Roboter Orchestrierungssoftware

Exposé für die Lehrveranstaltung Systemadministration

Stefan Geiring, 30658
Marvin Müller, 32850
Nicolas Baumgärtner, 32849

Wintersemester 22/23
05.November 2022

Betreuer: M.Sc. Aykan D. Inan
Ravensburg-Weingarten University of Applied Sciences

Inhaltsverzeichnis

1 Thema

Entwicklung einer Orchestrierungssoftware mit Weboberfläche auf Basis von ROS.
Roboter werden immer häufiger eingesetzt egal ob in Spezial Industrial bereichen oder auch einfach Zuhause. Die einfache und übersichtliche Orchestrierung vieler Roboter ist deshalb besonders wichtig. Roboterschwärme spielen außerdem, in der heutigen Zeit immer öfters eine wichtige Rolle. Ein Beispiel für große Roboterschwärme die bereits eingesetzt werden sind Lieferdrohen, Drohnenshows als ersatz für Feuerwerk oder die simplere Variante davon, eine Lagerverwaltung.

2 Motivation

Wir interresieren uns alle für die Welt der Robotik. Das Thema bietet unglaublich viel Lernpotential.
Desweitern ist das Thema fächerübergreifend, wir arbeiten in insgesamt 4 Sektoren: der reinen Softwareentwicklung, Webentwicklung sowie der Hardware/Elektronik Entwicklung und natürlich der Robotik Entwicklung/Forschung.

3 Ziel

Ziel des Projektes soll es sein eine Lauffähige Orchestrierungssoftware für Roboter auf ROS Basis zu entwicklen. Mit deren Hilfe man mehrere Roboter überwachen und auch steuern kann.
Desweitern eine selbst designde Hardware Platform für unsere kleinen Beispiel Roboter zu designen, welche im nachhinein durch diverse Komponenten erweitert werden können. Wie zum Beispiel Kamera, Bumber, Laser etc.

4 Eigene Leistung

Entwickung einer einfachen Weboberfläche auf ROS Basis welche als Orchestrierungssoftware für Roboter dienen soll. Die Weboberfläche der Orchestrierungssoftware soll in erster Linie als einfache Kontroll und Debugging Schnittstele dienen.
Um unsere Orchestrierungssoftware sinngemäß demonstrieren zu können sollen außerdem kleine Roboter auf ESP32 Basis gebaut werden. Diese Roboter sollen aus einem 3D-Gedruckten Gehäuse bestehen. Auf den ESP32 soll Micro-ROS auf Basis von FreeRTOS ausgeführt werden.
Unsere kleinen Beispiel Roboter sollen außerdem im Idealfall mit modular austauschbaren Erweiterungen ausstattbar sein. Diese Erweiterungen sollen Simple Sensorik und Aktorik zur Verfügung stellen.

5 Aufbau der Arbeit

- ROS in Docker Container lauffähig bekommen.
- Micro-ROS auf ESP32 oder anderer Hardware lauffähig bekommen. Hardware ist noch nicht final definiert, deshalb muss erst noch evaluiert werden ob der ESP32 unseren Anforderungen und Wünschen gerecht wird.
- Frontend für ROS auf Web-Basis programmieren.
- Evaluation des Frameworks für Web-Frontend (evtl. Software zwischenlayer nötig).
- Design des Roboter Gehäuses entwerfen, welches 3D-Gedruckt wird.
- Antriebsstrang des Roboters entwerfen.
- Evaluation und Testing welcher Antrieb unseren Anforderungen entspricht.

6 Grundbegriffe

Docker:

Software für die Container Verwaltung.

ROS:

Das Acronym ROS steht für Robot Operating System, es bietet eine Sammlung von Software-Bibliotheken, Werkzeugen und ein Framework um die Softwareentwicklung und Ausführung von Andwendugen für Roboter zu erleichtern.

Der Kern von ROS besteht aus Interfaces, genannt ROS-Graph, die eine anonymisierte und standardisierte Interprozesskommunikation ermöglichen. Dieser Graph ist ein Netzwerk aus 'nodes', welche über 'topics' miteinander kommunizieren. Auf einem 'topic' wird immer dieselbe 'message' mit einem definierten Datentyp von Nodes verbreitet. Für das Verbreiten und Empfangen von Messages müssen die Nodes 'publisher' und 'subscriber' implementieren.

Des weiteren gibt es noch viele Tools, die beispielsweise Daten visualisieren können, und eine große Menge an Bibliotheken, welche Standard-Algorithmen der Robotik implementieren. Diese Tools und Bibliotheken werden ebenfalls zu ROS gezählt weshalb ROS als mehr als ein Framework angesehen wird und den Titel 'Operating-System' im Namen trägt.

Es gibt eine ältere Version von ROS die einfach nur ROS genannt wird und eine neuere Version namens ROS2. Der Hauptunterschied zwischen ROS und ROS2 ist, dass ROS einen zentralen Server, genannt 'ROS-Master' verwendet über den die Kommunikation abläuft und ROS2 einen dezentralen Ansatz verfolgt.

ROS2 baut auf dem Data-Distribution Service Standard von OMG auf. Das ist eine Spezifikation für eine Middleware, welche ein 'Data-Centric Subscriber-Publisher (DCPS)'-Modell beschreibt.

React:

Web Frontend Framework was ursprünglich von Facebook ins leben gerufen wurde.

Arduino:

Entwicklungsplatform auf Basis von Atmel AtMega Prozessoren. Wurde entworfen um Leien den Einstieg in die Microcontroller Welt stark zu vereinfachen. Findet heutzutage Weltweit Anwendung in der Maker Szene.

ESP32:

ESP32 bezeichnet eine Microntroller Familie auf Basis der ARM Architektur. Ursprünglich wurde der ESP32 von Expresif entworfen und gefertigt.

RaspberryPi:

Toolchain:

Als Toolchain wird die Kombination von Micro-ROS und ESP-IDF bezeichnet. Mit Hilfe dieser beiden Frameworks wurde die Firmware für unseren Roboter programmiert und kompiliert.

RTOS

Abkürzung für Real Time Operating System. In diesem Projekt wurde in erster Linie das RTOS 'FreeRTOS' verwendet, das für die Verwendung mit Mikrocontrollern optimiert ist.

Micro-ROS

Micro-ROS ist eine Kombination von einem RTOS, einer Client-Bibliothek, einer Middleware und einem sogenannten ROS2 Agent, die Mikrocontroller möglichst effizient in das ROS-Oekosystem einbinden soll.

Mit Micro-ROS ist es nicht möglich ROS-messages direkt im Netzwerk zu versenden und mit anderen Teilnehmern direkt in Verbindung zu stehen. Die dafür benötigte Middleware würde zu viele Ressourcen auf einem Mikrocontroller verbrauchen. Stattdessen gibt es einen sogenannten ROS-Agent, mit dem sich der Mikrocontroller mit Hilfe einer für Mikrocontroller optimierten Middleware verbindet. Der ROS-Agent ist dann im Prinzip eine Brücke zwischen den verschiedenen Middlewares. Entsprechend muss der Agent auch auf leistungsfähigerer Hardware ausgeführt werden.

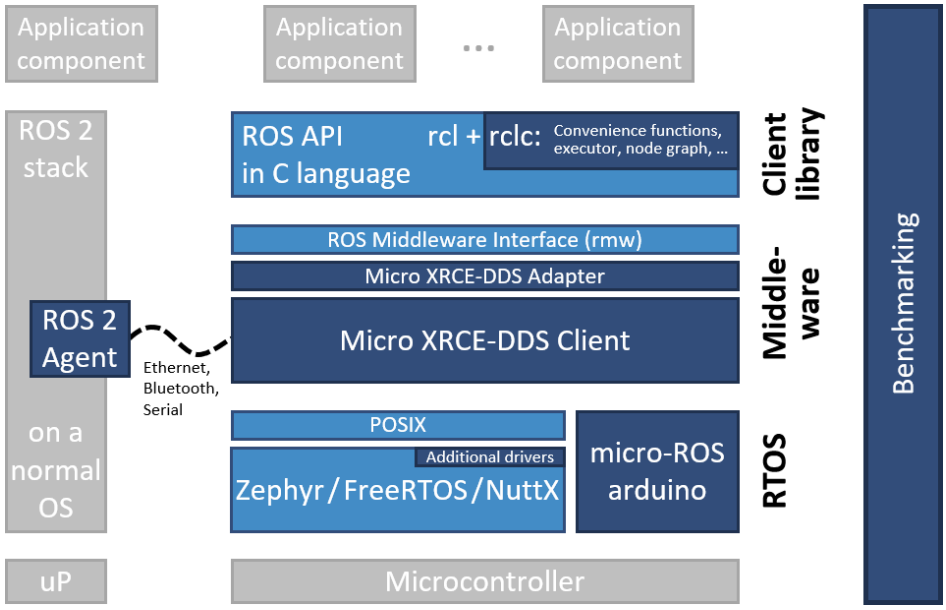


Abbildung 1: Micro-ROS Architektur

ESP-IDF

ESP-IDF steht für Espressif IOT Development Framework. Das ist ein Framework mit dessen Hilfe man ESP-Socs programmieren kann. Das betrifft in dem Fall dann auch den ESP32.

7 Zielsetzung und Anforderungen

Entwicklung einer Weboberfläche mit der Roboter auf ROS Basis gesteuert und überwacht werden können. Oberfläche zeigt alle Roboter an und stellt Basic tools zur verfügung um mit diesen zu kommunizieren und diese zu steuern. Außerdem soll damit eine Schnittstelle zwischen Browsern und ROS geschaffen werden.

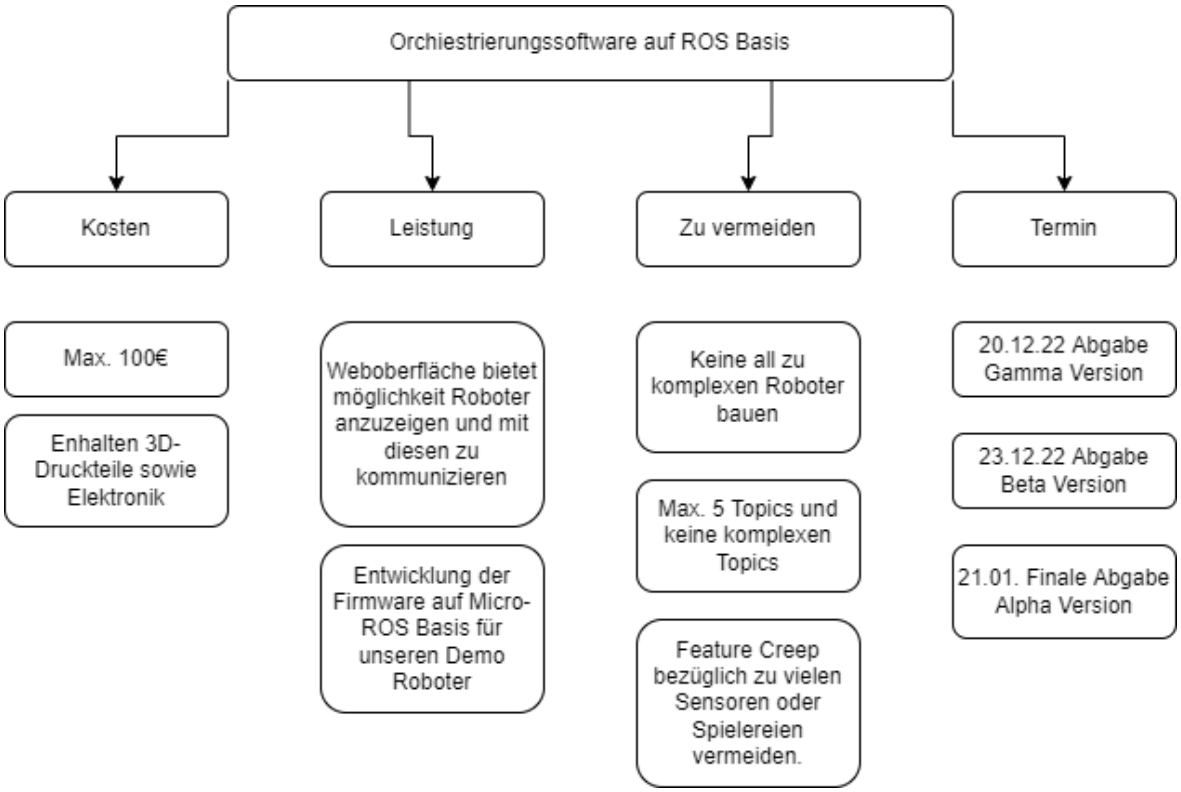


Abbildung 2: Anforderung Diagram

Kosten:

- Endkosten belaufen sich auf weniger als 100€.
- In diesen Kosten soll der Microcontroller inkl. 3D-Druck und gesamter Elektronik inkludiert sein.

Leistung:

- Weboberfläche ist lauffähig.
- Roboter kann mit ROS-Messages gesteuert werden.
- In Eigenleistung kleine fahrbare Roboterplatform auf ESP32 basis kreieren.

Zu vermeiden:

- Keinen zu komplexen Roboter designen. (Vollständig autonom fahrender Roboter)
- Feature creep mit all zu vielen Sensoren gilt zu vermeiden.

- Roboter Antrieb mit zu viel Technik ausstatten.
- Vorerst sollte nur ein Testroboter entwicklet werden.
- Webinterface nicht zu stark mit ROS Datentypen befüllen.

Termine:

- 20.11. Abgabe Gamma Version.
- 23.12. Abgabe Beta Version.
- 30.12. Feedback zu Beta Versionen.
- 21.01. Finale Abgabe.

8 Stand der Technik und Forschung

Technologischer Standpunkt Software:
WEBROS TODO ROS bietet bereits ein starkes und recht einfach nutzbares Framework, allerdings vermisst man eine schöne und einfach bedienbare "graphischeÖberfläche. Es existieren kleinere Projekte welche sich dieser Frontend entwicklung annehmen. Allerdings hat uns keines dieser Projekte zufrieden gestellt. Vor allem was die Orchestrierungsmöglichkeit mehrerer Roboter angeht.

Technologischer Standpunkt Hardware:
Es existieren bereits viele Forschungen und Beispiele für diverse Roboter und deren verschiedensten Antriebskinematiken. Für unser Projekt werden wir allerdings eine eigene Roboter Platform designen und 3D-Drucken. Viele der bereits vorhandenen Roboter sind entweder zu Groß und teuer oder viel zu klein und deshalb ebenfalls nicht gut für eine demonstration geeignet. Unter anderem wollen wir das unser Roboter den Vorteil bietet weitestgehend 3D-gedruckt zu sein. Die ETH Zürich (LINK?) hat bereits eine kleine Zusammenfassung über die wichtigsten Antriebsarten mobiler Roboter zur Verfügung gestellt. Für unser Projekt haben wir uns vorerst für einen simplen Diferentialantrieb entschieden.

9 Lösungsideen

9.1 Frontend

Das Frontend könnte durch normales plain HTML, CSS und JS realisiert werden. Da dies aber nicht mehr dem heutigen technischen Standart entspricht kämen viele verschiedene Frontend Frameworks in frage. Zur Auswahl stehen React, Vue oder Angular.

9.2 Roboter

Zur Demonstration unserer Orchestrierungssoftware soll ein kleiner Roboter gebaut werden. Der Roboter sollte, wie in unseren Anforderungen bereits aufgelistet, einfach und schnell gebaut werden können. Und damit unser kleiner Roboter auch problemlos in der Welt navigieren kann muss dieser auch mit einer Art Lenkung ausgestattet werden.
Für solch einen Steuermechanismus kommen viele verschiedene Lösungen infrage:
Ein Kettenantrieb ähnlich wie in Baggern wäre denkbar. Er benötigt nur 2 Motoren und bietet viel Bewegungsfreheit ohne komplizierte Mechanik.
Die Ackermann Lenkung wie Sie heute in allen PKW und LKW vorkommt wäre ebenfalls denkbar. Hierbei müsste man nur einen Motor einbauen für den allgemeinen vortreib und einen kleinen Servo um die tatsächliche Lenkbewegung der Vorder- oder Hinterache zu realisieren.
Der Differentialantrieb wäre eine weitere Methode unseren Antrieb zu realisieren, hierbei werden wie beim Kettenantrieb ebenfalls zwei Motoren benötigt. Allerdings entfällt hierbei die Notwendigkeit einer Laufkette.
Unsere letze Idee wäre eine Knicklenkung wie sie oft in Baustellenfahrzeugen eingesetzt wird. Es bräuchte nur einen Motor als Antrieb und die etwas kompliziertere Mechanik von der Ackermann Lenkung könnte durch eine simplere ersetzt werden.

Unser Roboter kommt natürlich nicht nur mit einem einfachen Antrieb aus sondern muss diesen auch entsprechend ansteuern können. Ein Arduino könnte sehr gut dafür geeignet sein.
Ein ESP32 käme auch in frage. Er bietet weitaus mehr Rechenpower als ein Arduino bei minimaler Preiserhöhung. Außerdem unterstützen die meisten ESP32 eine Verbindung über WLAN.
Als teuerste aber auch Leistungsstärke Kontrolleinheit könnte auch ein Raspebrry Pi dienen. Da wir unsere Roboter mit hilfe vom ROS Framework ansteuern wollen, sollte unsere gewählte Steuereinheit Micro-ROS unterstützen. Micro ROS ist eine fork von ROS selbst und dadurch auchauf schwächeren Systemen lauffähig.

10 Evaluation der Lösungsideen anhand der Anforderungen

10.1 Evaluierung der Frontend-Frameworks/Bibliotheken

Zur Auswahl standen uns mehrere Frontend-Frameworks, zur Umsetzung unserer Roboter-Orchestrierungssoftware, zur Verfügung. Das erste und wichtigste Kriterium war es hier eine Schnittstelle zu ROS2 zu bekommen, um verschiedenste Topics senden (publish) und empfangen (subscriben) zu können.
Außerdem wollten wir die Webanwendung in einer der bekannteren Webframeworks wie React oder Vue.js umsetzten. Hier stand kein Favorit der beiden im Vorfeld fest, sowie auch wenig bis keine Vorerfahrung. Wie entschieden uns hier für das React Framework, da wir es für sinnvoll hielten, unser Wissens-Repertoire diesbezüglich auch zu erweitern. Ebenso gibt

es im Netz mehr Hilfestellungen zu React als zu Vue.js, was bei so einer speziellen Aufgabe wie einer Schnittstelle mit ROS2, schlussendlich zum Vorteil tragen kann.
Nach etwas Recherche fanden wir eine React-ROS Schnittstelle die im npm-Paketmanager zur Verfügung gestellt wurde:

`https://www.npmjs.com/package/react-ros`

Problem bei diesem Paket war es, das es nicht mit der aktuellen React Version (18) kompatibel war und die Entwicklung hierzu wohl auch schon eingestellt wurde. Man findet auch kaum Erklärungen hierzu, weshalb wir es mit diesem Paket nicht zum laufen bekommen haben und uns eine andere Möglichkeit überlegt haben.
Doch wenn man etwas weiter recherchiert stößt man sehr schnell auf die *roslibjs* Bibliothek, auf die auch das vorher erwähnte Paket aufbaute.

`https://github.com/RobotWebTools/roslibjs`

Diese basiert auf Javascript und verwendet Websockets um sich mit der *rosbridge* zu verbinden und somit Funktionen wie publishen, subscriben, service calls und vieles mehr zur Verfügung stellt. Die *rosbridge* ist ein ROS Paket, welches eine JSON-API zu ROS-Funktionalitäten für nicht ROS-Programme zur Verfügung stellt.
Auch diese Bibliothek ist mittlerweile schon etwas älter, wird aber trotzdem noch in sehr vielen Projekten verwendet, weshalb wir uns auch für diese Option entschieden. Auch haben wir zu diesem Zeitpunkt keine andere akzeptable Alternative gefunden (hierzu später im Punk zu rosboard mehr), weshalb wir uns für einen Prototypen mit der *roslibjs* Bibliothek entschieden.
Um unserer Anwendung ebenso einen modernen Look zu verpassen, beschlossen wir eine UI-Komponentenbibliothek zu verwenden. Auch hier standen uns mehrere Optionen zur Auswahl. Zu den Beliebtesten und Bekanntesten zählen hier Material-UI (MUI), Ant Design (AntD) und React Bootstrap. Die Entscheidung fiel uns hier sehr leicht, da schon Erfahrungen im Standard-HTML Bootstrap Framework bestanden, weshalb wir hier auf eine schnellere Einarbeitung in das React Bootstrap Framework hofften.
Nun zusammengefasst verwenden wir:

- React (als Frontend-Framework)
- roslibjs (als Javascript Bibliothek im Frontend)
- rosbridge (ein ROS-Paket mit JSON-API)
- React-Bootstrap (UI-Komponentenbibliothek)

10.2 Warum ROS?

Da wir alle bereits mit ROS gearbeitet haben, ist die Softwareentwicklung für einen Microcontroller deutlich einfacher, da ROS viele Standard-Aufgaben wie die Interprozesskomunikation erledigt. Außerdem ist es für uns leichter bekannte Hürden leichter zu umgehen. ROS stellt auch eine große Auswahl an Bibliotheken und Tools bereit, die die Entwicklung ebenfalls erleichtern und beschleunigen.

Ein weiterer Grund dafür, dass wir uns für die Entwicklung mit ROS entschieden haben ist, dass wir im Robotiklabor viel Hilfe bekommen können, da dort alle Roboter mit ROS entwickelt werden. Weitere Vorteile von ROS sind, dass es open-source ist und kostenlos.

10.3 Micro-ROS und RTOS

Die Entwicklung mit dem Mikrocontroller findet mit Micro-ROS statt, da das die Standard-Bibliothek für die Entwicklung mit Microcontrollern mit ROS2 ist.
Das Besondere an ROS2 und Micro-ROS im Vergleich zu ROS1 ist, dass es ermöglicht ein Real Time Operating System auf dem Microcontroller auszuführen. Ein RTOS wird standardmäßig mitinstalliert, ist aber für uns auch interessant, da wir bisher noch nicht mit RTOS gearbeitet haben und es eine gute Lernmöglichkeit mit beschränkten Aufwand ist.
Wie ROS selbst, ist Micro-ROS open-source und kostenlos.
Außerdem gibt es eine bereits bestehende Toolchain namens ESP-IDF, die das Cross-Compilieren, Flashen, Monitoring und Debugging erleichtert.

10.4 Roboter Platform

Die Anforderungen an die Roboter Demo Platform für dieses Projekt waren recht klein und einfach gehalten.
Das Hauptaugenmerk bei der Neuentwicklung und dem Design unserer Roboter Platform sollte zum einen die 3D-Druckbarkeit des Roboters sein aber auch das der Roboter vor allem in kurzer Zeit gedruckt und betriebsfertig gemacht werden kann.
In unserem Fall zeigten sich für die Antriebsart beim Differentialantreib die meisten Vorteile. Wir benötigen nur 2 Motoren die unabhängig voneinander angesteuert werden müssen. Es können normale Reifen verwendet werden und keine aufwendig zu bauende Ketten. Und beim Differentialantreib wird lediglich eine um 360° frei rotierbare Rolle benötigt. Somit entfällt auch ein großer Aufwand bei der Lenkmechanik.
Im allgemeinen sollte die 3D-Druckbarkeit der Teile somit kein Problem darstellen.
Für die Elektronik bzw. die Steuerung unseres Roboter fiel die Entscheidung auf einen ESP32. Dieser ist kostengünstig und recht einfach zu beschaffen. Zusätzlich bietet der ESP32 weitaus mehr Rechenleistung als ein Arduino und garantiert somit auch eine gewisse Skalierbarkeit. Die restlichen elektronischen Bauteile wie z.B. Lineare Spannungswandler können in diversen Elektronik Versandshops gekauft werden.

11 Implementierung

11.1 React Front-End

11.1.1 Erstellung des Prototypes

Der wichtigste Punkt war es nun eine lauffähige Schnittstelle zwischen ROS2 und einer Webanwendung herstellen zu können. Für schnelles prototyping bietet die Open Source Robotic Foundation (OSRF) vorkonfigurierte Docker Container, in jeder beliebigen ROS Version, an. Hierzu muss nur Docker Compose auf dem Rechner installiert sein, und ein

fertiges Image kann sogleich erstellt werden. Um beispielsweise ein ROS2 Image mit der Foxy Fitzroy Version zu installieren, muss folgender Befehl ausgeführt werden.

```
docker pull osrf/ros:foxy-desktop
```

foxy-desktop kann hier mit jeder anderen beliebigen Version ausgetauscht werden. Mehr Informationen hierzu findet man im entsprechenden Docker Hub:

```
https://hub.docker.com/r/osrf/ros/
```

Da ROS Nodes über TCP/UDP Sockets kommunizieren, muss nun eine Brücke geschaffen werden, um Daten mit dem Web-Browser austauschen zu können. Hier schafft das *Rosbridge* Paket Abhilfe, in dem es einen Websocket erstellt, welcher in allen gängigen Web-Browsern unterstützt wird. Mit diesem Paket ist es nun Möglich die bekannten Publish- und Subscribe-Funktionalitäten der ROS-Umgebung zu nutzen. Die *Rosbridge* kann ganz einfach über den apt-Paketmanager installiert werden:

```
sudo apt install ros-<ROS_DISTRO>-rosbridge-server
```

Für eine genaue Anleitung um die *Rosbridge* lauffähig zu bekommen, wird auf den Anhang verwiesen. (fehlt noch..)
Im nächsten Schritt wird eine einfache Webanwendung gebaut, die mit Hilfe der *roslibjs* Bibliothek nun mit der *Rosbridge* kommuniziert und Daten austauscht. Hierzu wird eine simple HTML-Seite erstellt und *roslibjs* inkludiert:

```
//Inkludierung und Quelle in HTML
<script
    type="text/javascript"
    src="http://static.robotwebtools.org/roslibjs/current/roslib.min.js">
</script>
```

Wenn nun die *Rosbridge* erfolgreich in der ROS-Umgebung, in diesem Fall innerhalb des Docker Containers unseres Prototypens, gelauncht wurde, kann man sich mit einem Javascript-Objekt eine Verbindung zu dem Websocket aufbauen. Hierzu wird die IP-Adresse des Docker-Containers mit dem Port 9090 benötigt. Mit der URL *ws://171.17.0.3:9090* haben wir uns beispielsweise in unserer Anwendung verbunden. Ist die Verbindung erfolgreich hergestellt, können Publisher und Subscriber an das Objekt angebunden werden. Ausführliche Erklärungen und Code Beispiele sind im Anhang nachzulesen. (fehlt noch..)
Nachdem eine simple Schnittstelle zwischen ROS2 und einem Web-Browser erfolgreich hergestellt wurde, wird es im Folgenden um die Umsetzung der React-App gehen.

11.1.2 Aufbau des Frontends (Architektur)

Da zuvor noch keine Vorerfahrungen zum Erstellen von React-Apps bestand, erfolgte hier erst eine Einarbeitung in diverse Grundlagen und Funktionalitäten in React. React ist ein Javascript Framework zum Entwickeln von Webseiten und Webanwendungen. Statt einfachen statischen HTML-Seiten wird hier mit sogenannten Komponenten gearbeitet, die mehrfach verwendet werden können. Weitere Pakete und Bibliotheken können mit dem Paketmanager npm jederzeit nachinstalliert werden.

Ebenso gibt es auch die *roslibjs* Bibliothek im npm-Paketstore. Mit dem Befehl:

```
npm install roslib
```

wird diese dem Projekt hinzugefügt und kann in folgender Weise inkludiert werden:

```
import ROSLIB from 'roslib';
```

Bezüglich der Architektur, bzw. dem Aufbau der Website, haben wir uns an die standardmäßige Vorgehensweise in React gehalten, in dem man die Seite in Komponenten aufteilt und diese so an mehreren Stellen wieder verwenden kann. Dafür haben wir einen extra Verzeichnis */components* im */src* Verzeichnis angelegt, sowie eines mit */pages* für die jeweiligen Seiten. Diese werden in der *App.js* Datei mit dem *react-router-dom* Paket geroutet.
Wir haben uns für eine linksbündige Navigationsleiste entschieden, da diese mehr zu einem Dashboard und einer Konfigurationsseite passt. Auch hier wurde im ersten Moment nicht viel Wert auf ein besonders ausgefallenes Design gelegt. Eine schwarze Navigationsleiste mit einem aufklappbarem Burgermenü war hier für uns ausreichend.
Für den generellen Aufbau und dem Design der Website hatten wir uns im Vorfeld schon Gedanken gemacht, so war es uns auf jeden Fall wichtig eine Übersichtseite mit allen Topics zu haben und von diesen die Inhalte auslesen zu können. Im laufe der Wochen hatten wir ein Gespräch mit Benjamin Stähle aus dem RoboLab an der RWU, wir erzählten ihm von unserem Vorhaben und er zeigte uns ein ROS-Webdashboard namens *rosboard*. Diese Plattform hatte quasi die Funktionen, die wir für unsere Webanwendung auch geplant hatten. Zu diesem Zeitpunkt überlegten wir uns, ob wir von nun an diese verwenden, oder unsere eigene Webanwendung programmierten. Natürlich hätte es hier schon alle unsere gewünschten Funktionen zur Verfügung gehabt, allerdings entschieden wir uns dafür, einmal diesen Prozess von Grund auf selber zu entwickeln und unsere eigene ROS-Webanwendung zu erstellen. Wir wollten uns aber vom Design und der Vorgehensweise trotzdem an der vorgestellten Anwendung von *rosboard* orientieren. Ebenso unterscheidet diese sich auch von der Implementierung, da diese auf ganze andere Bibliotheken basiert.

11.1.3 ROS-Web Kommunikation

Um in der Webanwendung mit ROS kommunizieren zu können sind unsere wichtigsten Funktionen das Subscriben und Publishen von Topics. Im folgenden Codebeispiel wird gezeigt, wie mittels Javascript und der *roslibjs* Bibliothek ein Subscriber erstellt wird und mit einem Listener in einer Callback-Funktion Änderungen aus einem Topic ausgibt.

```
const my_topic_listener = new ROSLIB.Topic({
    ros,
    name: topicName,
    messageType: msgType,
});

my_topic_listener.subscribe((message) => {
    const newTopics = [...alltopicslist,
```

```
                {name: topicName, content: msgType}
            ];
        setTopics(newTopics);
    });
```

In *my_topic_listener* müssen zur Initialisierung sowohl das ROS-Objekt, der Topic-Name, sowie der Nachrichtentyp des angeforderten Topics enthalten sein.
Als Gegenbeispiel soll hier noch das Publishen von Topics gezeigt werden. Hier haben wir uns erst auf das */cmd_vel* Topic beschränkt, welches für die Bewegungsteuerung des Roboters zuständig ist. Hier können lineare Geschwindigkeiten, sowie Einschlagwinkel der Räder übermittelt werden.

```
const cmd_vel_listener = new ROSLIB.Topic({
    ros : ros,
    name : '/cmd_vel',
    messageType : 'geometry_msgs/Twist'
});

var twist = new ROSLIB.Message({
  linear: {
    x: linear,
    y: 0,
    z: 0
  },
  angular: {
    x: 0,
    y: 0,
    z: angular
  }
});
cmd_vel_listener.publish(twist);
```

11.2 ROS Back-End

11.2.1 Firmware Kompilierungs Toolchain

Um Programme auf dem ESP32 ausführen zu können müssen sie zuvor crosscompilt und auf den Speicher des Microcontrollers geflasht werden. Für das Crosscompilen von Programmen, die das Micro-ROS Framework verwenden wird eine Pipeline empfohlen, die aus 4 Teilen besteht.
1. ROS 2. Micro-ROS 3. RTOS 4. ESP-IDF
Zu Grunde liegt ersteinmal die ROS2 Compilierungspipeline welche colcon verwendet. Darauf aufbauend folgt die Micro-ROS Pipeline. In dieser Pipeline wird die Crosscompilierung für das entsprechende Realtime Operating System übernommen. Des weiteren wird noch die ESP-IDF Pipeline verwendet, die für die Crosscompilierung für den

ESP32 zuständig ist und mit deren Hilfe die Programme auf den Mikrocontroller geflash werden können.
Die Pipeline welche gerade beschrieben wurde, wurde mit Hilfe von Docker implementiert. Als erstes wird ein neuer User inklusive Home-Verzeichnis und entsprechenden Cgroup Berechtigungen angelegt. Für die Berechtigungen ist vor allem die dialout-Gruppe wichtig, da diese Zugriff auf seriellen Ports gibt.
Die Verwendung eines eigenen Users ohne root-Rechte wird verwendet, da es von Docker als Best-Practice empfohlen wird um ungewollte Veränderungen zu verhindern.
Nach dem Aufsetzen des Users wird die Micro-ROS Pipeline heruntergeladen und mit HIlfe von mitgelieferten Skripten installiert.
Nach der Installation müssen noch verschiedene Einstellungen vorgenommen werden. So ist es in eine ersten Schritt notwendig das passende RTOS, in diesem Fall "freertos", und den Mikrocontroller zu spezifizieren. Anschließend werden Einstellungen wie beispielsweise die Zugangsdaten für das WLAN-Netzwerk eingetragen.
Es ist außerdem noch notwendig den host-user zu den cgroups docker und dialout hinzuzufügen, da es ansonsten nicht möglich ist den Container zu starten oder Programme auf den Mikrocontroller zu übertragen.
Für das erleichterte Ausführen des Containers wird eine docker-compose-Datei verwendet. Der Container wird mit zwei bind-mounts eingerichtet. Einer für das Verzeichnis, in dem der Code des Roboters ist und ein anderer für das Verzeichnis /devïn dem die Ports für das flashen des Mikrocontrollers sind.
Es werden bind-mounts verwendet um die Daten und Ports, während der Entwicklung ständig aktuell zu halten um den Container nicht immer wieder neu starten zu müssen, wenn sich eine Datei ändert. Das vereinfacht die Entwicklung deutlich, da man wenn man den Mikrocontroller an und absteckt nicht den Container neu starten muss.

11.3 Roboter Platform

11.3.1 3D-Druckteile

Die Wichtigste Anforderung an die Platform war das diese Modular aufgebaut werden kann. Sprich man kann im Nachhinein neue Teile, Module oder komplett andere Sensoren einfach anschrauben aber auch kaputte bzw. alte Teile problemlos erneuern.
Da es sich beim Differentialantreib nur um eine Mechanische Anordnung der Motoren selbst handelt, musste noch ein geeignetes Bauteil als Motor selbst gefunden werden. Die Motoren müssen ein recht hohes Drehmoment aber eine kleine Umdrehungszahl pro Minute aufweisen. Initial war es deshalb die Idee für normale 5V Motoren wie sie auch in DVD Laufwerken verwendet werden ein kleines Getriebe zu konzeptionieren und zu bauen.

Abbildung 3: Prototyp des eigenbau Motorgetriebe

Die Idee bewies sich aber recht schnell als zu schwierig und wurde deshalb fallen gelassen. Als Ersatz für unseren Misslungenen Versuch fiel die Entscheidung auf Modellbau Getriebemotoren. Siehe Abbildung 4.

Abbildung 4: Modelbau Getriebemotor

Ein Anforderung an unsere Demo Roboter Platform war die 3D-Druckbarkeit, sowie deren Modularer Aufbau. Das Fahrgestell unseres Roboters setzt sich aus insgesamt 3 Modulen zusammen. Die Motorenhalterung woran die Motoren selbst und auch die H-Brücke befestigt sind. Die Zentrale Platform für unseren Microcontroller. Und die beiden Freirollen an der Front des Roboters.
Die gesamte Roboter Platform wurde mit Hilfe eines iterativen design Prozesses gestaltet. Zu Begin wurde nur das Fahrgestell des Roboters designed und gedruckt.
Sobald das Ergebniss zufriedenstellend war wurde das nächste Modul designed und gedruckt. Dadurch erreichten wir eine Art DEsignkette und konten Sicherstellen das alle

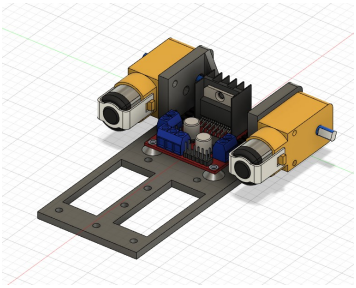
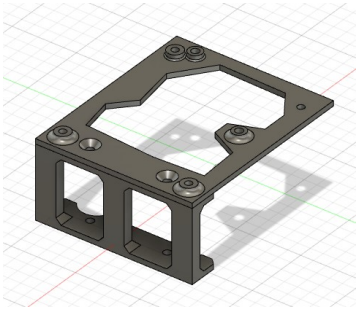
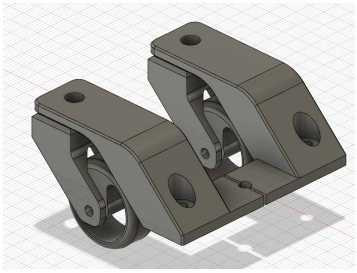


Abbildung 5: CAD Render vom finalen Fahrgestell

Teile bzw. Module zusammen passen und allen anforderungen gerecht werden. Die letzten beiden Module für unseren Demo Roboter waren die Platform für die Elektronik sowie die Freilaufrollen an der Front des Roboters.



(a) Elektronik Platform (finales Design)



(b) Freilaufrollen (2. Version)

Abbildung 6: CAD Render der Elektronik Platform und den Freilaufrollen

11.3.2 Elektronik

Wie in unserer Evaluation der Lösungsideen bereits beschrieben übernimmt der ESP32 Microcontroller, siehe Abbildung 7, die Steuerung unseres Roboters.

Abbildung 7: ESP32 Microcontroller

Dieser muss natürlich auch mit Spannung versogt werden. Urprünglich waren sogenannte 18650 LiIon Akkus geplant. Allerdings wurde diese Idee verworfen da Sie doch mit viel Aufwand verbunden war. Inzwischen nutzen wir wiederaufladbare 9V Block Batterien. Die Akkus haben die perfekte Größe für unseren kleinen Roboter.

Die Versorgungsspannung für die Motoren kann direkt von unserem Akkupack abgegriffen werden und zur H-Brücke geführt werden. Die H-Brücke benötigt genau so wie der ESP32 eine Versorgungsspannung für die Logik. Diese sollte laut Datenblatt bei 5V liegen, erlaubt sind aber auch 3V. Was in unserem Fall ideal ist da unser ESP32 ebenfalls nur mit 3,3V maxmimal versorgt werden darf.
Die 3,3V Logikspannung werden auf unserer Platine von einem LM317T, einem Lineareren Spannungswandler zur Verfügung gestellt.

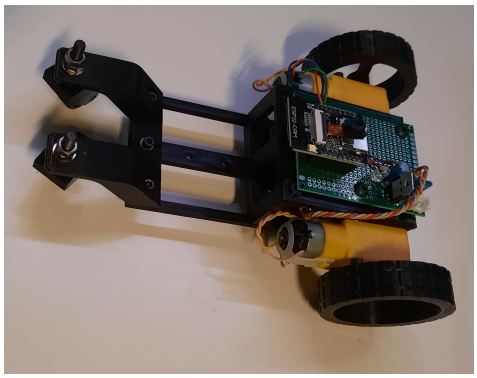


Abbildung 8: Roboter Demo Platform

Da durch die Verwendung der 9V Blockbatterie die Versorgungsspannung der Motoren nun die maximal zulässigen 6V überschreitet ist ein zweiter Spannungsregler von nöten der sich um die Reduzierung der Motorenspannung kümmert. Das ganze hat aber auch einen Vorteil da wir nun sehr große Freiheit haben was die Akkuspannung angeht. Theoretisch möglich wären nun bis zu 37V.

12 Evaluation der Implementierung

12.1 Fehler und Verbesserungen Frontend

12.2 Fehler und Verbesserungen Backend

Beim Aufsetzen der Pipeline ist es mehrfach zu Problemen gekommen, die es notwendig gemacht haben, die Implementierung anzupassen.
Zu Beginn wurde erst einmal mit Hilfe einer virtuellen Maschine ausprobiert, ob die Micro-ROS-Pipeline überhaupt funktioniert mit dem ESP32. Das hat gut funktioniert, es hat sich aber herausgestellt, dass die Pipeline sehr viel Speicherplatz benötigt und, dass sie relativ tief ins System eingreift. (TODO Beispiele)
Als nächstes sollte dann die Pipeline mittels Docker umgesetzt werden um zum einen die Verwendung einer virtuellen Maschine zu vermeiden aber trotzdem die Virtualisierung beizubehalten.
Für die Virtualisierung mit Micro-ROS mit Docker gab es 3 Möglichkeiten. 1. Die Verwendung der ESP-IDF Toolchain als Extension für VSCode. 2. die Verwendung von micro-Ros docker Extension 3. die Erstellung eines eigenen Docker-Images
Zuerst wurde die ESP-IDF Pipeline mit Hilfe einer Anleitung aufgesetzt. Während dem Aufsetzen ist es zu einem Problem mit docker-desktop gekommen und zwar sollte die

ESP-IDF-Extension mit Docker funktionieren, allerdings funktioniert sie nur mit dem standard-Context von Docker, der aber von Docker-Desktop verändert wird. Somit war es notwendig Docker-Desktop wieder zu entfernen und zu docker-engine zu wechseln.
Mit Docker-engine war es dann möglich die ESP-IDF-Extension mit VSCode auszuführen und Cdoe auf den ESP32 zu flashen. Die ESP-IDF Extension hat aber den Nachteil, dass es für uns notwendig wird, die ganze restliche Micro-ROS Pipeline selbst aufzusetzen.
Also wurde als nächstes die Micro-Ros docker Extension verwendet, die TODO. Hierbei wurde einmal versucht den debugger zu testen. Grundsätzlich ist der openOCD Server hochgefahren. Allerdings war es danach nicht merh möglich mit der ESP-IDF-Extension aus dem ersten Schritt auszuführen. Anscheinend hatte die Ausführung dieses OpenOCD-Servers Einstellungen in der Konfiguration von VS-Code docker gemacht, die zur Folge hatten, dass die ESP-IDF-Extension nicht mehr richtig ausgeführt werden kann.
Da sich herausgestellt hat, dass die ersten beiden Methoden nicht zuverlässig waren, wurde schließlich die letzte Methode gewählt.

12.3 Fehler und Verbesserungen Roboter Platform

- Initial sollten kleine 5V Motoren aus alten DVD-Laufwerken mit selbstgebauten getriebe als Antrieb dienen. Jedoch musste man recht schnell festellen dass das Design von einem Getriebe doch nicht so einfach ist. Es muss auf den perfekten Abstand der Zähne der Zahnräder geachtet werden. Das die Zahnräder konzentrisch und sich leichtgängig drehen und noch vieles mehr. Letzendlich war unser erster Getriebeversuch auch der letzte. Der Motor hatte einfach viel zu wenig Drehmoment um alle auftrettenden Reibungsverluste zu überkommen. Ein netter versuch um das ein oder andere zu lernen war er aber denoch.

- Der erste Versuch eine Freilaufrolle zu designen und zu drucken funktionierte nur zu 50%. Die Rolle konnte zwar ohne Probleme gedruckt werden, allerdings konnte sie sich nicht frei und leichtgängig genug drehen. Das lag zum einen daran das der Rollendurchmesser zu klein gewählt war und der Versatz von der Rolle selbst und dem Drehpunkt zu groß war. Das hatte ein zu hohes axiales Drehmoment am freilaufdrehpunkt zur Folge. Genau das sorgte für große Reibung und schlussendlich dafür das sich die Rolle nicht in Fahrtrichtung ausrichten konnte.

- Wie auch schon oben erwähnt war geplant den Akku für den Roboter selbst aus 18650 Zellen zu fertigten. Allerdings hätte macn dafür noch eine Lade- & Entladeelektronik benötigt und zusätzlich ein Punktschweißtgerät. Es war einfach zu viel Aufwand für zu wenig gewinn. Deshalb entschieden wir uns einfach für die 9V Blockbatterie. (Großen Dank an Dipl.-Ing. Joachim Feßler für die Bereitstellung der Akkus)

13 Fazit und Ausblick