Systemintegration for mobile Robotics

Masterarbeit von

AAYUSH YADAV

Matrikel-Nr.: 3549452

Contents

SIEMENS

1 Introduction

1.1 Motivation

1.2 Problem Statement

asdf fdsa

Nach [**Krypczyk2018**] müssen folgende Fragen nach der Analysephase des Softwareentwicklungsprozess beantwortbar sein:

- Was sind die entscheidenden Anforderungen an die zu erstellende Software?

- Welches Problem soll mithilfe des Anwendungssystems gelöst werden?

- Welche Wünsche haben Ihre Kunden und Nutzer an das System?

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. [1, 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. [1, 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. [2]
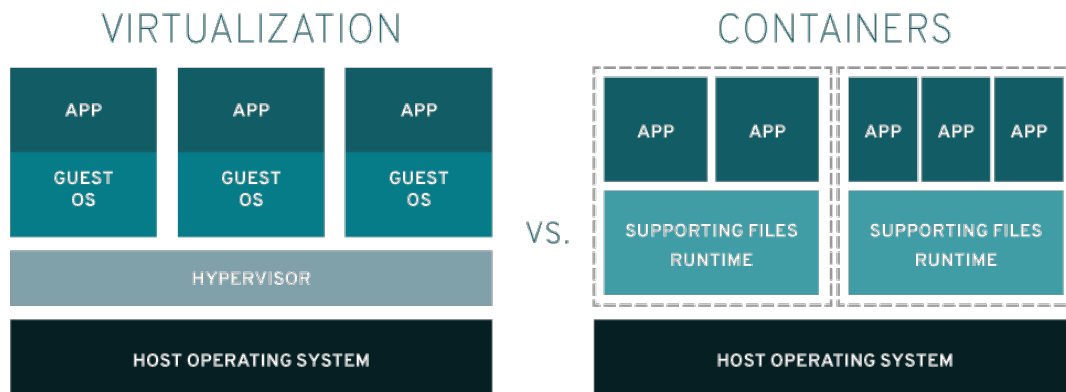
Figure 2.1: Differences between Virtualization and Containerization [2]

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.[2]

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.[2]

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 [3]

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. [4]

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.[5] 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.[1, 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.

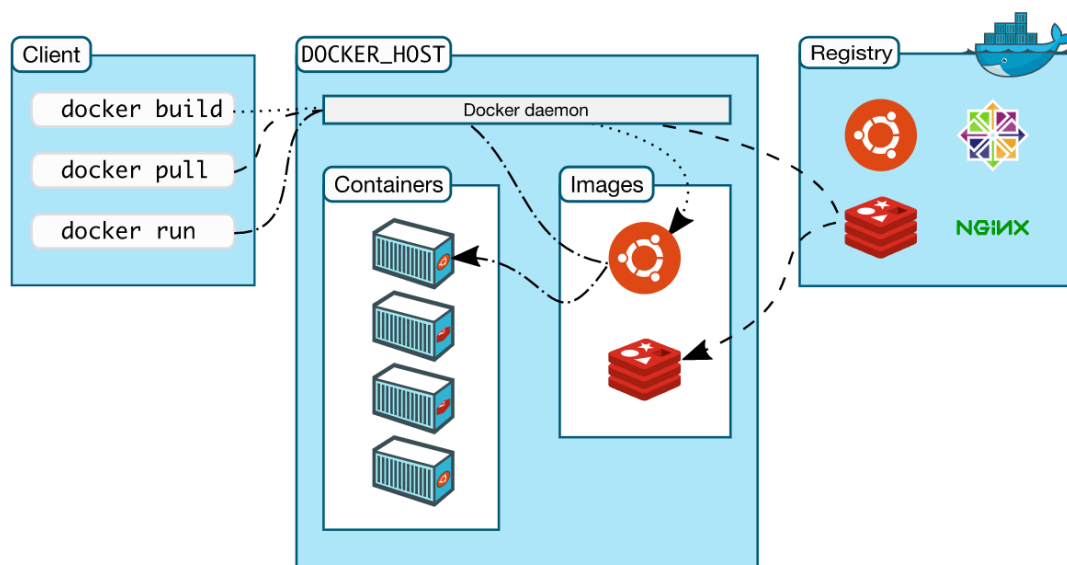Following Diagramm illustrates the Docker System Architecture.

Figure 2.2: Docker System Architecture [6]

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.[6]

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.[1, 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.[1, 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.[7]

   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.
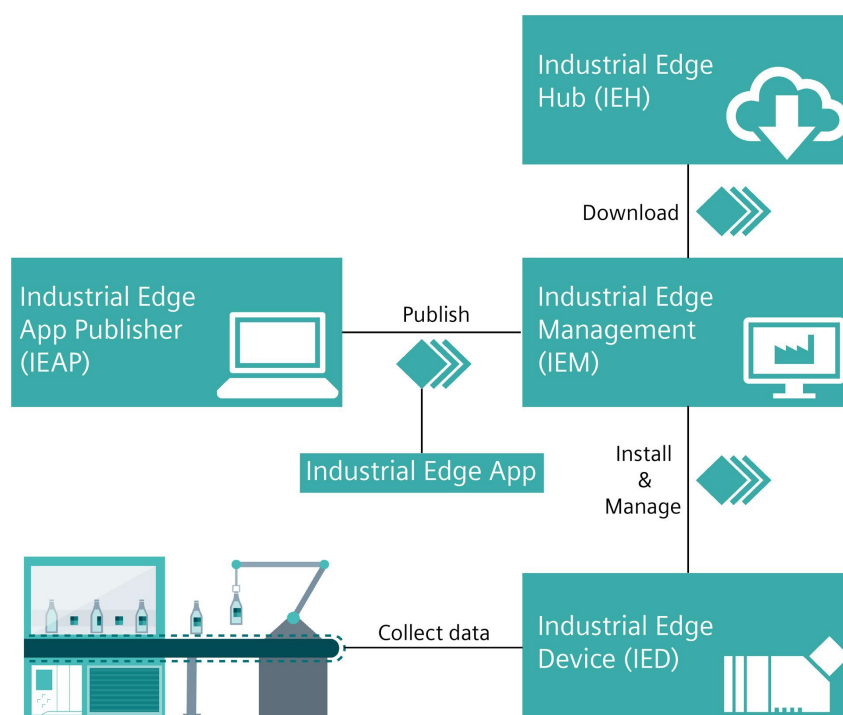

Figure 2.3: Industrial Edge overview [8]

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.[7]

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.[7]

Industrial Edge App   An Industrial Edge Application (IEA) (edge app) is used for the intelligent processing of industrial automation tasks.[7] 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.[9] 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.[10]

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.[10]

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 [11]

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.[12, P.7]
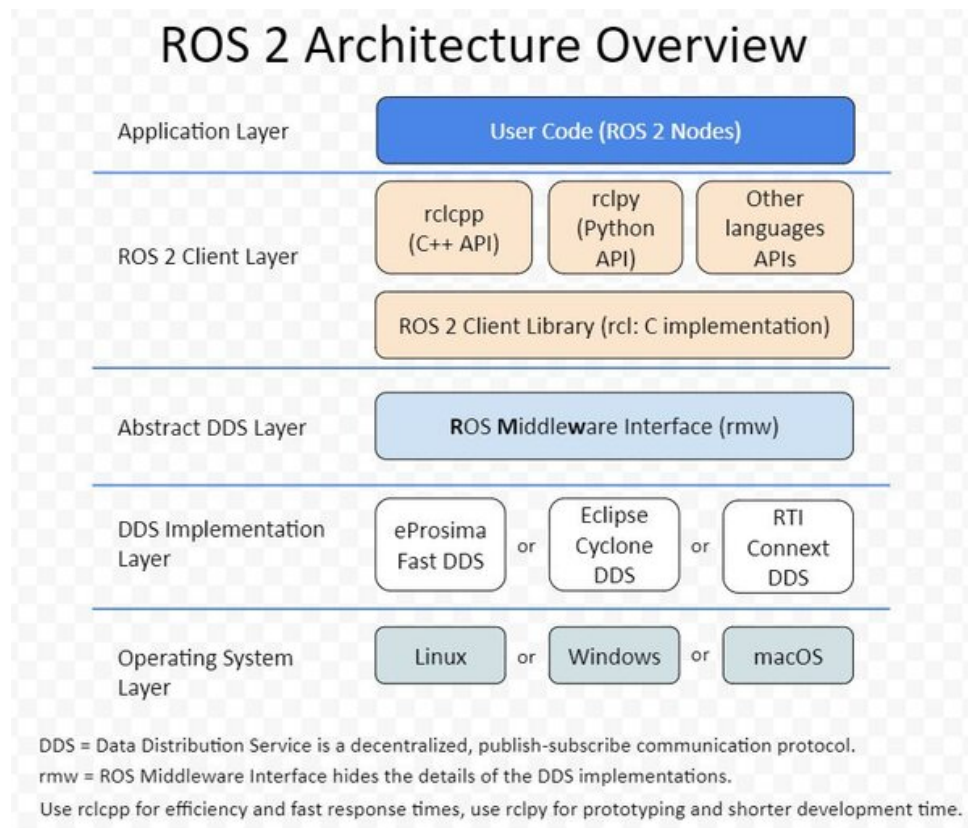

Figure 2.4: ROS2 Architecture [13]

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.
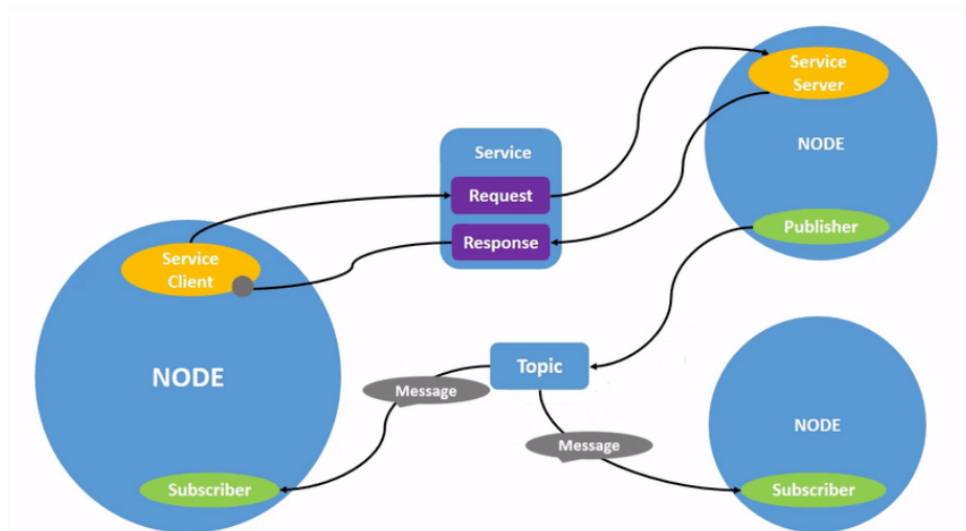

Figure 2.5: ROS2 Communication Pattern [10]

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.
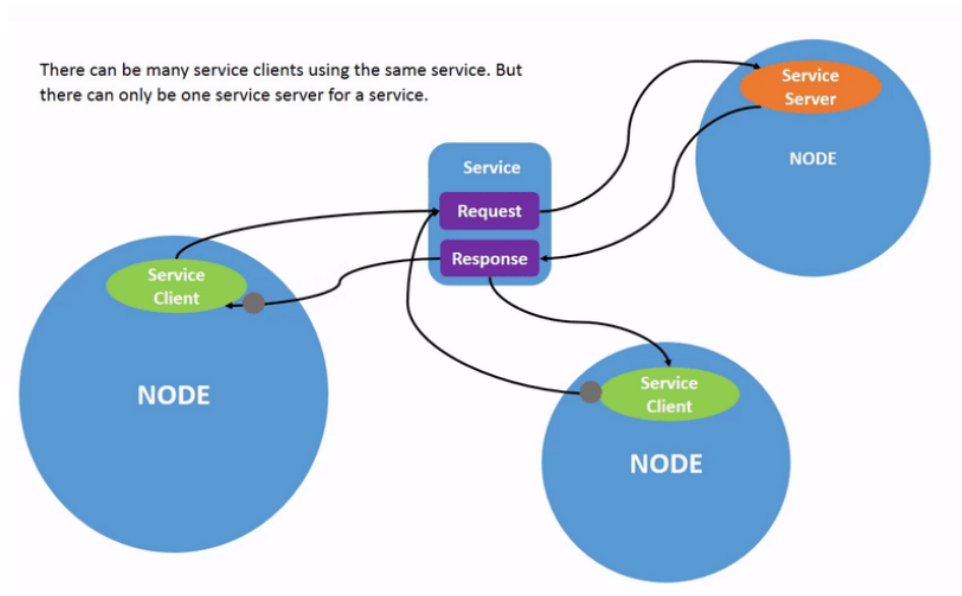

Figure 2.6: ROS2 Services [14]

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.
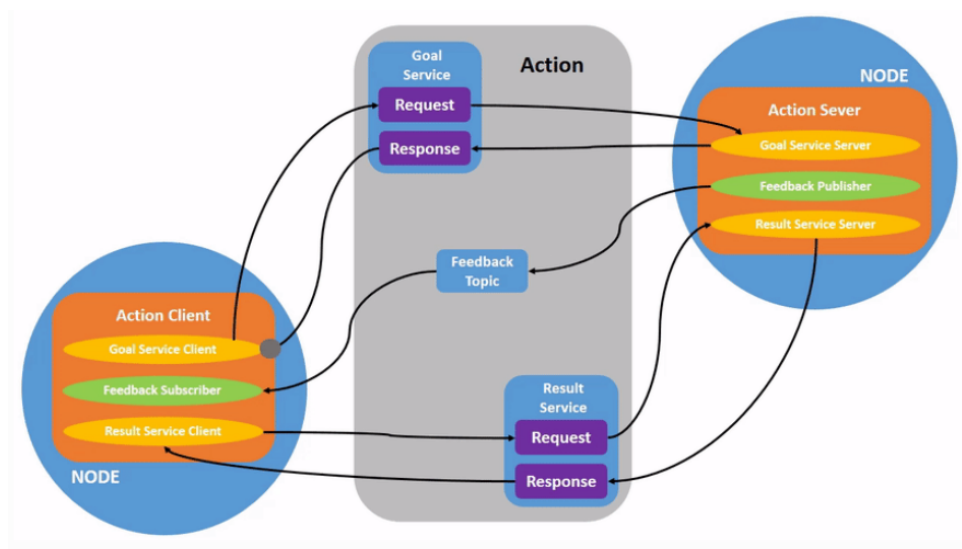

Figure 2.7: ROS2 Actions [14]

Figure 2.8 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.
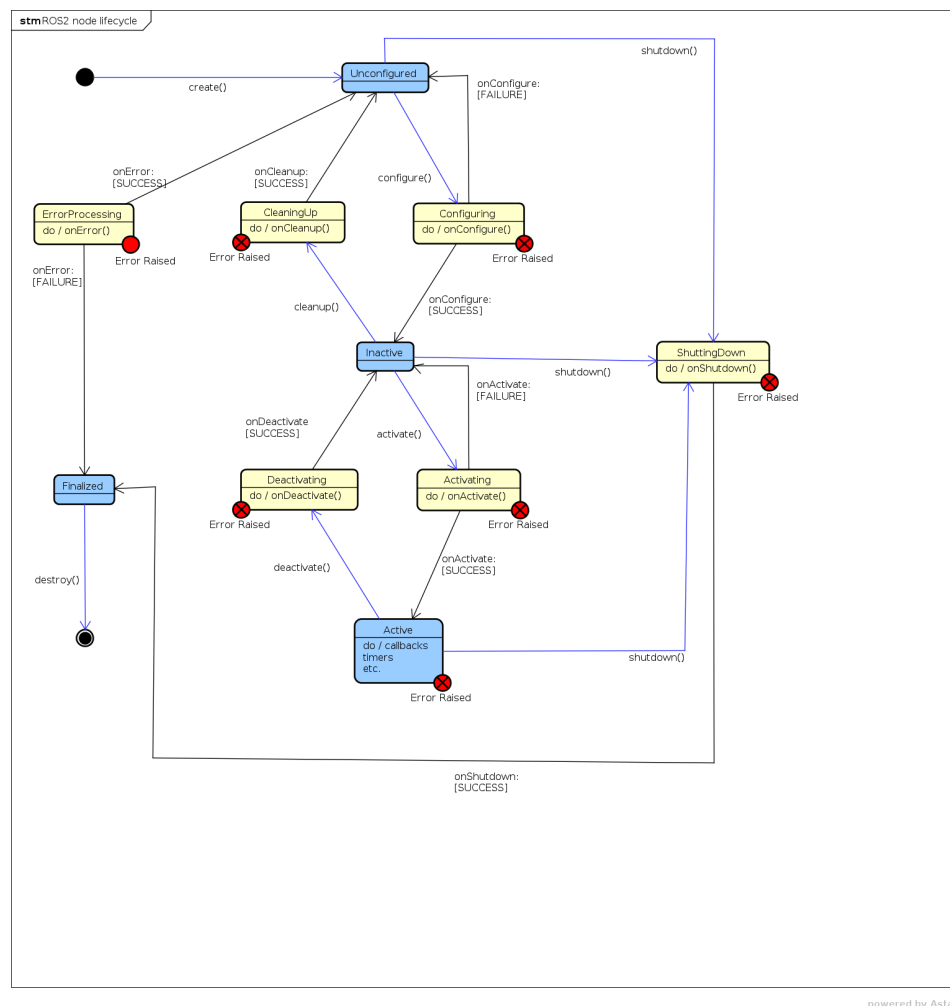

Figure 2.8: ROS2 Lifecycle Statemachine [14]

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 [15]. 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. [16]

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 [17].

Table 2.3: Vue vs React vs Angular

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

[18]

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) (node package manager)`.

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 [19].

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 [20].

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:

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

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

```
1  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:

```
1  npm run serve
```

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

```
1  npm run build
```

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

Table 2.4: 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 2.5). 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.

```
1  import Vue from 'vue'
2  import App from './App'
3  import router from './router'
```

```
4
5   // Root Instance
6   new Vue({
7       el: '#app',
8       router,
9       template: '<App/>',
10      components: { App }
11  })
```

Listing 2.1: 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 [15]. 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.
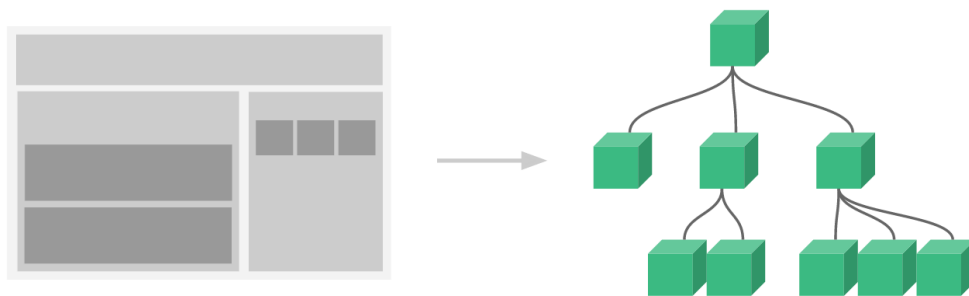
Figure 2.9: *Organization of Componenten* [21]

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):

```
1  <template>
2      <div>
3          <h1>Data: {{ beispielData }}</h1>
4          <BeispielComponent/>
5      </div>
6  </template>
```

Listing 2.2: 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.

```
1
2          <script>
3            import BeispielComponent from
                 'components/BeispielComponent'
4            export default {
5              components: {
6                BeispielComponent
7              },
8              props : [],
9              data () {
10               return {
11                 beispielData: 'Beispiel␣
                     String'
12               }
13             },
14             methods: {
15               beispielFunction: function
                   () { }
16             },
17             computed: {
18             }
19           }
20         </script>
```

Listing 2.3: Skriptelement

The most important parts of a component are described in the table 2.5.

Table 2.5: 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.
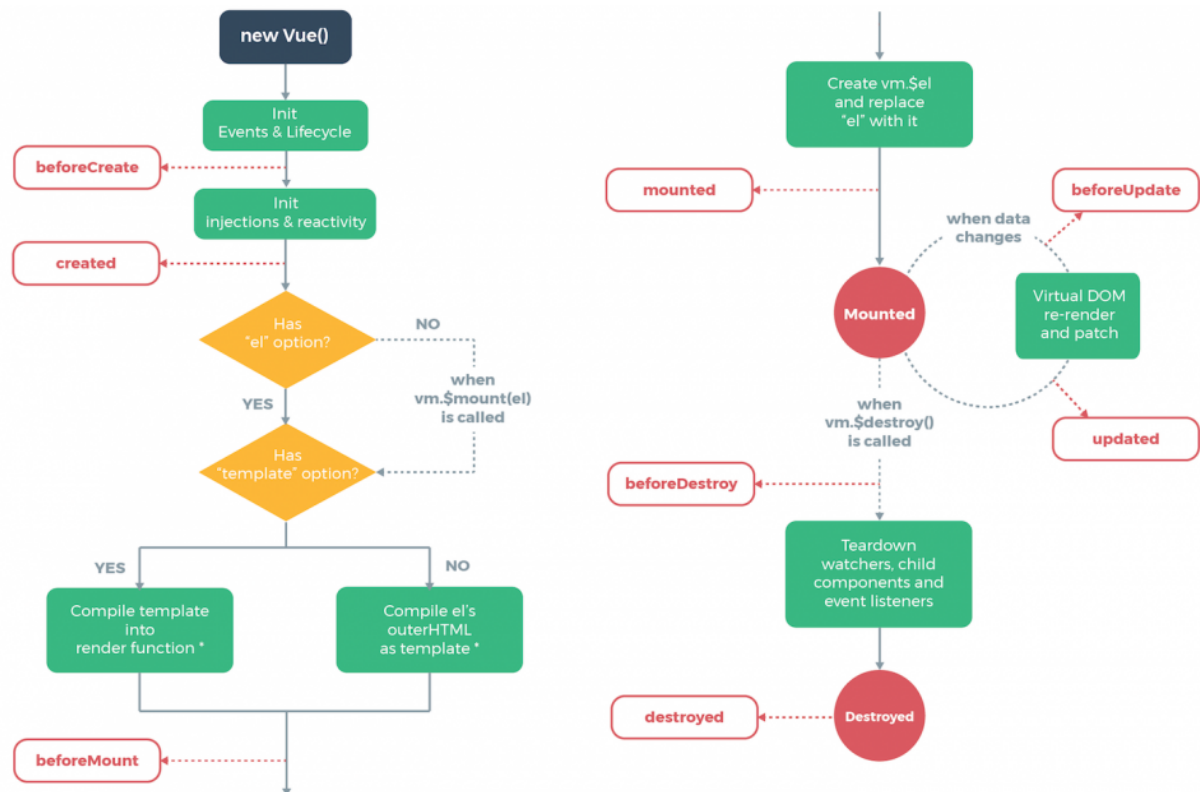
Figure 2.10: *Vue Instance-Lifecycle Hooks* [22]

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 [16].

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 [16].

2.7 Nginx

3 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 3.1 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).
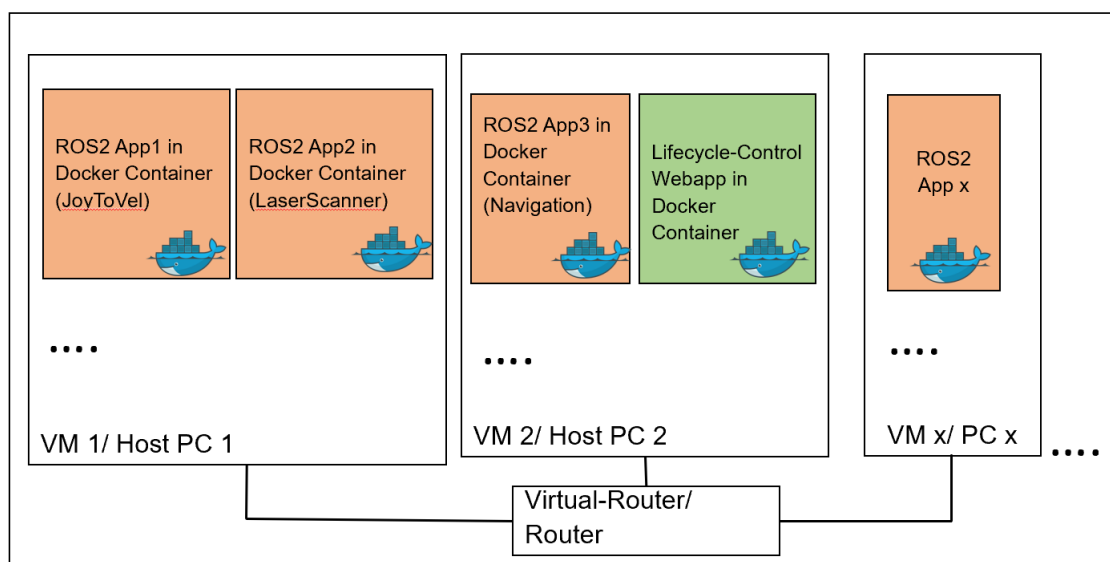
Figure 3.1: Structure of Lifecycle Management Application

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.

3.1 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.

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

```
1  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)

```
1  # install bootstrap tools
2  RUN apt-get update && apt-get install -y
      --no-install-recommends \
3          apt-utils \
4          build-essential \
5          nano \
6          git \
7          curl \
8          tmux \
9          inetutils-ping \
10         python3-colcon-common-extensions \
11         python3-colcon-mixin \
12         python3-rosdep \
13         python3-vcstool \
14         && 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.

```
1  # bootstrap rosdep
2  RUN rosdep init && \
3  rosdep update --rosdistro $ROS_DISTRO
4  # setup colcon mixin and metadata
5  RUN colcon mixin add default \
6          https://raw.githubusercontent.com/colcon/
7          colcon-mixin-repository/master/index.yaml && \
8          colcon mixin update && \
9          colcon metadata add default \
10         https://raw.githubusercontent.com/colcon/
11         colcon-metadata-repository/master/index.yaml && \
12         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.

```
1  # install ros2 packages
2  RUN apt-get update && apt-get install -y
      --no-install-recommends \
3          ros-foxy-ros-base=0.9.2-1* \
4          ros-foxy-joy \
5          ros-foxy-diagnostic-updater \
6          # install for lifecycle
7          ros-foxy-lifecycle \
8          nginx \
9          python \
10  # update system
11         && apt-get upgrade -y \
12         && 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.

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

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

```
1  # create and build workspace
2  ENV ROS2_WS /root/colcon_ws
3
4  COPY ./evo_siemensrob_ctrl
      ${ROS2_WS}/src/evo_siemensrob_ctrl/.
5  COPY ./joy_converter ${ROS2_WS}/src/joy_converter/.
6  COPY ./joytovel ${ROS2_WS}/src/joytovel/.
7  COPY ./twist_mux ${ROS2_WS}/src/twist_mux/.
8
9  RUN cd ${ROS2_WS} \
10         && . /opt/ros/foxy/setup.sh \
11         && colcon build
12
```

```
13  # install RIB support
14  COPY ./RIBInstall .
15
16  RUN ./RIBInstall \
17          && 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.

```
1   # For Lifecycle-Dashboard Vue App
2   ENV NODE_VERSION=12.6.0
3   # Already installed RUN apt install -y curl
4   RUN curl -o- https://raw.githubusercontent.com/
5   creationix/nvm/v0.34.0/install.sh | bash
6   ENV NVM_DIR=/root/.nvm
7   RUN . "$NVM_DIR/nvm.sh" && nvm install ${NODE_VERSION}
8   RUN . "$NVM_DIR/nvm.sh" && nvm use v${NODE_VERSION}
9   RUN . "$NVM_DIR/nvm.sh" && nvm alias default
      v${NODE_VERSION}
10  ENV PATH="/root/.nvm/versions/node/v${NODE_VERSION}/bin/
11  :${PATH}"
12  RUN node --version
13  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.

```
1   # build stage
2   WORKDIR /app
3   COPY ./lifecycle-dashboard/package*.json ./
4   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.

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

Setting up a Nginx web server   In this step the nginx configuration file 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.

```
1  # For nginx
2  COPY ./nginx.conf /etc/nginx/nginx.conf
3  RUN cp -a /app/dist/. /var/www/html
4  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.

```
1
2  # install ros2-web-bridge
3
4  # set workdirectory as /root
5  WORKDIR /root
6
7  #for automatic launch when container gets startet
8  CMD 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
9  # && ros2 launch evo_siemensrob_ctrl agv_control_launch.py
```

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

```
1  RUN ros2 launch evo_siemensrob_ctrl agv_control_launch.py
      &
```

3.2 ROS2 Lifecycle Application

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.

3.2.1 Conversion of a node into a Lifecycle Node

3.3 Vue based Webinterface

Figure 3.2: Lifecycle Dashboard

3.4 Integration

4 Result

- Schnittstelle für TIA Projekte definieren
- Funtionserweiterung für TIA Openess
- Device seite unabhängig von Master Client

5 Discussion

Mal schauen was so kommt :-)

Literaturverzeichnis

[1] 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`.

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

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

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

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

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

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

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

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

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

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

[12] 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`.

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

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

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

[18] 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).

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

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

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

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

List of Figures

List of Tables

Listings