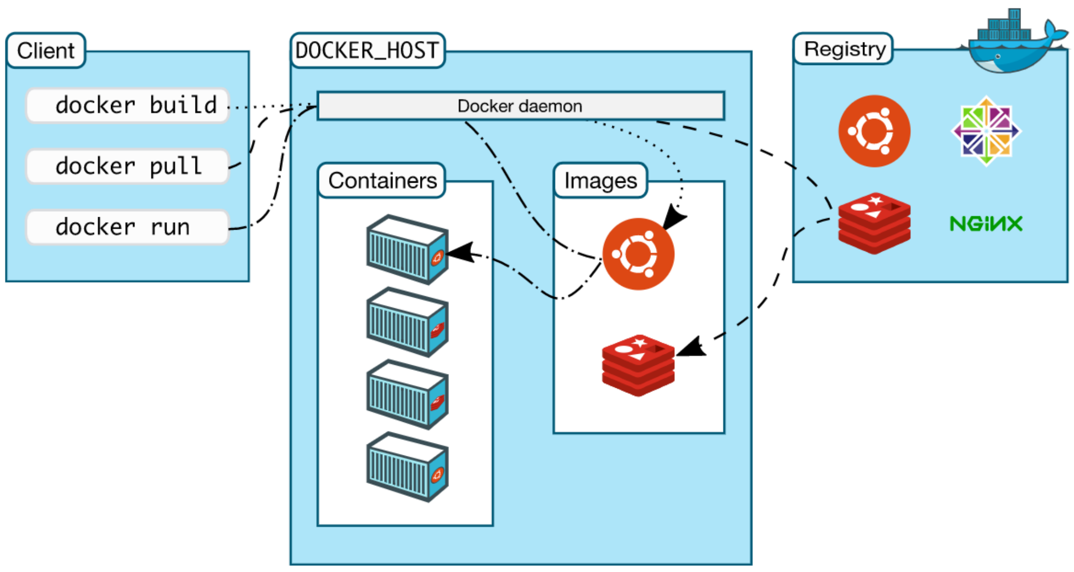


Figure 2.2: Docker System Architecture [7]

Docker daemon (dockerd) is a server process that runs in the background. It continuously listens to the REST API interface and listens for incoming requests and manages Docker objects (images, containers, networks, and volumes). A daemon also has the ability to communicate with other daemons to manage Docker services.[7]

Docker client represents the primary means for most users to interface with Docker. The commands run through the command-line interface are sent to the Docker daemon through the Docker API interface. The Docker daemon(dockerd) then executes these commands. The Docker client has the ability to connect with multiple Docker daemons.[2, P. 32]

Docker registry The images created by the Docker daemon are stored in the Docker registry. Docker looks for images in the Docker Hub by default, but it is possible to have a self hosted private registry. Docker Hub is a public registry and is freely accessible.[2, P. 33]

2.2.2 Docker Objects

Images A Docker image is a read-only file system that contains instructions to create a container in which an application can run. In most cases, a Docker Image is based on another image and is customized. You can either use existing images published in public repositories such as Docker Hub or Create your image. A Dockerfile is used to create a Docker image. A Dockerfile contains simple instructions that can be understood by the Docker daemon to create the image and run it. Docker images are layers that correspond to each instruction in the Dockerfile. Part of what makes a Docker image super easy is that when you change a part of the Dockerfile, only that layer is changed, and not the entire image.

Containers A Docker container is an instance of an image. An image runs inside a container. You can manage a container with the stop, start, and delete commands to manage it. Multiple containers can be connected over a network. They can be connected to the memory, and they can also communicate with each other. Containers are much more lightweight than VMs because their startup times are very fast. To create a container, in addition to the container's configuration and settings, you also create an image Configuration and settings an image is created. When a container is deleted everything related to the container is also deleted, including state and memory.

Services In a distributed application, different functionalities of the app provide different services. For example, if you are building an application that suggestions based on keywords that the user enters, you might want to have A front-end service that takes the word and sends it to the service that will Verifies the legitimacy of the word. This in turn could be sent to another service that runs an algorithm to generate the suggestions, etc., which are then returned to the service. These are all different services on different Docker containers, sitting Sit behind different Docker daemons. These Docker daemons are all connected over the network and interact with each other. All these services work together as a swarm, managed by different Managers and workers to manage. Each swarm contains a Docker daemon. These Daemons communicate with each other using the Docker API. A Docker Compose YAML file is used to get all these services running together. together to get them running.

Networks

Volumes

Dockerfiles Dockerfile is a text document that contains a set of instructions or commands for assembling an image that is understood by the build engine. The Dockerfile defines what goes into the environment inside your container. Accessing resources, mapping volumes, passing arguments, copying files that need to be inside your container are defined in this file. According to Dockerfile created, you need to build it to create the image of the container. Create container image. The image is just a snapshot of all the executed statements in the Dockerfile. Once this application image is created, you can expect that it will run on any machine that uses the same kernel.

Docker-Compose Docker Compose is the tool for running multi-container Docker applications. It is essentially a YAML file that can be thought of as a Compose of multiple Dockerfile containers, which can be used to put commands into a single file. This Docker Compose YAML file contains configurations of multiple services. Then, with a single command, you can run all services to run in Docker containers at the same time. Docker Compose can be used to create a microservices architecture and link the containers together, or it can be used for a single service. In addition, Docker Compose can create images, scale containers, and re-run containers that have been stopped. All of these functions are part of Docker. Docker-compose is just a higher-level abstraction of container execution commands. Everything that a compose file can do, can likewise be performed with simple Docker commands, except that this requires more memory and additional overhead to execute any additional commands, to connect to the network, etc. . Docker-compose helps to simplify this process.

2.3 Industrial Edge

Industrial Edge is a development from Siemens that makes it possible to analyze data generated in various industrial processes on the device itself. It combines local engineering with cloud engineering.[8]

This eliminates unnecessary data transport between the end device and the server, only processed data is sent to the server. The processing load is in this way directed from the server towards the end device. Edge applications serve this purpose, they contain a fixed set of functions and can be accessed with the edge infrastructure on the respective end device (edge device). These apps are based on Docker technology and run one or more Docker containers in them. Figure 2.3 shows a typical infrastructure from the top server to the end devices.

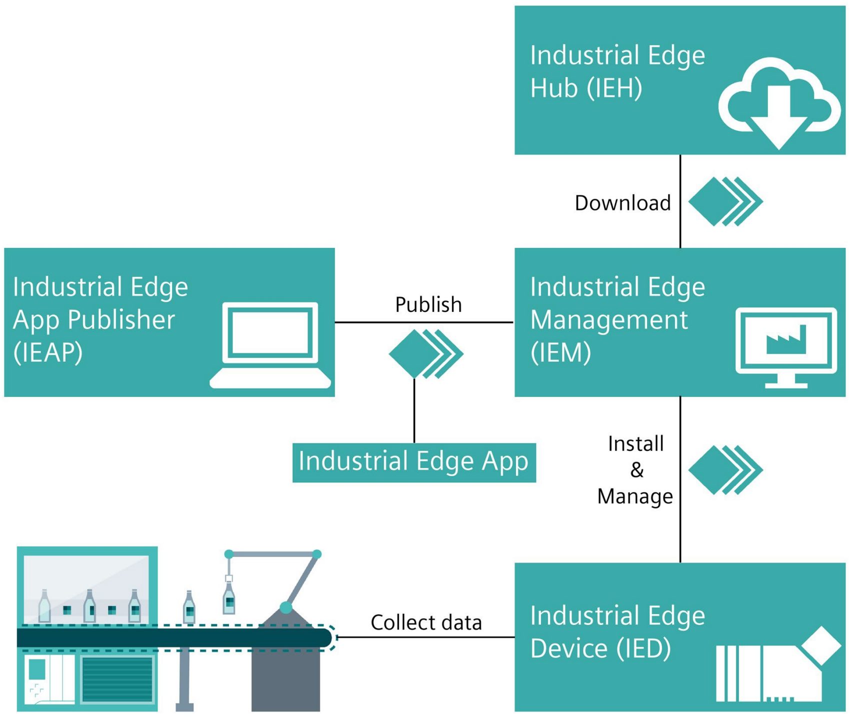

Industrial Edge Hub (IEH)
Download
Industrial Edge App Publisher (IEAP)
Publish
Industrial Edge Management (IEM)
Industrial Edge App
Install & Manage
Collect data
Industrial Edge Device (IED)

Figure 2.3: Industrial Edge overview [9]

Industrial Edge Management System The Industrial Edge Management (IEM) is a server, which can be operated independently in a local network. It allows the user to create their own server which avoids the transmission of raw sensitive data exchanged between the edge device and the IEM over the Internet.

In addition, the end(edge) devices can be managed via this server, which includes the installation of apps and software updates and further analysis of individual apps. A self-developed Application can be uploaded to the IEM, which in turn can distribute it to the edge devices.[8]

Industrial Edge Hub The Industrial Edge Hub (IEH) is the top server-level Application hub provided by Siemens. On this platform, Siemens offers applications developed in-house. These can be loaded onto the IEM for a license fee. Any required software packages and documentation can be downloaded from here. This streamlines the development process of custom applications including deployment, and operation on desired IEM.[8]

Industrial Edge App An Industrial Edge Application (IEA) (edge app) is used for the intelligent processing of industrial automation tasks.[8] It is a Docker-based image, which is runs on the IED. A docker-compose file acts as the top description level file, it specifies various parameters and the sequence in which the Docker images are started. The parameters include network configuration, data storage, and other application-specific configurations. It supports all Docker-compose version 2.4 settings. Modification of this docker-compose file allows configuring any Edge app, even after it has been already downloaded to an edge device. An Edge App can be downloaded to an edge device through the IEM.

Edge Device Edge devices are required in order to run individual edge apps. The edge device is a custom Linux Machine running the Industrial Edge OS. For test and simulation purposes it can also be run in a Virtual machine(VM). Edge devices can save automation data locally and retrieve it when required. In addition, edge devices can upload this data into the cloud infrastructure and retrieve it at any moment. After proper configuration and connection, an Edge Device can be activated through IEM using an Edge Device configuration file.

Industrial Edge Publisher The Industrial Edge Publishers is a tool that converts Docker images into Edge Apps and uploads them to the IEM. It can also be used to manage, modify or delete apps that have already been created.[10] A prerequisite for creating an app is a docker-compose file, which contains the startup parameters of a Docker container. These parameters can also be configured through a menu within the Edge Publisher.

2.4 ROS

This chapter covers the basics of the robot operating system (ROS) and all the tools necessary to create, debug, and understand robot applications. This chapter describes some high-level concepts and low-level API commands which enable developers to develop, maintain and support multi-robot applications.

The software stack of a typical robot system requires several software tools which include hardware drivers, network modules, communication architecture, and several application-specific algorithms. ROS provides these tools in one package, which saves developers a lot of time and redundant work. ROS includes several sub-packages for robot navigation, vision, control, simulation.

The Robot Operating System (ROS) has long been one of the most widely used middleware for robotic. The large open-source robotics community has contributed to a lot of new features since the introduction of ROS 1 in 2007, limitations of ROS1 have led to the development of ROS2.[11]

2.4.1 ROS2

Robot Operating System 2 (ROS2) was launched with an improved architecture and upgraded features. It is new and various organizations and open-source communities are trying to port existing packages to ROS 2.[11]

Table 2.2 illustrates the key differences between ROS1 and ROS2

Parameters	ROS1	ROS2
Networking	Utilizes the TCPROS communication protocol	Uses DDS (Data Distribution System) for communication.
ROS Master	Uses ROS Master for centralized discovery and registration. The whole communication system fails if the master fails.	Uses the distributed DDS discovery. ROS 2 provides a custom API to get all information about nodes and topics.
Compatibility	ROS runs only in Ubuntu.	ROS 2 is compatible with Ubuntu, Windows 10, and OS X.
Programming language	Uses C++ 03 and Python2.	Uses C++ 11 and Python3.
Build system	ROS uses only the CMake build system and has a combined build for multiple packages via a single CMakeLists.txt file	ROS 2 offers options to use other build systems. Supports isolated independent builds for packages to better handle dependencies between packages.
Default values	Data types in message files do not support default values.	Data types in message files support default values.
roslaunch	roslaunch files are written in XML.	roslaunch files are written in Python which makes it more configurable and and supports conditional execution.
Realtime Support	Does not support Real-time behavior.	Supports real-time responses with a suitable RTOS.

Table 2.2: Differences between ROS1 and ROS2 [12]

2.4.2 ROS2 Structure

The key ROS2 feature is that it uses DDS (Data Distribution System) for communication. It eliminates ROS Master and utilizes distributed DDS discovery. A set of related nodes located on the same computer are called a hub, where one or more hubs can be present on the same computer. ROS hubs are considered to have an accompanying set of bridging nodes that pass messages between ROS and ROS2.[1, P.7]

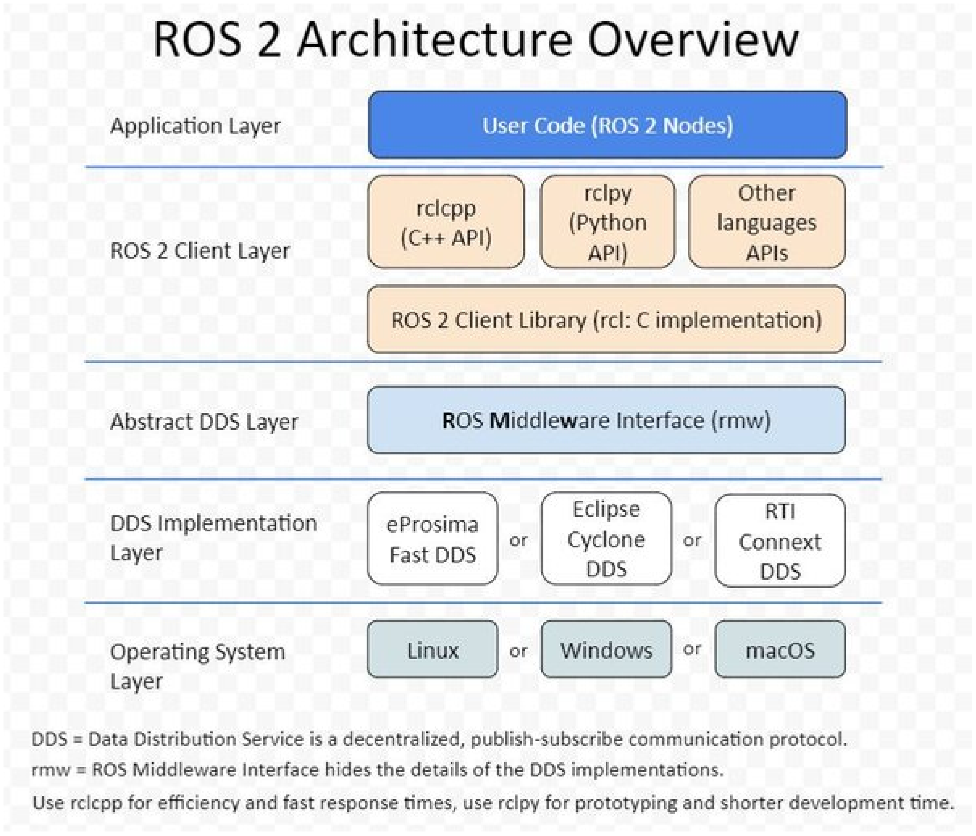

ROS 2 Architecture Overview
Application Layer
User Code (ROS 2 Nodes)
ROS 2 Client Layer
rclcpp (C++ API)
rclpy (Python API)
Other languages APIs
ROS 2 Client Library (rcl: C implementation)
Abstract DDS Layer
ROS Middleware Interface (rmw)
DDS Implementation Layer
eProsima Fast DDS
or
Eclipse Cyclone DDS
or
RTI Connext DDS
Operating System Layer
Linux
or
Windows
or
macOS
DDS = Data Distribution Service is a decentralized, publish-subscribe communication protocol.
rmw = ROS Middleware Interface hides the details of the DDS implementations.
Use rclcpp for efficiency and fast response times, use rclpy for prototyping and shorter development time.

Figure 2.4: ROS2 Architecture [13]

2.4.3 ROS2 Nodes (Application layer)

In ROS distributed applications are typically designed as entities referred to as nodes. Each Node in ROS is supposed to be responsible for a single, modular function. In a Robotic Control System, sensors (lidar, camera), motion controllers (motors for movement), and algorithm components (route planners) can each be nodes. A node can send and receive data to and from other nodes via topics, services, actions, or parameters. ROS 2 separates the node concept from the process structure at the operating system level. Multiple nodes can be created within a process as desired, and these nodes can independently interact with other nodes. All nodes in the system can run on a single computer or they can be distributed and run on multiple computers.

ROS2 Communication Patterns The nodes communicate with each other via topics, service calls, and actions. Topic-based Communication is based on a publish-subscribe architecture, where the data generated and published by a node can be subscribed to by more than one node, and likewise, a Topic can produce data for more than one node. The distributed architecture of the topic topology offers improvements in terms of performance and fault tolerance. Additionally, the extensive quality of service(QoS) features in the DDS middleware layer make ROS-2 architecture more robust and enable developers to use it in an industrial-/real-life application.

A comprehensive robotic system has several nodes working together in harmony. In ROS 2 a single executable (C++ program, Python program, etc.) can hold one or more nodes in them. This network of ROS 2 elements running simultaneously and processing data is referred to as a ROS graph. It includes all executables and the connections between them.

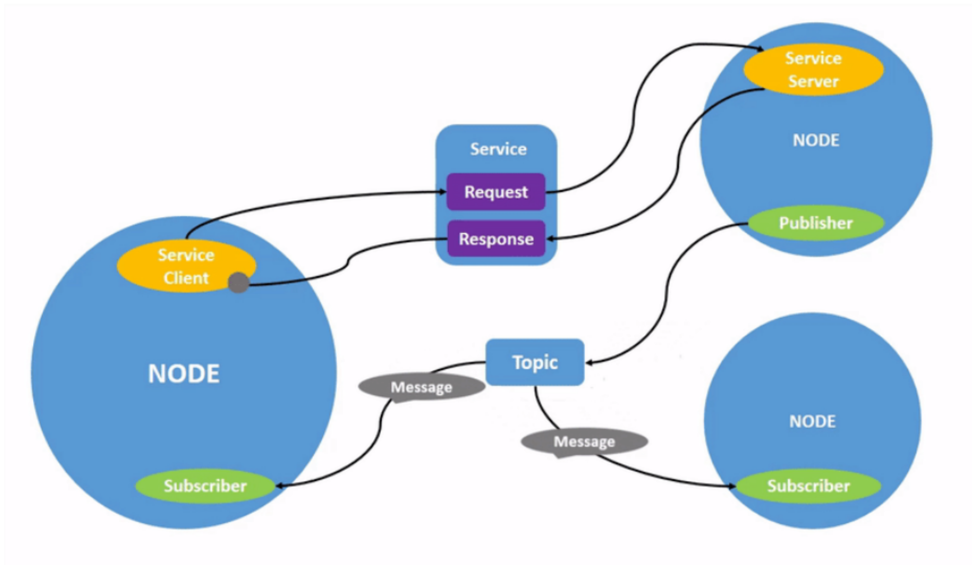


Figure 2.5: ROS2 Communication Pattern [11]

Messages: A message is a data file that is sent via a communication channel called. Messages in ROS 2 are of great importance for interoperability. A publisher sends its message to a specific topic. This topic can be subscribed to by one or more subscribers. Several publishers can send messages to a topic and several subscribers can receive them. The communication in messages is unidirectional. A topic must use a message type that is known to all participants. There are predefined standard message types that are supplied with the ROS2 installation. Custom message types can be defined in a .msg- file. A message can contain other message definitions and can be extended using this pattern.[14]

Topics: Topics act as virtual communication channels (bus) through which data is moved between nodes and consequently between various components of the system. Topics aren't just one-to-one connections moreover, they can be many-to-one, one-to-many, and many-to-many connections. Topics are a key element in realizing Publish-Subscribe architecture. Topics define the name and data structure used. Each topic in ROS2 is also an instance that can store historical message data in that topic.

Service: Two-way communication in ROS2 is made possible with services. A request is sent to a service server, which then responds to the request sent by the service client. A single service communication channel can contain several service clients but only one service server. The service server and service client must be both able to recognize the message type used. Custom service message types can be defined with a .srv file.

This file defines the request and the response of service communication. A service can contain other service message definitions.

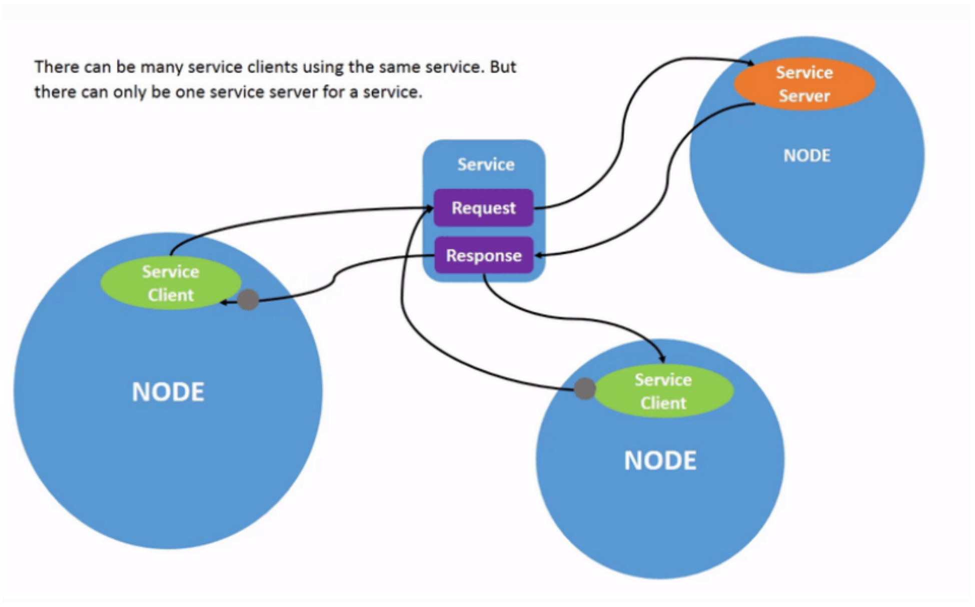

There can be many service clients using the same service. But there can only be one service server for a service.
Service
Request
Response
Service Server
NODE
Service Client
NODE
Service Client
NODE

Figure 2.6: ROS2 Services [14]

Action: Actions are an advanced communication pattern in ROS2 that's designed for long-running tasks. They are made up of three parts: a goal, feedback, and result. Topics and services are the building blocks of Actions. Their functionality is similar to that of services, with the difference that actions are interruptible (they can be individually ended during execution). Unlike services, which only return a single response, they provide constant feedback.

Actions use a client-server model, similar to the publisher-subscriber model. An "action client" node sends a goal to an "action server" node, which acknowledges the goal and returns a stream of feedback and a result.

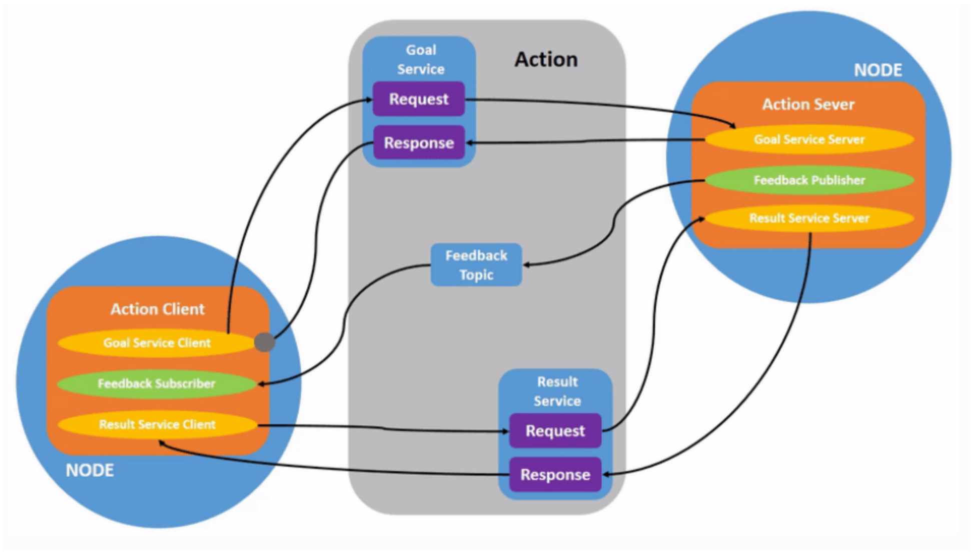


Figure 2.7: ROS2 Actions [14]

Figure 2.7 gives an overview of the inner workings of an Action. The action client makes a Goal Request, which after processing can either be accepted or rejected by the action server. In case, the Goal Request is accepted it is forwarded to the action client. Then a user-defined function is called that executes the desired task of the action server. This Function continuously sends its current status or other custom information as feedback to the action client. Upon completion of the task and achievement of the defined goal, the action server sends the corresponding result response. Within an action communication channel, there can be several action clients, but only one action server. The action message type used must be recognized by all the participants in advance. Own action message types can be defined with a .action file. This contains the definition of Goal, Request, and Feedback. Other message definitions can be used from an existing action definition.

2.4.4 ROS2 Client Library

Client libraries are the APIs that users can use to implement their ROS code. Client libraries give users access to ROS concepts such as nodes, themes, services, etc. Client libraries come in a variety of programming languages, so users can write ROS code in the language that best suits their application.

Nodes written with different client libraries can exchange messages with each other because all client libraries implement code generators that allow users to interact with ROS interface files in the respective language.

2.4.5 ROS Middleware Interface

2.4.6 Important ROS2 Concepts

The ROS DOMAIN ID
About different ROS 2 DDS/RTPS vendors
About logging and logger configuration
About ROS 2 client libraries
About ROS 2 interfaces
About parameters in ROS 2
About topic statistics
Introspection with command line tools
About Composition
On the mixing of ament and catkin (catment)

2.4.7 ROS2 Web Bridge

2.4.8 roslibjs

2.4.9 DDS

- Grundlage für Kommunikation von ROS2
- Realisiert die eigentliche Kommunikation
- Wenn das hier geht, geht auch ROS2! - Verschiedene Systemanbieter, näher wird RTI und Fastrtps untersucht: https://ros.org/reps/rep-2000.html (Beide TIER 1)

2.5 Ros Lifecycle nodes

Having a managed lifecycle for nodes allows for better control over the state of the ROS system. This ensures that in a ROS2 Robotics System all components have been correctly instantiated before allowing a component to perform its behavior. It also allows nodes to be restarted or replaced online.

A managed node has a known interface, executes according to a known lifecycle state machine. This allows the application (node) developer to decide how to implement various managed lifecycle functions while securing compatibility with other lifecycle nodes.

The base functionality of a managed lifecycle node is described using a state machine illustrated in figure 2.8.

"A state machine is a mathematical abstraction used to design algorithms. A state machine reads a set of inputs and changes to a different state based on those inputs. A state is a description of the status of a system waiting to execute a transition. A transition is a set of actions to execute when a condition is fulfilled or an event received." [15]

There are 4 primary states (colored blue in figure 2.8):

- Unconfigured
- Inactive
- Active
- Finalized

In addition, there are 6 transition states, which are intermediate states during a requested transition(colored in yellow in figure 2.8):

- Configuring
- CleaningUp
- Activating
- Deactivating
- ShuttingDown
- ErrorProcessing

There are 7 transitions exposed to the lifecycle management interface, which can be invoked only when the necessary conditions are met. These conditions are described in detail in the next section. The table below lists these transitions with the corresponding behavior invoked during their execution.

Table 2.3: Availabe Transitions

Transition	Description
create	with this transition the node is instantiated, but will not run any code beyond the constructor. On successful execution of the constructor the node is in Unconfigured state.
configure	*onConfigure()* callback is called
cleanup	*onCleanup()* callback is called
activate	*onActivate()* callback is called
deactivate	*onDeactivate()* callback is called
shutdown	*onShutdown()* callback is called
destroy	with this transition the node is destroyed (deallocated). The node is completely removed from the system.

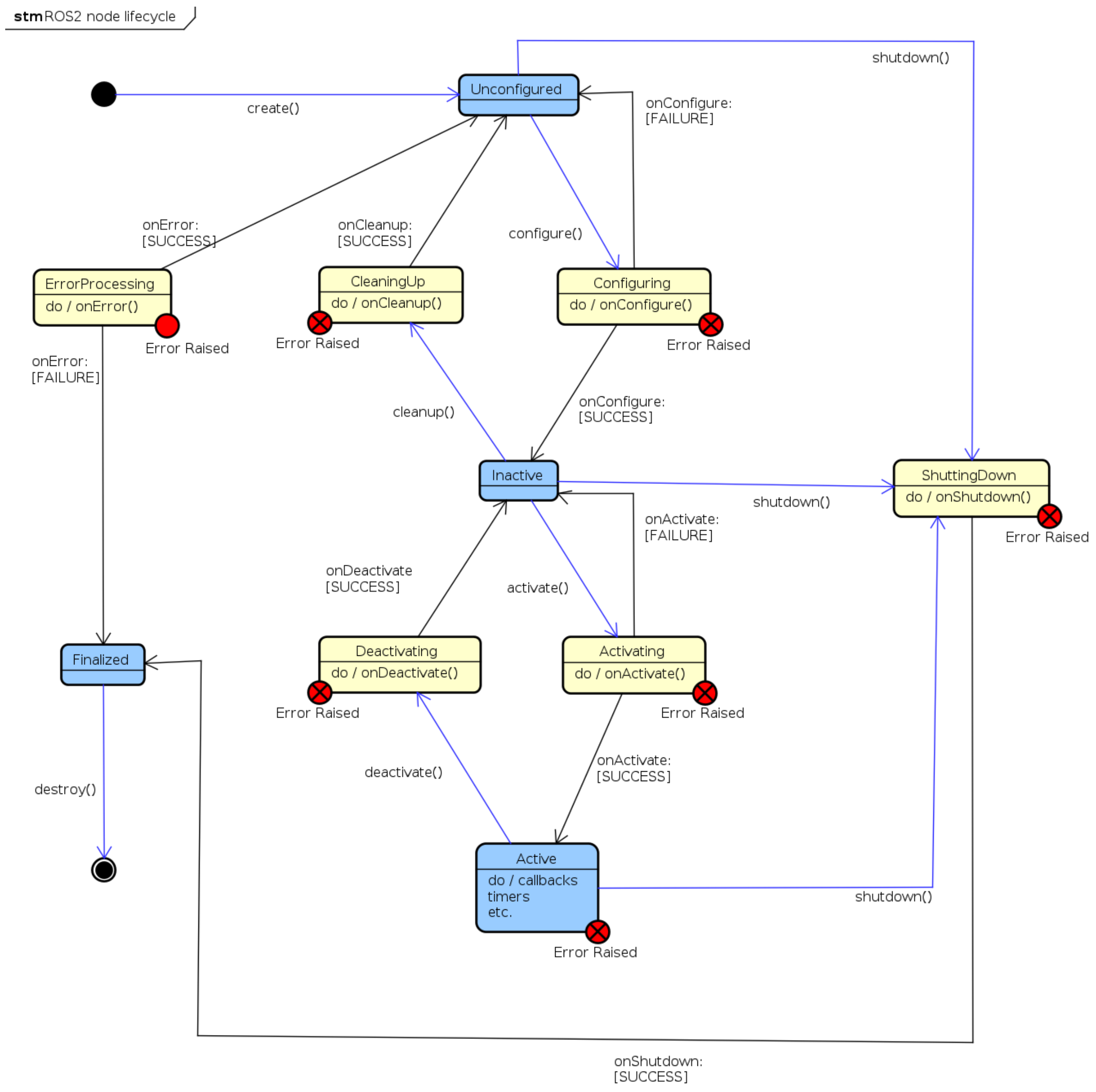


Figure 2.8: ROS2 Lifecycle Statemachine [16]

2.5.1 Primary States

The transition from a primary state requires an action of an external monitoring process, except in the case of an error triggered in the 3rd primary ("active") state.

Unconfigured This is the starting state of the state machine. When a node is instantiated, it finds itself in this lifecycle state. In case of any unexpected error, a node can be returned to this state. In this state, it is expected that there is no saved state. From this state the following transitions via corresponding transition states are availabe:

Transition via	Final state
configure	Inactive
shutdown	Finalized

Inactive This state represents a node that is not currently operating. The main purpose of this state is to allow reconfiguration of a node (modifying configuration parameters, adding and removing topic publications/subscriptions, etc.) without changing its behavior during operation. In this state, the node does not read topics, process data, respond to functional service requests, etc. In the inactive state, all data arriving in managed topics is not read or processed.

Data storage is subject to the configured QoS policy for the topic. Any managed service requests to a node that is in the inactive state will not be answered (they will fail immediately for the caller). From this state the following transitions via corresponding transition states are availabe:

Transition via	Final state
shutdown	Finalized
cleanup	Unconfigured
activate	Active

Active It is the central state of the node's lifecycle. In this state, the node performs all processing, responds to service requests, reads and processes data, generates output, etc. All the important user functionality should be implemented in this state.

When an error occurs in this state that cannot be handled by the node/system, the node enters the ErrorProcessing state. From this state the following transitions via corresponding transition states are availabe:

Transition via	Final state
deactivate	Inactive
shutdown	Finalized

Finalized In this state, the node ends its execution immediately before its deallocation. This state is always final, the only transition from here for a node is to be destroyed and completely removed from the system.

The main purpose of this state is to support debugging and introspection. A failed node remains visible for introspection in the system and can potentially be examined by debugging tools before being destroyed. If a node is started in a respawn loop or there are known reasons for the cycling into this state, the supervising monitoring process can automatically destroy and recreate the node based on a predefined policy. From this state the following transitions via corresponding transition states are availabe:

Transition via	Final state
destroy	Deallocated

2.5.2 Transition States

Application-specific logic is executed in the transition states to determine if the requested transition was successful. The lifecycle management software is notified of success or failure via the lifecycle management interface before transitioning to another state.

Configuring In this transition state, the node's *onConfigure()* callback is invoked to allow the node to load its configuration and make any necessary changes to its settings. Configuring a node typically involves tasks that need to be done once during the node's lifetime, such as allocating permanent memory buffers and setting up topic publications/subscriptions that do not change. The node uses this to set up all the resources it needs to keep throughout its lifetime (whether it is active or inactive). From this state the following transitions are availabe:

Transition condition	Final state
onConfigure callback succeeds	Inactive
onConfigure callback fails with an error code that can be handled by the node/system	Unconfigured
onConfigure callback fails with an error that cannot be handled by the node/system	ErrorProcessing

CleaningUp In this transition state, the node's *onCleanup()* callback is invoked, which is expected to clear all states and return the node to a functionally equivalent state as when it was first created. If the clean-up cannot be performed successfully, the method moves on to ErrorProcessing. From this state the following transitions are availabe:

Transition condition	Final state
onCleanup callback succeeds	Unconfigured
onCleanup callback fails with an error that cannot be handled by the node/system	ErrorProcessing

Activating In this transition state, the node's *onActivate()* callback is invoked to allow the node to make the final preparations before starting execution. This may include obtaining resources necessary when the node is active(e.g. accessing the hardware). Ideally, no time-consuming operations should be made in this callback (e.g. a lengthy hardware initialization). From this state the following transitions are availabe:

Transition condition	Final state
onActivate callback succeeds	Active
onActivate callback fails with an error that cannot be handled by the node/system	ErrorProcessing

Deactivating In this transition state, the node's *onDeactivate()* callback is invoked, which is expected to do any cleanup necessary to start execution, and is supposed to reverse the changes made by *onActivate()* callback . From this state the following transitions are availabe:

Transition condition	Final state
onDeactivate callback succeeds	Inactive
onDeactivate callback fails with an error that cannot be handled by the node/system	ErrorProcessing

ShuttingDown In this transition state, the node's *onShutdown()* callback is invoked, which is expected to do any cleanup necessary before destruction of the node. It can be transitioned into from any Primary State except Finalized. The originating state is passed to this state as a parameter . From this state the following transitions are availabe:

Transition condition	Final state
onShutdown callback succeeds	Finalized
onShutdown callback fails with an error that cannot be handled by the node/system	ErrorProcessing

ErrorProcessing This state exists in order to be able to clear any unhandled errors. All error processing functionality should be implemented in this state. This state can be transitioned into from any state where user code is executed. In case of successful error handling, the node can return to the Unconfigured state. If complete clean-up is not possible, the node must fail, and transition to the Finalised state. Transitions to ErrorProcessing can be triggered by errors in callbacks as well as by other methods within a callback or an uncaught exception. From this state the following transitions are availabe:

Transition condition	Final state
onError callback succeeds	Unconfigured
onError callback fails	Finalized

2.5.3 Management Interface

A managed node is exposed to the ROS ecosystem through the following interface. This interface can be used by applications that perform the actual management. This interface is independent of the lifecycle status of individual nodes and is thus not be subject to any communication restrictions.

A managed node or lifecycle node has a common pattern which it inherits from a container class that loads a managed node implementation from a library and, through a plug-in architecture, automatically exposes the required management interface by implementing callback methods.

A lifecycle node does not inherit from the regular `rclcpp::node::Node` but from `rclcpp_lifecycle::LifecycleNode`. Every child of LifecycleNodes has a set of callbacks provided. These callbacks go along with the applied state machine attached to them. These callbacks are:

```
rclcpp_lifecycle::node_interfaces::
LifecycleNodeInterface::CallbackReturn
on_configure(const rclcpp_lifecycle::State & previous_state)

rclcpp_lifecycle::node_interfaces::
LifecycleNodeInterface::CallbackReturn
on_activate(const rclcpp_lifecycle::State & previous_state)

rclcpp_lifecycle::node_interfaces::
LifecycleNodeInterface::CallbackReturn
on_deactivate(const rclcpp_lifecycle::State & previous_state)

rclcpp_lifecycle::node_interfaces::
LifecycleNodeInterface::CallbackReturn
on_cleanup(const rclcpp_lifecycle::State & previous_state)

rclcpp_lifecycle::node_interfaces::
LifecycleNodeInterface::CallbackReturn
on_shutdown(const rclcpp_lifecycle::State & previous_state)
```

Listing 2.1: Transition Callbacks

These management services can be implemented via attributes and method calls (for local management) and via ROS messages and topics/services (for remote management). In case any another ROS middleware interface is used, specific topics placed in a suitable namespace are used. Each possible management transition is provided as a ROS service which can be called directly with the ROS2 command-line interface. The service reports whether the transition was completed successfully. Following is a list of available services which maybe called from ROS CLI or RosLibJS (ROS2 JavaScript API):

```
/<name_of_lifecycle_node>/change_state
/<name_of_lifecycle_node>/describe_parameters
/<name_of_lifecycle_node>/get_available_states
/<name_of_lifecycle_node>/get_available_transitions
/<name_of_lifecycle_node>/get_parameter_types
/<name_of_lifecycle_node>/get_parameters
/<name_of_lifecycle_node>/get_state
/<name_of_lifecycle_node>/get_transition_graph
/<name_of_lifecycle_node>/list_parameters
/<name_of_lifecycle_node>/set_parameters
/<name_of_lifecycle_node>/set_parameters_atomically
```

Listing 2.2: Available Services for a Lifecycle Node

2.5.4 Lifecycle management CLI

ROS2 developers have developed a basic CLI to interact with the lifecycle nodes and control their specific behaviours. It offers following functionalities and can be used as described below.

```

usage: ros2 lifecycle <command> <options>

Various lifecycle related sub-commands
Commands:
        get     Get lifecycle state for one or more nodes
        list    Output a list of available transitions
        nodes   Output a list of nodes with lifecycle
        set     Trigger lifecycle state transition
```

Get current lifecycle state

```

usage: ros2 lifecycle get [-h] [--spin-time SPIN_TIME]
   [--no-daemon][--include-hidden-nodes]
   [node_name]

Get current lifecycle state

positional arguments:
node_name                  Name of the ROS node
```

Trigger lifecycle state transition

```

usage: ros2 lifecycle set [-h] [--spin-time SPIN_TIME]
   [--no-daemon][--include-hidden-nodes] node_name
   transition

Trigger lifecycle state transition

positional arguments:
node_name                  Name of the ROS node
transition                 The lifecycle transition
```

Output a list of available transitions

```

usage: ros2 lifecycle list [-h] [--spin-time SPIN_TIME]
   [--no-daemon][--include-hidden-nodes] [-a] node_name

Output a list of available transitions

```

```
positional arguments:
node_name                 Name of the ROS node
```

Default values for transitions:

```
# Reserved [0-9], publicly available transitions.
# When a node is in one of these primary states, these
    transitions can be
# invoked.

# This transition will instantiate the node, but will not
    run any code beyond
# the constructor.
uint8 TRANSITION_CREATE = 0


# configuration and conduct any required setup.
uint8 TRANSITION_CONFIGURE = 1

# node to load its configuration and conduct any required
    setup.
uint8 TRANSITION_CLEANUP = 2

# The node's callback onActivate will be executed to do
    any final preparations
# to start executing.
uint8 TRANSITION_ACTIVATE = 3

# The node's callback onDeactivate will be executed to do
    any cleanup to start
# executing, and reverse the onActivate changes.
uint8 TRANSITION_DEACTIVATE = 4

# This signals shutdown during an unconfigured state, the
    node's callback
# onShutdown will be executed to do any cleanup necessary
    before destruction.
uint8 TRANSITION_UNCONFIGURED_SHUTDOWN  = 5

# This signals shutdown during an inactive state, the
    node's callback onShutdown
# will be executed to do any cleanup necessary before
    destruction.
uint8 TRANSITION_INACTIVE_SHUTDOWN = 6

# This signals shutdown during an active state, the
    node's callback onShutdown
# will be executed to do any cleanup necessary before
    destruction.
```

```
uint8 TRANSITION_ACTIVE_SHUTDOWN = 7

# This transition will simply cause the deallocation of
    the node.
uint8 TRANSITION_DESTROY = 8

# Reserved [10-69], private transitions
# These transitions are not publicly available and cannot
    be invoked by a user.

# Reserved [90-99]. Transition callback success values.
# These return values ought to be set as a return value
    for each callback.
# Depending on which return value, the transition will be
    executed correctly or
# fallback/error callbacks will be triggered.

# The transition callback successfully performed its
    required functionality.
uint8 TRANSITION_CALLBACK_SUCCESS = 97

# The transition callback failed to perform its required
    functionality.
uint8 TRANSITION_CALLBACK_FAILURE = 98

# The transition callback encountered an error that
    requires special cleanup, if
# possible.
uint8 TRANSITION_CALLBACK_ERROR = 99

##
## Fields
##

# The transition id from above definitions.
uint8 id

# A text label of the transition.
string label
```

2.6 Vue JS

Vue is described as a progressive web framework. This means that it adapts to the needs of the developer. While other frameworks require a complete adoption of a technology by a developer or team and often require an existing application to be rewritten because of certain framework-specific conventions. Vue can be introduced to an application with a simple script tag and can grow with requirements, from a small web component to managing the entire view layer. Vue is a scalable JavaScript framework for building user interfaces. Unlike other frameworks, Vue is designed from the ground up to be incrementally customizable. The core library is view-only and easily integrates with other libraries or existing projects. In combination with modern tools and additional libraries, Vue is capable of running sophisticated single-page applications [17]. It is not necessary to have prior knowledge of Webpack, Babel, npm or the like to get started with Vue. HTML, CSS, and the basics of JavaScript are enough to get started with Vue.

The leap from a pure HTML and CSS-based website to a sophisticated web application is very simple and can be learned during the development process. This is a strong selling point, especially in the current ecosystem of JavaScript front-end frameworks and libraries, which makes newcomers and even experienced developers feel lost in the ocean of possibilities and choices. The goal of Vue is to be able to create reasonable web applications with minimal prior knowledge.

2.6.1 Main Features of Vue

Together with React and Angular, Vue is one of the most popular frameworks in the web development landscape. The following features characterize the Vue framework:

Components: Vue components extend basic HyperText Markup Language (HTML) elements to encapsulate reusable code. The Vue component system is an abstraction that enables the creation of large applications from small, self-contained, and often reusable components.

Templates: Vue.js uses an HTML/Cascadimg Style Sheets (CSS)-based template syntax. Vue compiles the templates into Virtual DOM render functions. In combination with the reactivity system, Vue can intelligently determine the minimum number of components that need to be updated. This minimizes the number of DOM manipulations when the application changes state.

Reactivity: Vue has a reactivity system that uses pure JavaScript objects and optimizes rendering. Each component keeps track of its reactive dependencies during rendering. This allows the system to intelligently and optimally re-render the components.

Integration: It is very easy to integrate Vue into existing projects or to add third-party libraries to Vue. All JavaScript libraries that use ES6+ can be used within Vue. For example, Bootstrap, Material Design, ThreeJS, Web Sensors API, etc. Vue also offers the possibility to use TypeScript instead of JavaScript. [18]

2.6.2 Vue in comparision to Angular und React

Vue borrowed some of the Angular templating syntaxes but removed the complex stack that Angular required. This made it very powerful. Vue took many good ideas from React, especially the virtual DOM. But Vue implements it with a kind of automatic dependency management that tracks which components are affected by a change in the state so that only those components are re-rendered when the state property changes. In React, on the other hand, if any part of the state that affects a component changes, that component is re-rendered, and thus any associated child components are also re-rendered. This is an advantage for Vue in terms of usability and performance enhancement [19].

Table 2.4: Vue vs React vs Angular [20]

Criterion	Vue	React	Angular
Focus	Usability	Flexibility	TypeScript
Complexity	Low	Medium	High
Size	80 KB	100 KB	500+ KB
Release	2014	2013	2010
Developed by	Evan You	Facebook	Google
Language	JavaScript	JavaScript	TypeScript
Model	virtual DOM	virtual DOM	Model View Controller (MVC)
Supported by	Open Source	Facebook	Google
Latest version	2.6.11	16.13.1	9.1.11

2.6.3 Structure of a Vue.js project

Vue has a very intuitive project structure. In addition to the core framework, it includes a lot of utilities that make front-end development with Vue very enjoyable. Like most JavaScript projects, NodeJS is used for Vue development. Vue provides the core library and the add-on utilities via `Node Package Manager (npm)`.

NodeJS is a JavaScript runtime environment based on Chrome's V8 JavaScript engine. It allows JavaScript code to be executed outside of a web browser. NodeJS allows developers to use JavaScript to write command-line tools and execute scripts server-side to generate dynamic web page content before the page is sent to the user's web browser. NodeJS supports unifying web application development around a single programming language, rather than using different languages for server-side and client-side scripting [21].

npm (node package manager) is the largest software registry in the world. Open-source developers from every continent use npm to share and access packages, and many organizations also use npm to manage private development [22].

Vue-CLI: Vue provides an official CLI(Command Line Interface) for quickly setting up modern single-page applications. It provides a comprehensive build setup for a modern front-end workflow. It takes just minutes to get up and running with hot-reload, lint-on-save, and production build.

Installation The installation of the required packages is done with NodeJS and npm. The following commands must be executed to create a basic Vue application:

- Install the Vue CLI globally:

```
npm install -g @vue/cli
```

- The Vue-Command Line Interface (CLI) is used to create a new Vue project.

```
vue create example_vue_app
```

This opens a CLI where the desired configuration for a Vue project can be set.

- To start the development server, the following command is executed:

```
npm run serve
```

- To create a production build for the app, the following command is executed:

```
npm run build
```

A typical Vue project with Vue-Router and Vuex has the following structure:

Table 2.5: Structure of the project folder

Files/Folders	Description
public/index.html	This is the main app file loaded by the browser. The file contains only a simple HTML tag in the body: `<div id=app> </div>`. This is the element that attaches the Vue application to the DOM.
src/main.js	This is the JavaScript file responsible for configuring the Vue.js application. It is also used to register all third-party packages.
src/App.vue	This is the root component that contains the HTML content that is displayed to the user.
/package.json	In this file npm stores the names and versions of the package it installed.
src/components/	This folder contains the additional components.
src/assets/	This folder is used to store static content, such as images.
src/router/	This folder contains the implementation of Vue-Router.
src/store/	The folder contains the implementation of Vuex-Store.

2.6.4 Vue Instance

Every Vue application begins by creating a new Vue instance (root instance) with the Vue function. When a Vue instance is created, it adds to Vue's reactivity system all the properties found in its data object(described in table 2.6). All `Vue.js` internal modules and external plugins to be used are imported in the (`main.js`) file with ES6(ECMAScript 2015) import syntax ('import'). The dependencies are defined as global parameters and are accessible by all components. In the main.js file, the main Vue component (`App.vue`) is registered.

```
import Vue from 'vue'
import App from './App'
import router from './router'

```

```
// Root Instance
new Vue({
     el: '#app',
     router,
     template: '<App/>',
     components: { App }
})
```

Listing 2.3: main.js

2.6.5 Vue Component

The most important criterion for choosing this framework (`Vue.js`) is the modularity of the individual components. The components are one of the strongest features of `Vue.js`. They allow basic HTML elements to be extended to encapsulate reusable code. At a high level, components are custom elements to which the `Vue.js` compiler attaches a specific behavior [17]. Each component is defined in a file with `.vue` extension. The definition contains an HTML-based template that specifies the design of the component and the JavaScript code that specifies all other functionality or behavior of the component. Each component is also a Vue instance.

Typically, a Vue.js app is organized in a tree of nested components. For example, it consists of components for a header, sidebar, and content area, each containing other components for navigation links, blog posts, etc. The component architecture simplifies such nesting.

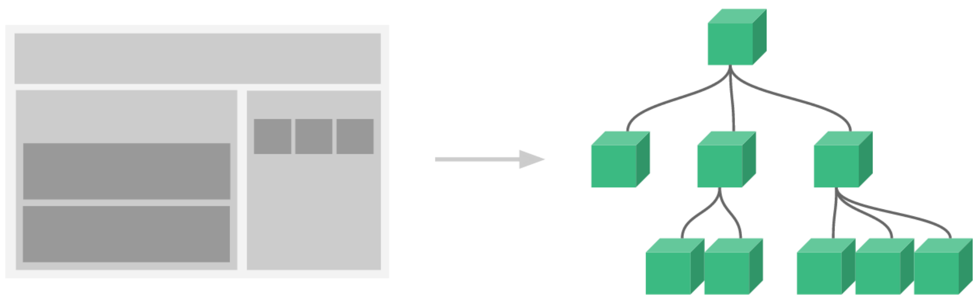


Figure 2.9: *Organization of Componenten* [23]

The template element contains the HTML template and the script element contains the JavaScript part of the component.

Template element: The design of the user interface is implemented in this element. Standard HTML and CSS technology are used for this. User-defined HTML elements, as well as HTML elements from external plug-ins, can be integrated very easily. The HTML-based template syntax of Vue.js allows binding the rendered DOM declaratively to the data of the underlying Vue instance. All Vue. js templates are valid HTML.

The simplest form of data binding is text interpolation using the *mustache* syntax (double curly braces):

```
<template>
    <div>
        <h1>Data: {{ beispielData }}</h1>
        <BeispielComponent/>
    </div>
</template>
```

Listing 2.4: Templateelement

Script element: This element contains the implementation of the desired functionalities of the component with `VueJS` specific construct. Further components can be imported and registered here. These are then available in the template element, which allows the nesting of components to be realized.

```

        <script>
          import BeispielComponent from 'components/BeispielComponent'
          export default {
            components: {
              BeispielComponent
            },
            props : [],
            data () {
              return {
                beispielData: 'Beispiel␣String'
              }
            },
            methods: {
                beispielFunction: function () { }
            },
            computed: {
            }
          }
        </script>
```

Listing 2.5: Skriptelement

The most important parts of a component are described in the table 2.6.

Table 2.6: Parameters responsible for data manipulation

Parameters	Type	Description
components	[key: string]: Object	This is a list of custom components to be used. The custom or external components are imported with `import` command to import them from the ES6 specification.
data	Object or Function	This is a JavaScript object or function that allows the storage of the necessary attributes for the component. Within the component, the original data object can be accessed with `this.data` directive can be accessed.
props	Array$< string >$ or Object	This is a list/hash of attributes intended to hold data from the parent component. It has a simple array-based syntax and an alternative object-based syntax that allows advanced configurations such as type checking, custom validation and default values.
methods	[key: string]: Function	This is a list of methods that allow the realisation of different functionalities of the component. All methods can be called within the component as a standard JavaScript method.
computed	[key: string]: Function	This is a list of methods that will be called automatically when the reactive dependencies change. The attributes within these parameters are cached and only recomputed on reactive dependency changes.

2.6.6 Instance lifecycle hooks

When created, each Vue instance goes through a series of initialization steps, e.g. it needs to set up data observation, compile the template, including the instance in the DOM and update the DOM when the data changes. In parallel, it also runs functions called lifecycle hooks that allow users to add their functions at certain stages of the instance lifecycle. This allows very precise control of the behavior of individual components.

For example, the *created* hook can be used to execute code after an instance has been created. Other hooks are called at different stages of the instance lifecycle, such as. *mounted*, *updated* and *destroyed*. The figure 2.10 illustrates the details of the complete lifecycle hook.

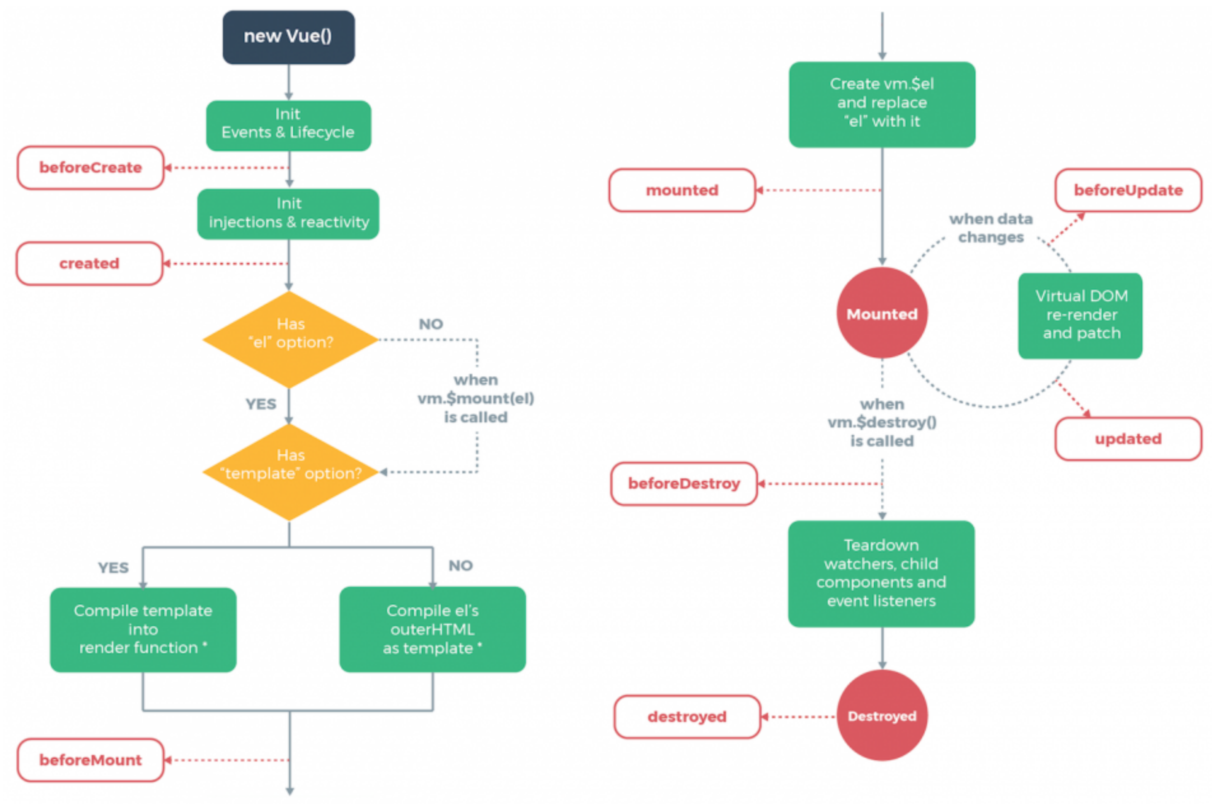

new Vue()
Init Events & Lifecycle
beforeCreate
Init injections & reactivity
created
Has "el" option?
NO
when vm.$mount(el) is called
YES
Has "template" option?
YES
NO
Compile template into render function *
Compile el's outerHTML as template *
beforeMount
Create vm.$el and replace "el" with it
mounted
beforeUpdate
when data changes
Mounted
Virtual DOM re-render and patch
updated
when vm.$destroy() is called
beforeDestroy
Teardown watchers, child components and event listeners
destroyed
Destroyed

Figure 2.10: *Vue Instance-Lifecycle Hooks* [24]

2.6.7 Vuex

Vuex is the official state management library for Vue. Its job is to exchange data between components in a Vue application.

Components in a Vue application can have their state. For example, an input field stores the data entered in it locally. Components can be input fields or a whole page. It becomes very costly to exchange state data between nested components.

Vuex provides central storage(Vuex store) for the state, and the state can be changed by calling functions defined in the Vuex store. Each component that depends on a particular piece of state accesses it via a getter function on the Vuex store, ensuring synchronization of data. Vuex-Store is a kind of temporary database for a particular application session [18].

2.6.8 Vue Router

In a web application, a router is a part that synchronizes the currently displayed view with the contents of the browser address bar.

A router is needed when Uniform Resource Locator (URL)s need to be synchronized with the views in an application. This is a very common need, and all major modern frameworks now allow routing to be managed. Vue Router is part of the Vue core library and handles route management in a Vue application [18].

2.7 Nginx

Nginx, pronounced like "engine-ex", is an open-source web server that is now also used as a reverse proxy, HTTP cache, and load balancer. It is one of the most used web servers along with the Apache HTTP server.[25]

This section describes the basics of NGINX (start, stop reload configuration), the structure of the configuration file and describes the process of setting up NGINX to serve out static content.

2.7.1 Starting, Stopping, and Reloading Configuration

Nginx has a single master process and several worker processes. The master process reads and evaluates the configuration and manages the worker processes. The worker processes operate the real processing of HTTP requests. Nginx uses an event-based model and operating system-dependent means to efficiently allocate requests among the worker processes. The number of worker processes is set in the configuration file.[25]

2.7.2 Structure of Configuration File

The configuration file is named by default nginx.conf and placed in the directory *"/usr/local/nginx/conf", "/etc/nginx", or "/usr/local/etc/nginx"*. Nginx consists of modules that are controlled by directives defined in the configuration file. Directives are split into simple directives and block directives. A simple directive is made up of the name and parameters separated by spaces and ends with a semicolon (;). A block directive ends with a sequence of additional arguments surrounded by curly braces. A block directive that contains other directives inside curly braces, is called a context (examples: events, http, server, and location). Directives that are outside contexts in the configuration file are in the main context. The directives events and http are in the main context. The server directive is inside the http directive and location inside the server. For comments, a hashtag (#) sign is used.[25]

2.7.3 Serving Static Content

3 Previous Work

Masterarbeit von Christoph (Für DDS und Routing service)

4 Design and Implementation

This chapter describes the Design process and the details of the realised implementation of a Docker based application developed for managing ROS2 Nodes in a local Network over a web interface. Figure 4.1 provides an overview of the Structure of the developed Application. The Application is supposed to have control over all the compatible ROS2 Applications running within a defined Network. It utilizes the ROS2 DDS System (for communication within same LAN) and ROS2 Routing Service (for communication between different Networks). It consists of a customized ROS2 based Docker Containers within which individual ROS2 Apps run and another customized Docker Container running a NodeJS based Web-Application (Lifecycle Management Dashboard).

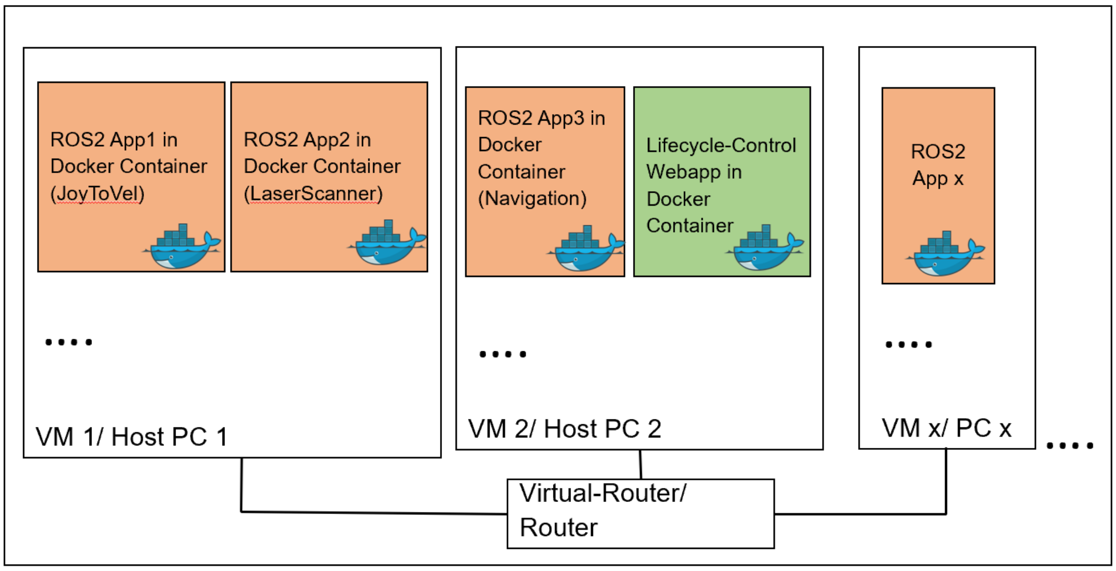

ROS2 App1 in Docker Container (JoyToVel)
ROS2 App2 in Docker Container (LaserScanner)
....
VM 1/ Host PC 1
ROS2 App3 in Docker Container (Navigation)
Lifecycle-Control Webapp in Docker Container
....
VM 2/ Host PC 2
ROS2 App x
....
VM x/ PC x
....
Virtual-Router/ Router

Figure 4.1: Structure of Lifecycle Management Application

The application consists of following major parts:

- The Docker environment
- ROS2 Lifecycle Application
- Vue based Webinterface
- Integration

They will be explained in detail in the following section.

4.1 The Docker environment

In order to develop an Industrial Edge Application, its necessary to pack an application into a Docker Image. Then it can be converted to an IE App using the Industrial Edge App Publisher.

This section describes the Docker image developed for the purposes of this thesis. A Dockerfile is used to define a Docker image. It consists of all the necessary software libraries and third party applications and precise order in which they have to installed. The following section will describe the details of the Dockerfile(structre of the Docker image) step by step.

4.1.1 The Dockerfile

Settng up the Base-Image This is the first line in any Dockerfile. In this line the Ubuntu Ros Image to use is specified.

```
FROM ros:foxy-ros-core-focal
```

Installing Bootstrap Tools This step installs the Bootstrap Tools(apt-utils, build-essentials, nano, git, curl, tmux) and necessary python3 extensions (python3-colcon-common-extensions, python3-colcon-mixin, python3-rosdep, python3-vcstool)

```
# install bootstrap tools
RUN apt-get update && apt-get install -y
   --no-install-recommends \
        apt-utils \
        build-essential \
        nano \
        git \
        curl \
        tmux \
        inetutils-ping \
        python3-colcon-common-extensions \
        python3-colcon-mixin \
        python3-rosdep \
        python3-vcstool \
        && rm -rf /var/lib/apt/lists/*
```

Setting up Ros Build environment The rosdep is initialized and updated before installing any ROS2 Package. Then colcon mixin and metadata are setup and any relataed libraries are updated. Colcon is the default and recommended build system for ROS2. Now ROS2 applications can be built using colcon in the next step.

```
# bootstrap rosdep
RUN rosdep init && \
rosdep update --rosdistro $ROS_DISTRO
# setup colcon mixin and metadata
RUN colcon mixin add default \
        https://raw.githubusercontent.com/colcon/
        colcon-mixin-repository/master/index.yaml && \
        colcon mixin update && \
        colcon metadata add default \
        https://raw.githubusercontent.com/colcon/
        colcon-metadata-repository/master/index.yaml && \
        colcon metadata update
```

Installimg necessary Ros packages Following ROS2 Packages are instllen in this step: ros-foxy-ros-base=0.9.2-1*, ros-foxy-joy, ros-foxy-diagnostic-updater,ros-foxy-lifecycle. Depending on the type of ROS2 Appliation it might be necessary to install further Packages, which should be done in this to avoid furthr complications. Finally the system is updated to fix any broken installs and install any mising libraries.

```
# install ros2 packages
RUN apt-get update && apt-get install -y
   --no-install-recommends \
        ros-foxy-ros-base=0.9.2-1* \
        ros-foxy-joy \
        ros-foxy-diagnostic-updater \
        # install for lifecycle
        ros-foxy-lifecycle \
        nginx \
        python \
# update system
        && apt-get upgrade -y \
        && rm -rf /var/lib/apt/lists/*
```

Sourcing Ros2 Sourcing is a common term among UNIX users. In order to make the desired changes applicable to the current shell , it is necessary to "source" the file. Furthermore adding sources to .bashrc files preconfigures any new shell opened. In this step ROS2 setup files are sourced so that other ROS2 commands can be run from the command line.

```
#add sources to bashrc
RUN echo "source␣/opt/ros/foxy/setup.bash" >> ~/.bashrc \
        && echo "source␣
           /root/colcon_ws/install/setup.bash" >>
           ~/.bashrc
```

Create and build Ros Workspace In this step first of all the build workspace is setup, then the necessary source files are copied from external working directory to the Docker internal workspace and finally built using colcon.

```
# create and build workspace
ENV ROS2_WS /root/colcon_ws

COPY ./evo_siemensrob_ctrl
   ${ROS2_WS}/src/evo_siemensrob_ctrl/.
COPY ./joy_converter ${ROS2_WS}/src/joy_converter/.
COPY ./joytovel ${ROS2_WS}/src/joytovel/.
COPY ./twist_mux ${ROS2_WS}/src/twist_mux/.

RUN cd ${ROS2_WS} \
        && . /opt/ros/foxy/setup.sh \
        && colcon build

```

```
# install RIB support
COPY ./RIBInstall .

RUN ./RIBInstall \
        && rm -rf RIBInstall
```

Setup Node enviroment for Vue This step installs a version of NodeJS in this Docker environment. NodeJS will be necessary to build the Vue-application and run ros2-web-bridge in forthcoming steps.

```
# For Lifecycle-Dashboard Vue App
ENV NODE_VERSION=12.6.0
# Already installed RUN apt install -y curl
RUN curl -o- https://raw.githubusercontent.com/
creationix/nvm/v0.34.0/install.sh | bash
ENV NVM_DIR=/root/.nvm
RUN . "$NVM_DIR/nvm.sh" && nvm install ${NODE_VERSION}
RUN . "$NVM_DIR/nvm.sh" && nvm use v${NODE_VERSION}
RUN . "$NVM_DIR/nvm.sh" && nvm alias default
    v${NODE_VERSION}
ENV PATH="/root/.nvm/versions/node/v${NODE_VERSION}/bin/
:${PATH}"
RUN node --version
RUN npm --version
```

Copy package.json files to install Node Project dependencies A package.json file is a standard package specification file for NodeJS applications. It contains all the necessary configurations and list of packages to successfully build a NodeJS aplication. In this step a working directory for the Vue Application is created, the the package.json is copied to the Docker internal workdirectory and finally the project depencencies are downloaded and installed.

```
# build stage
WORKDIR /app
COPY ./lifecycle-dashboard/package*.json ./
RUN npm install
```

Build the web app for production The project source files for the "Lifecycle management application" are copied from external workspace to the Docker workdirectory and built for production.

```
# build stage
COPY ./lifecycle-dashboard/ .
RUN npm run build
```

Setting up a Nginx web server In this step the nginx configuration file (described in section 4.1.3) is copied from external workspace to the docker enviroment. The built application consisting of only .html, .css and .js files are copid to a location where nginx can easily find them and use them to start the web-application. Furthermore PORT 80 is exposed and the web-application can thus be accessed on locahost:8080 within the same network where this Docker container is running.

```
# For nginx
COPY ./nginx.conf /etc/nginx/nginx.conf
RUN cp -a /app/dist/. /var/www/html
EXPOSE 80
```

Installing ros2-web-bridge In this step ros2-web-bridge is installed and other services that need to launch at startup are setup. The web-server(nginx) is setup to start at launch.

```

# install ros2-web-bridge

# set workdirectory as /root
WORKDIR /root

#for automatic launch when container gets started
COPY ./dockerfile_startup.sh dockerfile_startup.sh
RUN ["chmod", "+x", "./dockerfile_startup.sh"]
CMD ./dockerfile_startup.sh
```

Starting Ros Application This step launches the installed ROS2-Application using its launchfile.

```
#!/bin/bash

nginx
. /opt/ros/foxy/setup.sh
. /root/colcon_ws/install/setup.sh
npm install ros2-web-bridge
node node_modules/ros2-web-bridge/bin/rosbridge.js &
ros2 launch evo_siemensrob_ctrl agv_control_launch.py
```

Startup and Build scripts Hh

```
#!/bin/bash
docker start agv_control_joy_rti

docker exec -it agv_control_joy_rti bash
```

4.1.2 Running multiple services in a container

The most important running process of a container is the ENTRYPOINT and/or CMD at the end of the Dockerfile. It is generally recommended to separate areas of concern by using one service per container. This service can split into several processes (e.g. the Apache webserver starts several working processes). It is fine to have multiple processes but to get the most benefit from Docker, you should avoid having one container responsible for multiple aspects of your entire application. You can connect multiple containers via custom networks and shared volumes.

The main container process is responsible for managing all the processes it launches. In some cases, the main process is not well thought out and cannot properly terminate child processes when the container is terminated (reaping). If your process falls into this category, you can use the –init option when starting the container. The –init option adds a tiny init process as the main process in the container and takes care of reaping all processes when the container is terminated. Handling such processes is better than using a full-fledged init process like sysvinit, upstart, or systemd to manage the process lifecycle inside your container.

```
#!/bin/bash

# turn on bash's job control
set -m

# Start the primary process and put it in the background
./my_main_process &

# Start the helper process
./my_helper_process

# the my_helper_process might need to know how to wait on the
# primary process to start before it does its work and returns


# now we bring the primary process back into the foreground
# and leave it there
fg %
```

```
# syntax=docker/dockerfile:1
FROM ubuntu:latest
COPY my_first_process my_first_process
COPY my_second_process my_second_process
COPY my_wrapper_script.sh my_wrapper_script.sh
CMD ./my_wrapper_script.sh
```

https://docs.docker.com/config/containers/multi-service_container/

4.1.3 Nginx server configuration

Configuring NGINX as a web server is about defining which URLs it processes and how it processes HTTP requests for resources at those URLs. On a deeper level, the configuration defines a set of virtual servers that control the processing of requests for specific domains or IP addresses.

Since the application in this thesis is only supposed to run within a local network (not the internet) the webserver configuration is drastically simplified. The basic HTTP settings can be left to the default configuration. Only the virtual server settings need to be configured. This section describes the configuration used for NGINX to serve static content. This includes the definition of paths searched to find requested files and set up for index files. The following listing shows the most important configurations needed to serve the developed web application on a local network.

```
http {
        # Basic Settings
        # ...
        server {
                # listen 80 default_server;
                # listen [::]:80 default_server;

                listen       80;
                listen   [::]:80;
                server_name  localhost;

                index index.html index.htm
                   index.nginx-debian.html;

                location / {
                        # First attempt to serve request
                           as file, then
                        # as directory, then fall back to
                           displaying a 404.
                        root    /usr/share/nginx/html;
                        index   index.html index.htm;
                        }
                }
        }
```

Listing 4.1: Nginx configuration file (nginx.conf)

A virtual server is defined by a server directive in the http context in `nginx.conf` file. The used directives in the above configuration file are described below:

listen The server configuration block usually includes a listen directive to specify the IP address and port on which the server listens for requests. Both IPv4 and IPv6 addresses are accepted; IPv6 addresses are enclosed in square brackets. In the above configuration the server listens on IP address 127.0.0.1 (localhost) and port 80.

server name If there are several servers that match the IP address and port of the request, the request's Host header field is tested against the `server_name` directives in the server blocks. The parameter to `server_name` can be a full name, a wildcard, or a regular expression. Since the above configuration has just one virtual server the this parameter is set to localhost.

root The root directive specifies the root directory to be used to search for a file. To obtain the path of a requested file, NGINX appends the request URI to the path specified by the root directive. The directive can be placed at any level within the http , server or location contexts. In case the root directive is defined for a virtual server. It applies to all location blocks that do not contain the root directive to explicitly redefine the root.[26]

index If a request ends with a slash, NGINX treats it as a request for a directory and tries to find an index file in that directory. The index directive defines the name of the index file (the default is index.html). If the URI of the request is `/some/path/`, based on the above configuration NGINX will return the file `/usr/share/nginx/html/some/path/index.html` if it exists. If it does not, NGINX returns the HTTP code 404 (Not Found) by default. Multiple file names can be listed in the index directive. In this case, NGINX searches for the files in the specified order and returns the first file found.

4.2 ROS2 Lifecycle Application

In oder to showcase the capabilities and possible use-cases of the managed lifecycle nodes in ROS2 ecosystem a demonstration application was developed. This appliaction has multiple states which allows for reconfiguration and respawning of the node while it is running. It also is a key part for demonstrating the capabilities of the remote lifecycle management interface described in section 4.3. This section describes the details of an example ROS2-Application with managed states and steps necessary to convert a normal ROS2-node into a managed lifecycle node.

4.2.1 Structure of a normal ROS2 publisher node

To demonstrate the process of developing a managed lifecycle node a JoyToVel publisher node is converted into a lifecycle node in this section. The JoyToVel publisher takes input from a PlayStation4 Controller and publishes velocity and direction commands to `\cmd_vel` topic in a ROS2 Network. Any application(node) subscribed to this topic can be controlled using this JoyToVel node.

A normal JoyToVel publisher consists of a constructor, application-specific callback functions and methods, and the main function. The necessary subscribers and publishers are initialized in the constructor. It also includes the application logic. In the main function, the node is started with the specified parameters. No changes to any parameters can be made once the node has been started. If the node crashes for some reason it has to be respawned manually. Since the JoyToVel node is an interface node, it would be beneficial to be able to dynamically reconfigure its different parameters and it demands higher reliability and stability.

A normal ROS2 C++ publisher node has the following structure:

```
#include "JoyToVel.hpp"
namespace evo {
     //Constructor
     JoyToVel::JoyToVel() : Node("joy_to_vel")
     {
          // init subscribers
          ...
          // init publishers
          ...
     }

     //Callbacks and Methods
```

```
    void JoyToVel::topic_callback(const sensor_msgs::msg::Joy::
        SharedPtr msg) const
    {
        ...
    }

    void JoyToVel::timer_callback()
    {
        ...
    }

    void JoyToVel::publishVel(const sensor_msgs::msg::Joy::
        SharedPtr msg)
    {
        ...
    }
}

int main(int argc, char** argv)
{
    rclcpp::init(argc, argv);

    rclcpp::spin(std::make_shared<evo::JoyToVel>());
    rclcpp::shutdown();

    return 0;
}
```

4.2.2 Structure of a ROS2 publisher node with managed lifecycle

A lifecycle based ros2 publisher has following structure:

```
namespace evo {
  class LifecycleTalker : public rclcpp_lifecycle::LifecycleNode
  {
  public:
    /// LifecycleTalker constructor
    explicit LifecycleTalker(const std::string &node_name, bool
        intra_process_comms = false)
        : rclcpp_lifecycle::LifecycleNode(node_name,
                                          rclcpp::NodeOptions().
                                              use_intra_process_comms
                                              (intra_process_comms))
    {
      // init subscribers
      ....
      // init publishers
      ....
    }

    /// Transition callback for state configuring
    rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::
        CallbackReturn
    on_configure(const rclcpp_lifecycle::State &)
    {
      // Initialize and configure publishers and timers.
      pub_ = this->create_publisher<std_msgs::msg::String>("lifecycle_chatter", 10);
       timer_ = this->create_wall_timer(
          1s, std::bind(&evo::LifecycleTalker::timer_callback, this));

      return rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn::SUCCESS;
    }

    /// Transition callback for state activating
    rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
    on_activate(const rclcpp_lifecycle::State &)
    {
      // Activate the lifecycle publisher.
      pub_->on_activate();

      // Important tasks in the activating phase.
        ....

      return rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn::SUCCESS;
    }

    /// Transition callback for state deactivating
    rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
    on_deactivate(const rclcpp_lifecycle::State &)
    {
      // Deactivate the lifecycle publisher.
      pub_->on_deactivate();

      return rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn::SUCCESS;
    }

    /// Transition callback for state cleaningup
    rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
    on_cleanup(const rclcpp_lifecycle::State &)
    {
      // Release the shared pointers to the timer and publisher.
      timer_.reset();
      pub_.reset();

      return rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn::SUCCESS;
    }

    /// Transition callback for state shutting down
    rclcpp_lifecycle::node_interfaces::LifecycleNodeInterface::CallbackReturn
    on_shutdown(const rclcpp_lifecycle::State &state)
    {
      // Tasks to be performed in shutdown phase
      ....

```

```
      return rclcpp_lifecycle::node_interfaces::
          LifecycleNodeInterface::CallbackReturn::SUCCESS;
    }

     // Publish method for the publisher
     void publish()
    {
      ...
    }


  private:
    // This is an instance of a lifecycle publisher. This lifecycle
       publisher
    // can be activated or deactivated based on state of the
       lifecycle node
    std::shared_ptr<rclcpp_lifecycle::LifecyclePublisher<std_msgs::
       msg::String>> pub_;

  };

  // Application specific Callbacks/Methods
    void evo::LifecycleTalker::topic_callback(const sensor_msgs::msg
       ::Joy::SharedPtr msg) const
    {
          ....
    }

    void evo::LifecycleTalker::timer_callback()
    {
          ....
    }

    void evo::LifecycleTalker::joyCallback(const sensor_msgs::msg::
       Joy::SharedPtr msg)
    {
          ....
    }

    void evo::LifecycleTalker::publishVel(const sensor_msgs::msg::
       Joy::SharedPtr msg)
    {
          ....
    }
}

int main(int argc, char** argv)
{
  setvbuf(stdout, NULL, _IONBF, BUFSIZ);
  rclcpp::init(argc, argv);

  rclcpp::executors::SingleThreadedExecutor exe;
  std::shared_ptr<evo::LifecycleTalker> lc_node = std::make_shared<
      evo::LifecycleTalker>("lifecycle_joytovel");

  exe.add_node(lc_node->get_node_base_interface());
  exe.spin();
  rclcpp::shutdown();

  return 0;
}
```

4.2.3 Conversion of a node into a Lifecycle Node

In order to convert a normal ROS2 node into a lifecycle node following steps need to be taken:

- Import necessary libraries. Following libraries are necessary for a lifecycle node. In order to import these libraries the package `rclcpp_lifecycle and lifecycle_msgs` must be installed on the system.

```
#include <chrono>
#include <thread>
#include <utility>
#include <numeric>

#include "lifecycle_msgs/msg/transition.hpp"
#include "rclcpp/rclcpp.hpp"
#include "rclcpp/publisher.hpp"
#include "rclcpp_lifecycle/lifecycle_node.hpp"
#include "rclcpp_lifecycle/lifecycle_publisher.hpp"

#include "rcutils/logging_macros.h"
```

- Encapsulate the node in a class that inherits from class `rclcpp_lifecycle::LifecycleNode` This brings in a series of callbacks that are called depending on the current state of the node. Each lifecycle node has a set of services associated with it that make it controllable from the outside.
- Initialize the public and private methods.
- Configure and implement transition callbacks listed in listing 2.1.
- Define appliation specific methods
- Perform necessary setup and initialization in main() function.

4.2.4 A multi node Applicaton

A realistic robot consists of several independent components which need to run simultaneously and interact with each other. In order to simulate a system with multiple managed nodes an example multi node ROS2 Application is developed in this section. For demonstration purposes three nodes which depend on each other during different states of their lifecycle is developed.

4.3 Vue based Webinterface

The developed web interface is supposed to be a web version of the lifecycle management CLI.

4.3.1 Web Application Requirements

4.3.2 Web Application Design

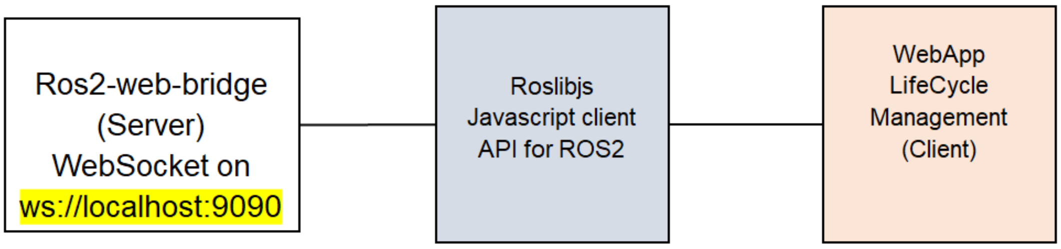


Figure 4.2: Overview of Lifecycle Management WebApp

4.3.3 ROS2 Message Types/ Javascript Objects

This section describes the ROS2 Messages and Services used to communicate with the ROS2 backend application. There was also a need to explore and examine various JavaScript counterparts to these message and service types. Messages and services used in the developed application are listed below with their definitions.

Messages:

- `lifecycle_msgs/msg/State`

```
# State definitions
uint8 PRIMARY_STATE_UNKNOWN = 0
uint8 PRIMARY_STATE_UNCONFIGURED = 1
uint8 PRIMARY_STATE_INACTIVE = 2
uint8 PRIMARY_STATE_ACTIVE = 3
uint8 PRIMARY_STATE_FINALIZED = 4

uint8 TRANSITION_STATE_CONFIGURING = 10
uint8 TRANSITION_STATE_CLEANINGUP = 11
uint8 TRANSITION_STATE_SHUTTINGDOWN = 12
uint8 TRANSITION_STATE_ACTIVATING = 13
uint8 TRANSITION_STATE_DEACTIVATING = 14
uint8 TRANSITION_STATE_ERRORPROCESSING = 15

## Fields

# The state id value from the above definitions.
uint8 id
# A text label of the state.
string label
```

Listing 4.2: Message definition State

- `lifecycle_msgs/msg/Transition`

```
# Transition id definitions
uint8 TRANSITION_CREATE = 0
uint8 TRANSITION_CONFIGURE = 1
uint8 TRANSITION_CLEANUP = 2
uint8 TRANSITION_ACTIVATE = 3
uint8 TRANSITION_DEACTIVATE = 4
uint8 TRANSITION_UNCONFIGURED_SHUTDOWN  = 5
uint8 TRANSITION_INACTIVE_SHUTDOWN = 6
uint8 TRANSITION_ACTIVE_SHUTDOWN = 7
uint8 TRANSITION_DESTROY = 8
uint8 TRANSITION_ON_CONFIGURE_SUCCESS = 10
uint8 TRANSITION_ON_CONFIGURE_FAILURE = 11
uint8 TRANSITION_ON_CONFIGURE_ERROR = 12
uint8 TRANSITION_ON_CLEANUP_SUCCESS = 20
uint8 TRANSITION_ON_CLEANUP_FAILURE = 21
uint8 TRANSITION_ON_CLEANUP_ERROR = 22
uint8 TRANSITION_ON_ACTIVATE_SUCCESS = 30
uint8 TRANSITION_ON_ACTIVATE_FAILURE = 31
uint8 TRANSITION_ON_ACTIVATE_ERROR = 32
uint8 TRANSITION_ON_DEACTIVATE_SUCCESS = 40
uint8 TRANSITION_ON_DEACTIVATE_FAILURE = 41
uint8 TRANSITION_ON_DEACTIVATE_ERROR = 42
uint8 TRANSITION_ON_SHUTDOWN_SUCCESS = 50
uint8 TRANSITION_ON_SHUTDOWN_FAILURE = 51
uint8 TRANSITION_ON_SHUTDOWN_ERROR = 52
uint8 TRANSITION_ON_ERROR_SUCCESS = 60
uint8 TRANSITION_ON_ERROR_FAILURE = 61
uint8 TRANSITION_ON_ERROR_ERROR = 62
uint8 TRANSITION_CALLBACK_SUCCESS = 97
uint8 TRANSITION_CALLBACK_FAILURE = 98
uint8 TRANSITION_CALLBACK_ERROR = 99

## Fields

# The transition id from above definitions.
uint8 id
# A text label of the transition.
string label
```

Listing 4.3: Message definition Transition

- `lifecycle_msgs/msg/TransitionDescription`

```
# The transition id and label of this description.
```

```
Transition transition
# The current state from which this transition
   transitions.
State start_state
# The desired target state of this transition.
State goal_state
```

Listing 4.4: Message definition TransitionDescription

- `lifecycle_msgs/msg/TransitionEvent`

```
# The time point at which this event occurred.
uint64 timestamp
# The id and label of this transition event.
Transition transition
# The starting state from which this event
   transitioned.
State start_state
# The end state of this transition event.
State goal_state
```

Listing 4.5: Message definition TransitionEvent

- `rcl_interfaces/msg/Log`

```
##
## Severity level constants
##
## Since there are several other logging enumeration
   standard for different implementations,
## other logging implementations may need to provide
   level mappings to match their internal
   implementations.
##
# Debug is for pedantic information, which is useful
   when debugging issues.
byte DEBUG=10
# Info is the standard informational level and is
   used to report expected
# information.
byte INFO=20
# Warning is for information that may potentially
   cause issues or possibly unexpected
# behavior.
byte WARN=30
# Error is for information that this node cannot
   resolve.
byte ERROR=40
# Information about a impending node shutdown.
byte FATAL=50
```

```
## Fields

# Timestamp when this message was generated by the
    node.
builtin_interfaces/Time stamp
# Corresponding log level, see above definitions.
uint8 level
# The name representing the logger this message came
    from.
string name
# The full log message.
string msg
# The file the message came from.
string file
# The function the message came from.
string function
# The line in the file the message came from.
uint32 line
```

Listing 4.6: Message definition Log

Services:

- `lifecycle_msgs/srv/ChangeState` A ROS2 Service has the following structure:

```
# The requested transition.
Transition transition
---
# Indicates whether the state transition was
    successful
bool success
```

Listing 4.7: Service definition ChangeState

- `lifecycle_msgs/srv/GetAvailableStates`

```
---
# Array of possible states that can be transitioned
    to.
State[] available_states
```

Listing 4.8: Service definition GetAvailableStates

- `lifecycle_msgs/srv/GetAvailableTransitions`

```
---
# An array of the possible start_state-goal_state
    transitions
TransitionDescription[] available_transitions
```

Listing 4.9: Service definition GetAvailableTransitions

- `lifecycle_msgs/srv/GetState`

```
---
# The current state-machine state of the node.
State current_state
```

Listing 4.10: Service definition GetState

ROSLIB.ServiceRequest ROSLIB.Service ROSLIB.Topic

JavaScript Objects

- ROSLIB.Service

```
{
        ros : this.ros,
        name : '/lifecycle_node/get_state',
        serviceType : 'lifecycle_msgs/GetState'
}
```

- ROSLIB.ServiceRequest

```
{
        requestParam1 : {requestObject1},
        requestParam2 : {requestObject2},
        ...

}
```

Listing 4.11: JavaScript object definition for ServiceRequest

- ROSLIB.Topic

```
{
        ros : this.ros,
        name : '/rosout',
        messageType : 'rcl_interfaces/msg/Log'
}
```

Listing 4.12: JavaScript object definition for Topic

JSON Objects A Javascript application uses JSON Objects to communicate over a network. A JSON object is returned when a ROS service is called or a topic is subscribed to. It is necessary to parse and process these objects to display or update information on a web page. To parse a JSON object, prior knowledge of its structure is required.

JSON (JavaScript Object Notation) is a lightweight format for data exchange. Working with it is easy for humans likewise machines. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that does not depend on any specific language but uses conventions familiar to the C language family, including `C`, `C++`, `C#`, `Java`, `JavaScript`, `Perl`, `Python`, *and many others. These* *properties make JSON an ideal language for data exchange.*[27]

In the following section JSON objects used by the developed application is described in detail.

- `current_state`

```
{
        "current_state": {
                "id": 1,
                "label": "unconfigured"
        }
}
```

Listing 4.13: JavaScript object definition for `state`

- `transition`

```
{
        "transition": {
                "id": 1,
                "label": "configure"
        },
        "start_state": {
                "id": 1,
                "label": "unconfigured"
        },
        "goal_state": {
                "id": 10,
                "label": "configuring"
        }
}
```

Listing 4.14: JavaScript object definition `transition`

- `available_transitions`

```
{
        "available_transitions": [
          {
                "transition": {
                  "id": 1,
                  "label": "configure"
                },
                "start_state": {
                  "id": 1,
                  "label": "unconfigured"
                },
                "goal_state": {
                  "id": 10,
                  "label": "configuring"
```

```
                        }
                 },
                 {
                        "transition": {
                          "id": 5,
                          "label": "shutdown"
                        },
                        "start_state": {
                          "id": 1,
                          "label": "unconfigured"
                        },
                        "goal_state": {
                          "id": 12,
                          "label": "shuttingdown"
                        }
                 }
               ]
   }
```

Listing 4.15: JavaScript object definition for `available_transitions`

- `available_states`

```
{
          "available_states": [
                    {
                    "id": 0,
                    "label": "unknown"
                    },
                    {
                    "id": 1,
                    "label": "unconfigured"
                    },
                    ...
          ]
}
```

Listing 4.16: JavaScript object definition `available_states`

4.3.4 Used libraries/ package.json

4.3.5 Implementing Automatic State Transitions

4.3.6 Code Overview

For developing the web appliaction a web framework called VueJS (described in section 2.6) is used. This allows the use of various pre-built html components and streamlined application structure with various plugins which in turn makes the UI development structured and easier. The UI part is encapsulatd inside the template tag and the JavaScript part in the script tag of a Vue component(described in section 2.6.5). The project structure described in 2.6.3 is used.

To secure a conection to the ROS2 backend through RosLibJS a ROSLIB.Ros Object needs to be initialized and assigned to a variable that can be accessed by all other functions in the application. This object consists of a url parameter, which will be used to assign the URL of the websocket. The websocket is made availabe by the library ros2-web-bridge. This is our main instance of RosLibJS and the same instance will be used throughout the application for working with a single Lifecycle Node. In case there are multiple Lifecycle nodes added, multiple instances of RosLibJS are created and stored. The details of the JavaScript part of the application is described below:

Library imports The libraries are imported using the standard ES6 syntax. The libraries including vue-plugins to be installed are listed in the package.json file. This import syntax allows the import of the entire library or only specific components of the library. The following libraries are used:

```
import * as ROSLIB from 'roslib';
import { MDBBtn, MDBBtnGroup, MDBInput, MDBContainer,
   MDBRow, MDBCol } from "mdb-vue-ui-kit";
```

Vue Component initialization To be able to use the imported components in the HTML-Layout they need to be initialized. The components to be used must be included within the component deirctive. Also the name of the component is assigned, this name will be used to import this component into any other component.

```
export default {

        name: 'LifecycleManagement',
        components: {
                MDBBtn, MDBBtnGroup, MDBInput,
                   MDBContainer, MDBRow, MDBCol
        },
        ....
}
```

Declaring global variables The data directive represents a storage of the necessary attributes for the component. All the global variables are declared in here.The original data object can be accessed with `this.data` directive anywhere inside the component.

```
data: function () {
        return {
                unsub: false,
                console_out: [],
                selected_logger_level: 20,
                input_node_name:"",
                added_nodes:["lifecycle_joytovel"],
                active_node: "lifecycle_joytovel",
                selected_transition: {},
                available_states: [],
                available_transitions: [],
                current_state: {},
                ros: new ROSLIB.Ros({
                        url : 'ws://localhost:9090'
                })
        }
},
```

Method definitions All the functions and methods necessary for the application are to be defined inside the methods directive. All the methods used in this application are described in the section 4.3.7.

```
methods: {
        ...
},
```

Vue instance lifecycle hook mounted A Vue component has several lifecycle hooks (described in section 2.6.6), which lets a developer customize its behaviour at any specific point in its lifecycle. In this application the mounted hook is used to establish a connection to the websocket created by the ROS2-Web-Bridge.

```
mounted: function () {
        var ros = new ROSLIB.Ros({
                url : 'ws://localhost:9090'
        });

        ros.on('connection', function() {
                console.log('Connected␣to␣websocket␣server.');
        });

        ros.on('error', function(error) {
                console.log('Error␣connecting␣to␣websocket␣server:␣', error);
        });

        ros.on('close', function() {
                console.log('Connection␣to␣websocket␣server␣closed.');
        });

        this.update();
},
```

4.3.7 Important Functions/Methods

Using RosLibJS the frontend application can be conected to the backend API(websocket) provided by ROS2-web-bridge. To get the required informaton about the ros2 nodes running inside the backend application several ROS2 services need to called and some ROS2 topics need to be subscribed to. The methods used to realize these functionalities are described below in detail. These methods are all included inside the *methods* directive described in previous section. As described in listing 4.3.3 a ROSLIB.Service object needs to be created everytime a service needs to be called. It has a name parameter which has the following syntax:

```
name: '/<name of the active node>/<name of the service>'
```

This convention is utilized in all the following methods in this section.

Methods to add lifecycle nodes and activate it The `add_node` method takes in the name of the lifecycle node provided through the UI and aappends it to an array. Multiple lifecycle nodes can be added to the application using this method. To activate a node the `set_active_node` method is called. This selects a node from an array of nodes and sets it as the active node. It also clears all the fields in the UI and updates it based on the data received for the active node. The methods for clearing and updateding the UI are described in listing 4.3.7.

```
add_node(){
  if (this.input_node_name !== "") {
        this.added_nodes.push(this.input_node_name);
        console.log(this.added_nodes);
  }
},

set_active_node(data){
        this.active_node = data;
        this.clear();
        this.update();
},
```

Service to get CurrentState This method calls a ROS service to get the current state of the active lifecycle node. A new service object is created, as parameters, the global RosLibJS instance, the complete name of the service, and the service type is passed. All the available services for a lifecycle node are listed in listing 2.2. According to the service definition in listing 4.10, a ROSLIB.ServiceRequest object is created. This ServiceRequest object is then passed as a parameter to the callService function of the Service object created in the previous step. The callService function takes three parameters:

```
serviceObject.callService(request, <result_handler>,
    <error_handler>)
```

The first parameter is the ServiceRequest object, second and third are JavaScript Arrow Functions to extract the result of the service and error message. The third parameter is optional. The structure of the result object is described in listing 4.13. The current state is extracted from the result object and the global currentState variable is updated.

```
srvGetCurrentState() {
        var lifecycleClient = new ROSLIB.Service({
                ros : this.ros,
                    name : `/${this.active_node}/get_state`,
                    serviceType : 'lifecycle_msgs/GetState'
        });

        var request = new ROSLIB.ServiceRequest({});

        lifecycleClient.callService(request,
                (result) => {
                        console.log(result);
                        this.current_state =
                           result.current_state;
                }
        )

        },
```

Service to get all available States This method calls a ROS service to get the all available states of the active lifecycle node. A new service object is created, similar to the last example. According to the service definition in listing 4.8, a ROSLIB.ServiceRequest object is created. The structure of the result object is described in listing 4.16. All available states are extracted from the result object, stored in a temporary array and the global availableStates array is updated.

```
srvGetAvailableStates() {
  var lifecycleClient = new ROSLIB.Service({
        ros : this.ros,
        name :
           `/${this.active_node}/get_available_states`,
        serviceType : 'lifecycle_msgs/GetAvailableStates'
  });

        var request = new ROSLIB.ServiceRequest({});
        var tempArr = [];

        lifecycleClient.callService(request,
          (result) => {
                console.log(result);
                result.available_states.forEach(element
                   => {
                        console.log(element);
                        tempArr.push(element);
                });
                this.available_states = tempArr;
        })
},
```

Listing 4.17: Method to get available States

Service to get AvailableTransitions This method calls a ROS service to get all available transitions of the active lifecycle node. A new service object is created, similar to the last example. According to the service definition in listing 4.9, a ROSLIB.ServiceRequest object is created. The structure of the result object is described in listing 4.15. All available transitions are extracted from the result object, stored in a temporary array and the global availableTransitions array is updated.

```
srvGetAvailableTransitions() {
  var lifecycleClient = new ROSLIB.Service({
        ros : this.ros,
        name :
           `/${this.active_node}/get_available_transitions`,
        serviceType :
           'lifecycle_msgs/GetAvailableTransitions'
  });

  var request = new ROSLIB.ServiceRequest({});
  var tempArr = [];

  lifecycleClient.callService(request,
        (result) => {
        result.available_transitions.forEach(
                element => {
                tempArr.push(element);
        });
        this.available_transitions = tempArr;
  })
},
```

Listing 4.18: Method to get available Transitions

Service to ChangeState This method calls a ROS service to change the current state of the active lifecycle node to any valid state. A new service object is created, similar to the last example. According to the service definition in listing 4.7, a ROSLIB.ServiceRequest object is created. The ServiceRequest object requires a transition parameter, which consist a transition ID and a label. Transition ID can be selected in the UI, the available selections in UI is determined by a automatic state transition logic (described in section **??**). The result in this case is a boolen, which indicates if the transition was succesfull. Then the update method is called.

```
srvChangeState() {
        var lifecycleClient = new ROSLIB.Service({
                ros : this.ros,
                name :
                   `/${this.active_node}/change_state`,
                // name :
                   '/lifecycle_joytovel/change_state',
                serviceType : 'lifecycle_msgs/ChangeState'
        });

        var r_id = parseInt(this.selected_transition);

        var request = new ROSLIB.ServiceRequest({
                transition: {id: r_id, label : ""}
        });

        lifecycleClient.callService(request,
           function(result) {
                console.log(result);
        }, function(error) {
                console.log(error);
        });
        this.update();
},
```

Subscribe to rosout This method is necessary to implement the logging functionality in the UI. A ROSLIB.Topic object is created, which is used to subscribe to ROS topic \rosout. The relevant log messages for the active lifecycle node is filtered and stored in a rolling array of a fixed size. This array is used by the logger component to disply log messages in the UI. Structure of the log message is described in listing 4.6. The desired log level (DEBUG, INFO, WARN, ERROR or FATAL) can be selected using the mapLogLevel method in listing 4.3.7. Using a global unsubscribe flag, the logging can be deactivated from the UI.

```
subscribeToRosOut(){
        var listener = new ROSLIB.Topic({
                ros : this.ros,
                name : '/rosout',
                messageType : 'rcl_interfaces/msg/Log'
                });

                var msg = [];
                listener.subscribe((msg)=> {
                        this.console_out.push(msg);
                        if (this.console_out.length > 10)
                           {
                                this.console_out.shift();
                        }

                        if (this.unsub) {
                                listener.unsubscribe();
                        }
                })
                console.log(msg);
        },
```

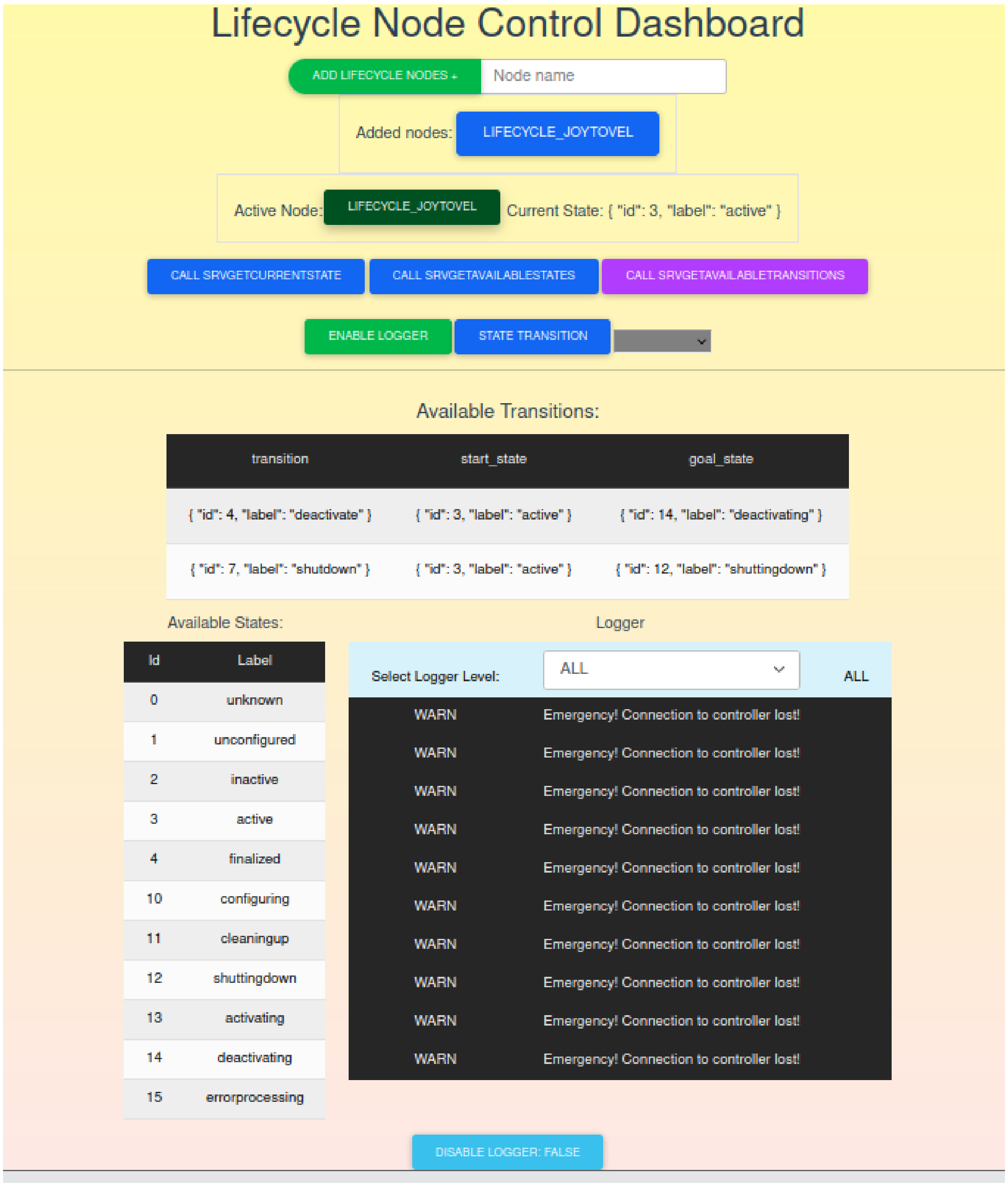
Auxillary Methods/Functions These are some auxillary methods/functions used in this Vue component.

```
clear() {
        this.available_states = [];
        this.current_state = [];
        this.available_transitions = [];
},

update() {
        this.srvGetCurrentState()
        this.srvGetAvailableTransitions()
        this.srvGetAvailableStates()
},

mapLogLevel(level){
        if (level === 10) {
                return "DEBUG"
        }
        else if (level === 20){
                return "INFO"
        }
        else if (level === 30){
                return "WARN"
        }
        else if (level === 40){
                return "ERROR"
        }
        else if (level === 10){
                return "FATAL"
        }
}
```

4.3.8 Lifecycle Dashboard UI



Added nodes: LIFECYCLE_JOYTOVEL

Active Node: LIFECYCLE_JOYTOVEL Current State: { "id": 3, "label": "active" }

Available Transitions:

transition	start_state	goal_state
{ "id": 4, "label": "deactivate" }	{ "id": 3, "label": "active" }	{ "id": 14, "label": "deactivating" }
{ "id": 7, "label": "shutdown" }	{ "id": 3, "label": "active" }	{ "id": 12, "label": "shuttingdown" }

Available States:

Id	Label
0	unknown
1	unconfigured
2	inactive
3	active
4	finalized
10	configuring
11	cleaningup
12	shuttingdown
13	activating
14	deactivating
15	errorprocessing

Logger

Select Logger Level: ALL ALL

```
WARN    Emergency! Connection to controller lost!
WARN    Emergency! Connection to controller lost!
WARN    Emergency! Connection to controller lost!
WARN    Emergency! Connection to controller lost!
WARN    Emergency! Connection to controller lost!
WARN    Emergency! Connection to controller lost!
WARN    Emergency! Connection to controller lost!
WARN    Emergency! Connection to controller lost!
WARN    Emergency! Connection to controller lost!
WARN    Emergency! Connection to controller lost!
```

Figure 4.3: Lifecycle Dashboard

4.4 Integration

4.4.1 Conversion to Industrial Edge App

The Industrial Edge Publisher can be used to convert a docker image into an Industrial Edge Application. The created docker-compose.yml file (listing 4.19) is first imported into the Industrial Edge Publisher. The created images along with their various configurations are recognized by the Industrial Edge environment and these configurations are automatically applied when the app is installed on the Edge Device. No further configuration is necessary for the Edge App. Once the app has been created, it can be uploaded to the IEM using the Industrial Edge Publisher, where it is available for further distribution.

Docker Compose File With a docker-compose.yml file, services and configurations can be specified for a Docker container. This application requires several Docker containers to communicate with each other, for this purpose a seperate network is created in the docker-compose file. In addition, a volume is created to store any persistent data that might be useful for the application.

In the services section of the docker-compose file, the Docker image in the current directory is built. The name of the created Docker image is then specified. Then with the start command a ROS2 Application with lifecycle node is run. The commands `mem_limit: 500mb` and `restart: on-failure` are necessary for Industrial Edge. Then the docker volume created in the previous step is mounted. The network to use is specified then the docker container is connected to the network.

```
networks :
         ie-app-network :
                 name : ie-app-network

volumes:
        lifecycle_edge:
                name: lifecycle_edge

services:
        ros2-lifecycle:
                build: ./
                image: ros2_foxy_lifecycle:1.0
                command: ros2 launch evo_siemensrob_ctrl
                   agv_control_launch.py
                mem_limit: 500mb
                restart: on-failure
                volumes:
                        - lifecycle_edge
                networks:
                        - ie-app-network
```

Listing 4.19: docker-compose.yaml used by IE Publisher

5 Test setup

To test the Lifecycle Management Interface and showcase a Use case for the Managed Lifecyle nodes a swarm of turtlesim nodes is created and various processes during their lifecycle are contorled (managed) with the help of a web based Lifecycle Management Interface.

5.0.1 Convertig Turtlesim into a Lifecycle node

5.0.2 Test Steps

- Reconfiguring the input sensitivity(in state Configure)
- Respawning on the Fly (transitions create, configure, activate)
- Shuttingdown individual Turtlesim Node (shutdown)

6 Result

7 Discussion

References

[1] Koubaa Anis. *Robot Operating System (ROS) The Complete Reference (Volume 5)*. Springer Nature Switzerland AG 2021, 2021. ISBN: 978-3-030-45955-0. URL: `https://doi.org/10.1007/978-3-030-45956-7`.

[2] Jangla Kinnary. *Accelerating Development Velocity Using Docker*. Apress, 2018. ISBN: 978-1-4842-3935-3. URL: `https://doi.org/10.1007/978-1-4842-3936-0`.

[3] Red Hat Inc. URL: `https://www.redhat.com/en/topics/containers/containers-vs-vms` (visited on 05/29/2021).

[4] Tarnum Java SRL(Baeldung). URL: `https://www.baeldung.com/cs/virtualization-vs-containerization` (visited on 05/29/2021).

[5] Cambridge english dictionary. URL: `https://dictionary.cambridge.org/dictionary/english/docker` (visited on 05/29/2021).

[6] MAUREEN O'GARA. URL: `https://web.archive.org/web/20190913100835/http://maureenogara.sys-con.com/node/2747331` (visited on 05/29/2021).

[7] Docker Inc. URL: `https://docs.docker.com/get-started/overview/` (visited on 05/29/2021).

[8] SIEMENS AG. URL: `https://new.siemens.com/global/de/produkte/automatisierung/themenfelder/industrial-edge.html` (visited on 05/20/2021).

[9] Siemens AG. *Industrial Edge Management – Getting Started, V1.2.0, A5E50177815-AC*. 2021.

[10] Siemens AG. *Industrial Edge App Publisher – Bedienung, V1.2.0, A5E50177958-AC*. 2021.

[11] Open Source Robotics Foundation Inc. URL: `https://docs.ros.org/en/foxy/index.html` (visited on 05/25/2021).

[12] Maker Pro. URL: `https://maker.pro/ros/tutorial/robot-operating-system-2-ros-2-introduction-and-getting-started` (visited on 05/25/2021).

[13] Automaticaddison.com. URL: `https://automaticaddison.com/ros-2-architecture-overview/` (visited on 05/25/2021).

[14] Open Source Robotics Foundation Inc. URL: `https://index.ros.org/doc/ros2/Tutorials/` (visited on 11/27/2020).

[15] MDN Web Docs. URL: `https://developer.mozilla.org/en-US/docs/Glossary/State_machine` (visited on 05/25/2021).

[16] Inc. Open Source Robotics Foundation. URL: `https://design.ros2.org/articles/node_lifecycle.html` (visited on 05/25/2021).

[17] vuejs.org. *Vue.Js Documentation*. `https://vuejs.org/v2/guide/index.html`. 2019 (accessed January 30, 2020).

[18] vuejs.org. *Vue Guide*. `https://vuejs.org/v2/guide/`. 2020 (accessed June 24, 2020).

[19] vuejs.org. *Comparison with Other Frameworks*. `https://vuejs.org/v2/guide/comparison.html`. 2020 (accessed June 24, 2020).

[20] Web Dev Zone. *React vs. Angular vs. Vue.js: A Complete Comparison Guide*. https://dzone.com/articles/react-vs-angular-vs-vuejs-a-complete-comparison-gu. 2020 (accessed June 24, 2020).

[21] NodeJS.org. *NodeJS Homepage*. https://nodejs.org/en/. 2020 (accessed June 24, 2020).

[22] npmjs.com. *NPM Homepage*. https://www.npmjs.com/. 2020 (accessed June 24, 2020).

[23] vuejs.org. *Components Basics*. https://vuejs.org/v2/guide/components.html. 2020 (accessed June 24, 2020).

[24] vuejs.org. *The Vue Instance*. https://vuejs.org/v2/guide/instance.html. 2020 (accessed June 24, 2020).

[25] F5 NGINX. *Beginner's Guide*. URL: http://nginx.org/en/docs/beginners_guide.html (visited on 11/27/2020).

[26] F5 NGINX. *Configuring NGINX as a Web Server*. URL: https://docs.nginx.com/nginx/admin-guide/web-server/web-server/ (visited on 11/27/2020).

[27] ECMA-404 The JSON Data Interchange Standard. URL: https://www.json.org/json-en.html (visited on 05/25/2021).

List of Figures

List of Tables

Listings