Entwicklung und Simulation eines Telepräsenzroboters mit ROS 2 und Gazebo

Maylis Grune, DHBW Heidenheim

August 2025

Abstract

Die Arbeit beschreibt Konzeption, Modellierung und Simulation eines modularen Telepräsenzroboters auf Basis von ROS 2 (Humble) und Gazebo Classic. Es wurden drei URDF/Xacro-Modelle (Basis-, SLAM- und Mosro-orientiertes Modell) erstellt, virtuelle Sensorik (Kamera, Ultraschall, Lidar, IMU) integriert sowie eine entkoppelte Teleoperationsarchitektur mit Flask-Webdashboard und MJPEG-Streaming implementiert. Das resultierende Paketlayout trennt Bringup, Steuerlogik und Schnittstellen und bildet eine erweiterbare Grundlage für zukünftige Funktionen wie autonome Navigation (Nav2), Sicherheitslayer, zusätzliche Sensorik oder Übertragung auf reale Hardware.

Schlüsselwörter: ROS 2, Gazebo, Telepräsenzroboter, SLAM, Teleoperation
GitHub-Repository: `https://github.com/Maytastico/TeleNav`

1 Einleitung

1.1 Motivation und Problemstellung

In der heutigen Welt sind hybride Lern- und Lehrformate, insbesondere seit der COVID-19-Pandemie, nicht mehr wegzudenken. Virtuelle Angebote haben sich als wesentlicher Bestandteil professioneller Entwicklung und Lehre etabliert und bieten gerade in bildungsferneren oder peripheren Regionen neue Chancen McNamee and Horn (2023).

Die Integration von virtuellem und physischem Lernraum ermöglicht es, die Vorteile beider Welten zu kombinieren und ein flexibles, anpassungsfähiges Lernumfeld zu schaffen. Virtuelle und hybride Optionen gleichen dabei Defizite aus, die vor allem in kleineren und isolierten Institutionen bestehen, indem sie den Zugang zu Expertise, Netzwerken und professioneller Weiterbildung erleichtern.

Roboter können in diesem Kontext zu einer immersiven Lernumgebung beitragen, indem sie physische Präsenz in virtuellen Lernumgebungen herstellen. So wird es möglich, dass Lehrpersonen unabhängig vom Standort auf die Lernumgebung zugreifen und direkt mit den Lernenden interagieren.

Gerade bei Universitäten außerhalb von Ballungsgebieten zeigt sich der Nutzen solcher hybriden Ansätze besonders deutlich. Wie McNamee and Horn (2023) belegt, verfügen ländliche Hochschulen oft über weniger Ressourcen und haben Schwierigkeiten, ausreichend qualifiziertes Lehrpersonal zu gewinnen und zu halten. Virtuelle und hybride Lehr- und Lernformen können daher einen wichtigen Beitrag leisten, diese strukturellen Nachteile zu kompensieren und eine qualitativ hochwertige akademische Ausbildung auch in ländlichen Regionen zu sichern.

1.2 Zielsetzung

Die Zielsetzung dieser Arbeit besteht darin, eine modulare, simulationsbasierte Entwicklungs- und Testplattform für einen Telepräsenzroboter zu konzipieren und umzusetzen, die als belastbare Grundlage für eine spätere physische Realisierung dient. Im Vordergrund steht nicht die vollständige Autonomie des Systems, sondern der Aufbau einer klar strukturierten Architektur, die Teleoperation, sensorische Wahrnehmung und kartierungsbasierte Erweiterungen (SLAM) integriert.

Konkret sollen folgende Teilziele erreicht werden:

1. **Robotermodellierung:** Erstellung mehrerer URDF/Xacro-Modelle mit zunehmender funktionaler Tiefe (Basis-, SLAM- und mosro-orientiertes

Referenzmodell) einschließlich virtueller Sensorik (Kamera, Ultraschall, Lidar, IMU).

2. **Simulationsumgebung:** Aufbau einer Gazebo-Classic Welt (Lehr-/Laborumgebung) inklusive Erzeugung einer 2D-Karte mittels SLAM zur späteren Wiederverwendung.

3. **Softwarearchitektur:** Trennung in klar abgegrenzte ROS 2-Pakete (Bringup, Steuerlogik, Interfaces) zur Förderung von Wartbarkeit und Erweiterbarkeit.

4. **Teleoperation:** Entwicklung eines webbasierten Dashboards (Flask) zur fernbedienten Steuerung mit Video-Streaming (MJPEG) und Eingabeverarbeitung über Tastatur und Maus.

5. **SLAM-Integration:** Einsatz der slam_toolbox zur Kartierung.

Abgrenzungen:

- Keine Implementierung vollautonomer Navigation (Nav2 nur perspektivisch berücksichtigt).
- Keine hardwareseitige Realisierung; ausschließlich virtuelle Simulation.
- Sicherheits- und Kollisionsvermeidungslogik nur in Ansätzen (reaktive Distanzprüfung), kein globales Trajektorien-Management.

Erwartete Ergebnisse:

- Funktionsfähige Start- und Spawn-Pipeline (Launch, URDF, Gazebo, RViz).
- Reproduzierbare Karten generiert aus der Simulation.
- Dokumentierte Paket- und Klassenstruktur als Basis für Erweiterungen (Nav2, zusätzliche Sensorik, physische Portierung).

2 Grundlagen

2.1 Definition und Einsatzbereiche

Bei dem Telepräsenzroboter handelt es sich um eine mobile Roboterplattform, die in unterschiedlichen Lernräumen eingesetzt wird. Die Räume

sind klimatisiert und verfügen über eine WLAN-Infrastruktur, sodass eine stabile Kommunikation zwischen Roboter und Nutzer gewährleistet ist.

Die Gestaltung der Projekträume ist in der Regel auf kooperatives Arbeiten ausgerichtet: Tischinseln bieten ausreichend Platz, sodass sich der Roboter frei bewegen und mit den Lernenden interagieren kann. Die Figure 1 zeigt einen beispielhaften Projektraum, in dem der Telepräsenzroboter eingesetzt werden kann.

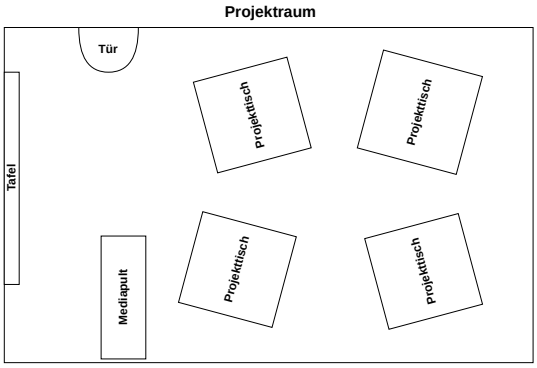


Figure 1: Beispielhafter Projektraum für den Telepräsenzroboter

In klassischen Unterrichtsräumen, die auf Frontalunterricht ausgelegt sind, sind die Tische dagegen in Reihen angeordnet. Dies führt zu einer deutlichen Einschränkung der Bewegungsfreiheit des Roboters und kann dessen Interaktionsmöglichkeiten im Raum begrenzen. Die Figure 2 zeigt beispielhaft den Aufbau eines solchen Lehrraums.

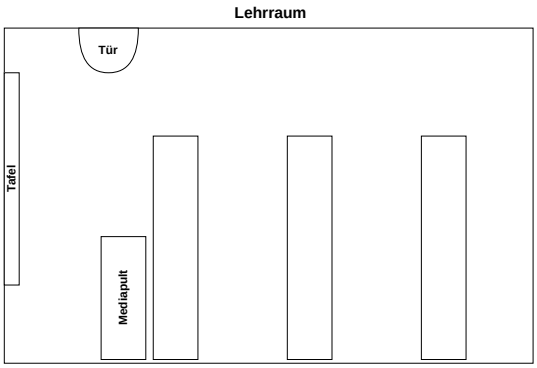


Figure 2: Beispielhafter Lehrraum für den Telepräsenzroboter

2.2 Robot Operating System (ROS 2)

Das Robot Operating System 2 (ROS 2) ist ein Software-Framework, das speziell für die Entwicklung moderner Robotersysteme konzipiert wurde. Es dient als Basis für Robotikanwendungen und zeichnet sich durch seine modulare und skalierbare Architektur aus. Die Struktur von ROS 2 besteht aus zahlreichen voneinander unabhängigen Paketen, die über mehrere Abstraktionsschichten verteilt sind. Ein zentrales Element ist die Middleware, die die Kommunikation zwischen den einzelnen Systemkomponenten ermöglicht. ROS 2 nutzt hierfür den Data Distribution Service (DDS), einen etablierten offenen Standard, der auch in sicherheitskritischen Anwendungen wie der Raumfahrt oder im Militär eingesetzt wird. DDS sorgt für eine zuverlässige und effiziente Datenverteilung innerhalb der Roboteranwendung (Macenski et al., 2022, S. 2).

Im Vergleich zu ROS 1, das auf TCP/IP setzt, bietet ROS 2 mit DDS eine verbesserte Kommunikation und Interoperabilität zwischen verschiedenen Robotersystemen, insbesondere über kabellose Netzwerke, wo der frühere Ansatz Einschränkungen hatte. Darüber hinaus werden alle Softwarebibliotheken über die Robot Client Library (rcl) . verwaltet, was die Integration und Nutzung von ROS 2-Paketen erleichtert. Während ROS 1 keine einheitliche Methode zur Verwaltung von Abhängigkeiten und Paketen bereitstellte, existiert diese in ROS 2 (Macenski et al., 2022, S.2).

Anwendungen in ROS 2 können über verschiedene Kommunikationsmechanismen interagieren: Topics ermöglichen eine Publish-Subscribe Kommunikation, Services bieten eine direkte Anfrage-Antwort-Kommunikation, und Actions erlauben eine zielorientierte, asynchrone Kommunikation mit Rückmeldung. Letztere ermöglicht es, Aufgaben als Ziele zu definieren, die der Roboter erreichen soll, während der Status der Ausführung überwacht wird. Dies erleichtert insbesondere die Implementierung von Aufgaben, die in der realen Welt ausgeführt werden, wie beispielsweise die Navigation von einem Punkt zu einem anderen(Macenski et al., 2022, S.4).

Die Middleware von ROS 2 (rmw) nutzt hierbei für die Kommunikation eine Interface Communication Abstraktion, die eine einheitliche Schnittstelle für verschiedene Transportprotokolle bereitstellt. Dadurch kann eine einheitliche Kommunikation zwischen verschiedenen Robotersystemen und -komponenten gewährleistet werden, unabhängig von den verwendeten Transportmethoden (Macenski et al., 2022, S.4).

Ein weiteres Features des ROS 2 Frameworks ist die Verwaltung des Lifecycles der Node. Hierbei kann die Node in verschiedene Zustände versetzt werden, wie z.B. "aktiv", "inaktiv" oder "fehlerhaft". Diese Zustandsver-

waltung ermöglicht eine bessere Kontrolle über die Ausführung und den Ressourcenverbrauch der Node (Macenski et al., 2022, S.4).

In dieser Arbeit wird dabei auf ROS2 Humble unter Ubuntu 22.04 gesetzt. Diese Version enthält Langzeitunterstützung (LTS) und ist daher für den Einsatz in stabilen und langfristigen Projekten geeignet.

2.3 Gazebo-Simulator

Robotersimulationen ermöglichen die schnelle und kostengünstige Erstellung von Prototypen, sowie deren Erprobung in unterschiedlichen Szenarien. Dabei können verschiedenste Sensoren und Aktoren realistisch nachgebildet werden, ohne dass physische Hardware erforderlich ist. Dies spart nicht nur Kosten, sondern reduziert auch den zeitlichen und organisatorischen Aufwand, der mit Aufbau und Wartung realer Robotersysteme verbunden wäre (Ayala et al., 2020, S. 3). In dieser Arbeit wird der Gazebo Classic Simulator eingesetzt. Zum Zeitpunkt der Erstellung war es sinnvoll, auf diese ältere, aber etablierte Version zurückzugreifen, da sie ein stabileres und umfangreicheres Funktionsspektrum bot, während die neuere Version Gazebo Harmonic noch mit verschiedenen Fehlern und Einschränkungen konfrontiert war.

Gazebo Classic bietet dabei Plugins an um mit der simulierten Welt zu interagieren. Diese können über das URDF Modell des Roboters eingebunden werden. Die Daten werden dabei als ROS2 Topic veröffentlicht und können von anderen Nodes abonniert werden. Das ermöglicht es, die simulierten Sensoren und Aktoren des Roboters in die ROS2-Architektur zu integrieren und die Steuerung und Navigation des Roboters zu realisieren.

3 Systemkonzept

Der Telepräsenzroboter soll die Interaktion zwischen Lehrenden und Lernenden über Distanz hinweg erleichtern. Im Rahmen dieser Arbeit wird ausschließlich mit einer virtuellen Simulationsumgebung gearbeitet. Ziel ist es, eine Grundlage für zukünftige Entwicklungen eines Telepräsenzroboters zu schaffen, der in realen Lernumgebungen eingesetzt werden kann. Der aktuelle Ansatz ermöglicht es, Sensordaten auszulesen, den Roboter zu steuern und Karten der Umgebung zu erstellen. Eine funktionierende Simulationsumgebung erlaubt es zudem, unterschiedliche Szenarien zu testen und die Funktionalitäten des Roboters zu evaluieren. Automatisierte Abläufe, wie beispielsweise die Navigation zu einem definierten Zielpunkt, sind hingegen nicht Bestandteil dieser Arbeit. Die folgenden funktionalen und nicht-

funktionalen Anforderungen beschreiben die wesentlichen Merkmale und Eigenschaften des Systems:

Funktionale Anforderungen

- **Remote-Steuerung:** Der Roboter wird innerhalb der Simulationsumgebung aus der Ferne gesteuert.
- **Hinderniserkennung:** Virtuelle Objekte in der unmittelbaren Umgebung werden über simulierte Abstandssensoren erkannt, um Kollisionen zu vermeiden.
- **Umgebungswahrnehmung:** Mithilfe simulierten Lidar-Sensorik erstellt der Roboter Karten der virtuellen Umgebung, die als Grundlage für Navigationstests dienen.

Nicht-funktionale Anforderungen

- **Benutzerfreundlichkeit:** Die Steuerung des Roboters innerhalb der Simulation soll intuitiv erfolgen und ohne lange Einarbeitung bedienbar sein.
- **Ressourceneffizienz:** Die Simulation soll auf handelsüblichen Rechnern lauffähig sein und die Systemressourcen effizient nutzen.
- **Erweiterbarkeit:** Die Simulationsumgebung soll modular aufgebaut sein, um zukünftige Anpassungen zu erleichtern.

Das konzeptionelle Design beschreibt den grundlegenden Aufbau des Telepräsenzroboters innerhalb der Simulationsumgebung.

Systemübersicht

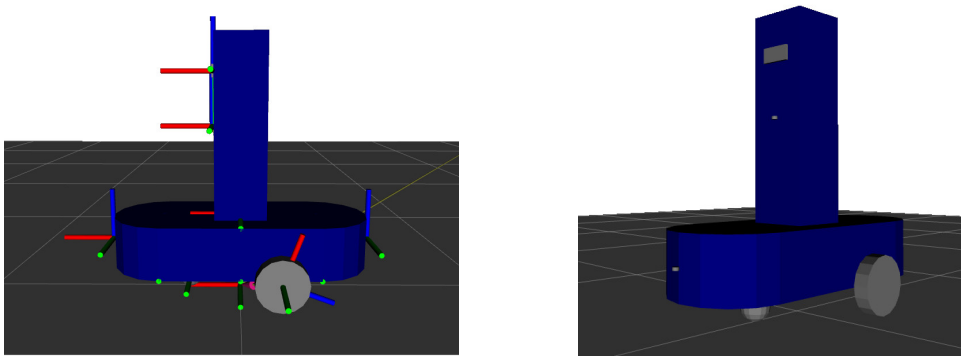
Das Robotersystem wird mithilfe von *Xacro*-Dateien modelliert, die anschließend als URDF in die Simulation integriert werden. Für die Sensorik kommen vorgefertigte *Gazebo*-Plugins zum Einsatz. Im Rahmen dieser Arbeit wurden drei Robotermodelle entworfen, die unterschiedliche Sensoren und Funktionalitäten abbilden.

Die Figure 3a zeigt das simpelste Robotermodell in *Rviz* aus der Seitenansicht. Dieses kann als Basisplattform eingesetzt werden. Erkennbar ist das Differenzialantriebssystem mit einem der beiden Antriebsräder sowie das Caster-Rad, das der Stabilisierung dient. Diese Grundkonfiguration bildet

die Basis für die weiteren Robotermodelle. Die Ansteuerung erfolgt über das *differential*-Plugin von Gazebo.

Die Figure 3b zeigt das Robotermodell in *Rviz* aus der Vorderansicht. Deutlich sichtbar ist die Sensorik: eine Kamera sowie zwei Ultraschallsensoren an der Front. Ein Ultraschallsensoren wird dabei für Hindernisse wie Tische oder Wände genutzt, der andere soll für Menschen in der Auf Grundlage dieser Basiskonfiguration wurden drei verschiedene Modellvarianten entwickelt, die sich wie folgt unterscheiden:

1. ein vereinfachtes Modell mit Ultraschallsensoren zur Hinderniserkennung,
2. ein erweitertes Modell mit Lidar für die Implementierung von SLAM und Navigation,
3. sowie ein Modell, das an den Mosro-Roboter angelehnt ist und den Eigenschaften des realen Roboters entspricht, für den in einer anderen Studienarbeit eine physische Umsetzung entwickelt wurde.



(a) Robotermodell in Rviz von der Seite

(b) Robotermodell in Rviz von vorne

Figure 3: Robotermodell in Rviz (Seiten- und Frontansicht)

Das vereinfachte Modell diente als Grundlage für ein Dashboard, das eine Teleoperation über eine Weboberfläche ermöglicht. Das erweiterte Modell wurde hingegen für die Entwicklung und Erprobung von Navigations- und Mapping-Funktionalitäten eingesetzt. Die Basis des Roboters besteht aus einem rechteckigen Korpus, an dem zwei Zylinder als Bumper angebracht sind. Auf dieser Basis befindet sich ein Turm, der die Kamera, Ultraschallsensoren und den Lidar-Sensor trägt.

Für die Simulation des Mosro-Roboters wurden Modelle aus den CAD-Dateien herangezogen. Auf dieser Grundlage entstand ein URDF-Modell mit Lidar, Kamera und Ultraschallsensoren, das in der Simulation eingesetzt wird.

Simulationsumgebung

Die virtuelle Lernumgebung ist so gestaltet, dass typische Szenarien wie Klassenzimmer oder Labore realistisch nachgebildet werden. Der Raum wurde mithilfe des Room-Tools in Gazebo Classic erstellt und stellt eine vereinfachte Nachbildung des DHBW-Gebäudes in Heidenheim dar. Die zugehörige Karte wurde durch SLAM erzeugt, indem der Roboter manuell durch den Raum gesteuert wurde. Als Sensorik kam dabei ein 2D-Lidar zum Einsatz.

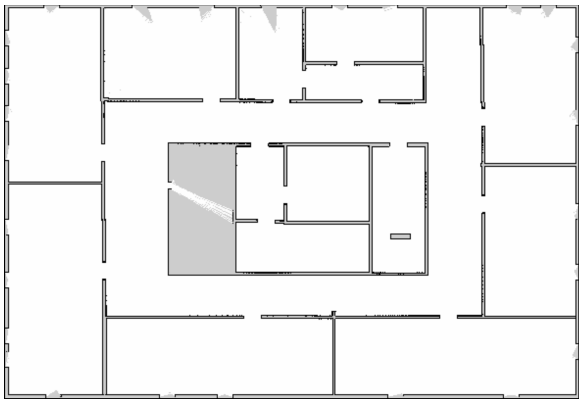
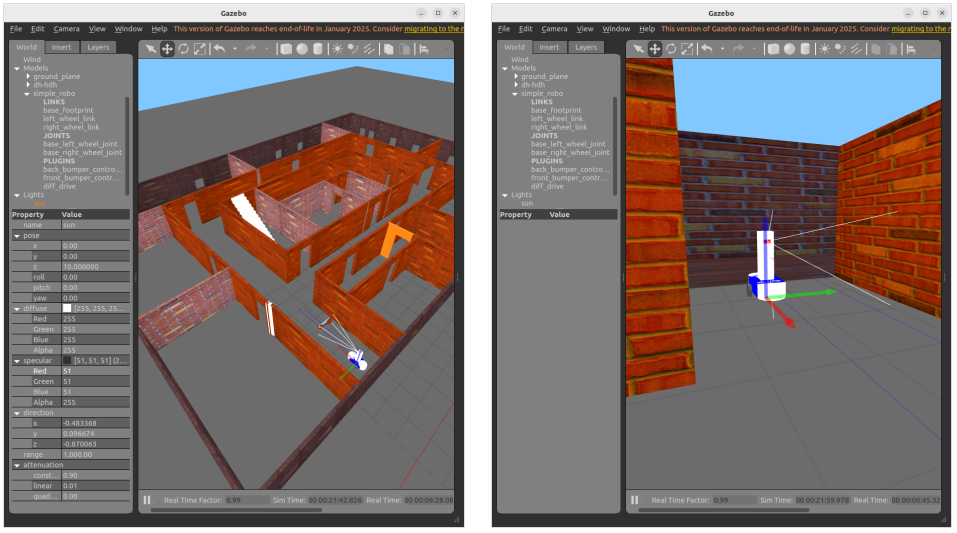


Figure 4: Von SLAM generierte Karte des Universitätsgebäudes

Die Figure 4 zeigt die durch SLAM generierte Karte des Universitätsgebäudes. Die Wände erscheinen als schwarze Linien, während weiße Bereiche die befahrbaren Flächen kennzeichnen. Die größeren Räume an den äußeren Rändern stellen Lehrräume dar, die kleineren Räume Büros und Besprechungsräume. Im mittleren Bereich sind die Toiletten und das Treppenhaus erkennbar, während der graue Bereich nicht begehbare Zonen markiert.

Die Figure 5a zeigt die in Gazebo implementierte Simulationsumgebung, in der die vereinfachte Nachbildung des DHBW-Gebäudes erkennbar ist. Das Robotermodell in Gazebo ist in Figure 5b dargestellt. Deutlich sichtbar sind die Sensoren: eine Kamera sowie Ultraschallsensoren an der Front.



(a) Simulationsumgebung in Gazebo (b) Robotermodell in Gazebo von vorne

Figure 5: Robotermodell in Gazebo

Die Sensoren des Roboters übermitteln ihre Daten über verschiedene ROS2-Topics an andere Systemkomponenten. Die Kamera publiziert Bilder über das Topic `/camera/image_raw`. Die Ultraschallsensoren geben Abstände zu Hindernissen über die Topics `/front_distance/out` und `/back_distance/out` aus. Diese Daten werden vom Controller verarbeitet, um Kollisionen zu vermeiden. In Gazebo wird die Sensordetektion visualisiert, sodass überprüft werden kann, ob die Sensoren korrekt arbeiten. Das Lidar publiziert seine Messdaten über das Topic `/scan`, welche für SLAM und Navigation genutzt werden.

Zur Unterstützung von Navigation und SLAM ist zudem eine IMU (Inertial Measurement Unit) erforderlich. Diese wird über ein Gazebo-Plugin simuliert und stellt ihre Daten über das Topic `/imu` bereit. Die IMU-Daten dienen dazu, die Position und Orientierung des Roboters im Raum zu bestimmen. Darüber hinaus versorgt das differential-Plugin das Topic `/odom` mit Odometrie-Daten, die die Bewegungen des Roboters beschreiben. Diese Sensordaten werden anschließend vom ROS2 Navigation Stack genutzt, um eine präzise Lokalisierung und Pfadplanung zu ermöglichen.

3.1 Übersicht ROS 2-Packages

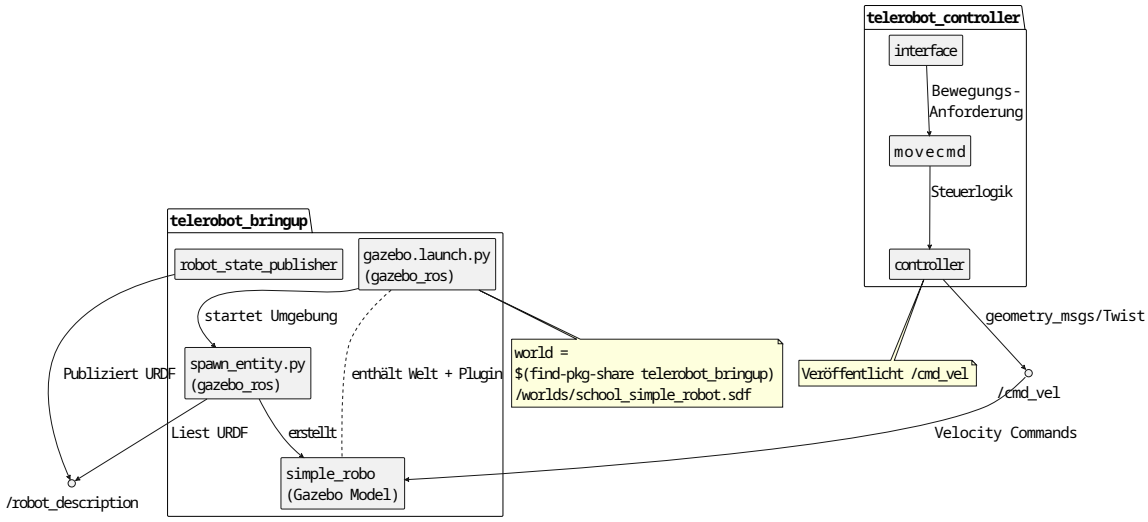


Figure 6: Übersicht der verwendeten ROS 2-Packages

Die Figure 8 zeigt die zentralen Komponenten des entwickelten Teleroboters, sowie deren Kommunikationsbeziehungen. Das System gliedert sich in zwei Hauptpakete: `telerobot_bringup` und `telerobot_controller`.

Im Paket `telerobot_bringup` werden die für den Simulationsstart benötigten Prozesse orchestriert. Der `robot_state_publisher` stellt die URDF-Beschreibung des Roboters über das Topic `/robot_description` bereit. Das Skript `spawn_entity.py` konsumiert diese Beschreibung und instanziiert daraus das Robotermodell (`simple_robot`) in der laufenden Gazebo-Instanz. Die Launch-Datei (`gazebo.launch.py`) startet den Simulator mitsamt Weltdefinition (`school_simple_robot.sdf`) und initialisiert die notwendigen Gazebo-Plugins. Das Modell empfängt anschließend Steuerkommandos über `/cmd_vel`.

Das Paket `telerobot_controller` kapselt die Eingabe- und Bewegungslogik. Die Node `interface` bildet die Schnittstelle zur Teleoperation (z. B. Web-Dashboard) und leitet Bewegungsanforderungen an `movecmd` weiter. Diese Komponente bereitet die Eingaben (Normalisierung, Begrenzung, ggf. Filterung) für die Steuerlogik auf. Die Komponente `controller` erzeugt daraus gültige `geometry_msgs/Twist` Nachrichten und veröffentlicht sie auf `/cmd_vel`. Damit wird eine klare Trennung zwischen Eingabeebene, Bewegungsaufbereitung und Ausgabe an die Simulationsumgebung erreicht.

Die Entkopplung über standardisierte Topics (insbesondere `/cmd_vel`

und `/robot_description`) erleichtert spätere Erweiterungen, wie den Austausch des Eingabekanals (z. B. autonome Navigation statt Teleoperation) oder die Ergänzung zusätzlicher Sensorik. Gleichzeitig bleibt die Simulationsschicht (Gazebo) schlank, da sie ausschließlich generische Velocity-Kommandos verarbeitet.

3.2 Teleoperation

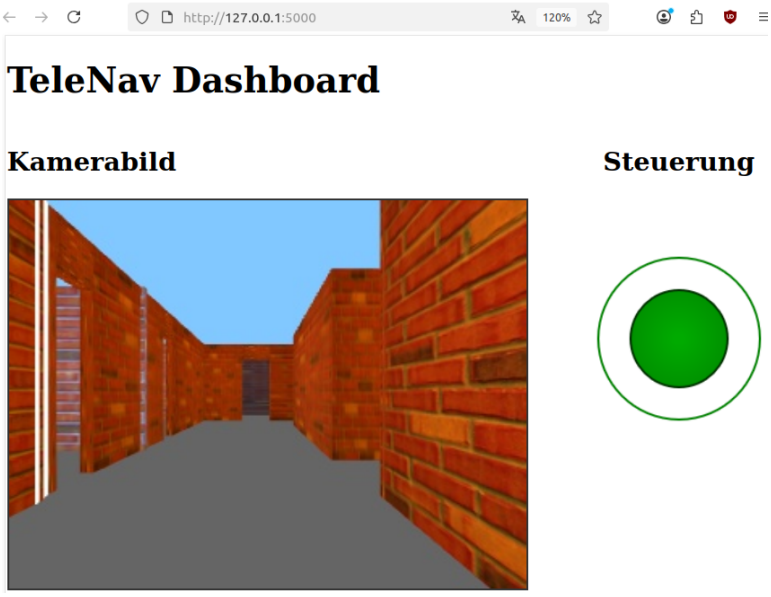


Figure 7: Übersicht der verwendeten ROS 2-Packages

Für die Teleoperation des Roboters wurde ein Dashboard entwickelt, das über einen Webbrowser zugänglich ist. Hier kann mithilfe von Tastatur oder der Maus über die Joystick-Funktion der Roboter gesteuert werden. Das Dashboard wurde in *HTML* und *CSS* geschrieben. Als Serverframework wurde *Flask* verwendet.

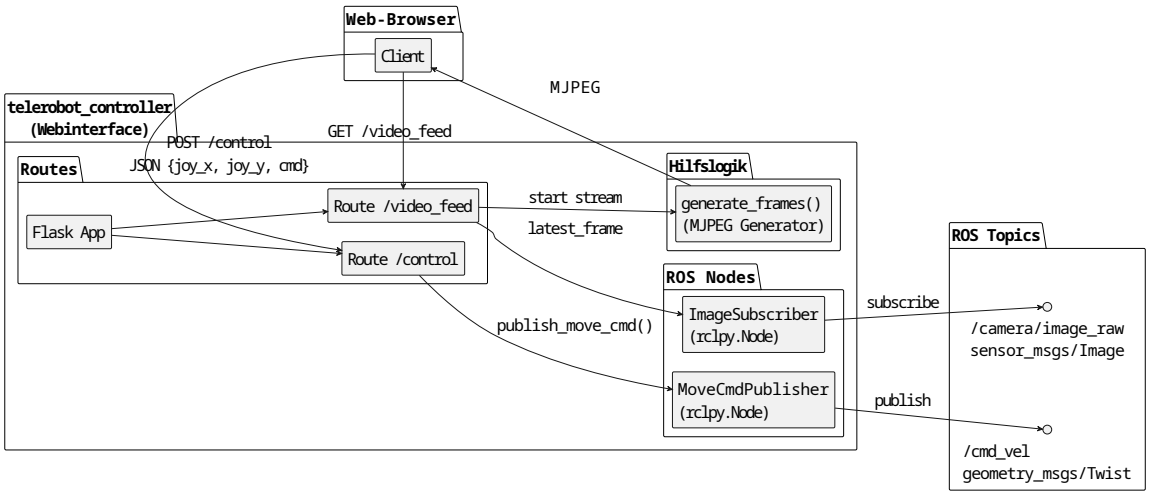


Figure 8: Übersicht der Komponenten des Webinterfaces

Die Figure 8 zeigt die einzelnen Komponenten der Teleoperation vom Webbrowser bis hin zur Ausführung der Bewegungsbefehle im ROS 2-System. Der Client kommuniziert dabei über zwei unterschiedliche Arten von HTTP-Anfragen. Zum einen wird eine GET-Anfrage an die Route `/video_feed` gesendet, um den Videostream zu initialisieren. Ändert sich das Bild der Kamera vom Roboter übermittelte Kamerabild, so wird dieses automatisch an den Client weitergeleitet. Zum anderen überträgt der Joystick über eine POST-Anfrage an die Route `/control` die Bewegungsbefehle.

Die Route `/video_feed` liefert einen MJPEG-Stream zurück, der aus fortlaufend erzeugten JPEG-Einzelbildern besteht. Diese werden serverseitig durch die Funktion `generate_frames()` bereitgestellt. Die Bilddaten stammen aus dem ROS-Kamera-Topic `/camera/image_raw`, die vom Image-Subscriber abonniert und intern zwischengespeichert wird. Durch diese Architektur entsteht eine Entkopplung zwischen der asynchronen ROS-Bilderfassung mit der Sensorfrequenz und dem HTTP-Streaming für das Rendering im Browser.

Die Steuerlogik der Bewegungen läuft hingegen über die Route `/control`. Hierbei werden eingehende JSON-Payloads zunächst validiert, etwa in Bezug auf den Wertebereiche und Plausibilität der Eingaben, und anschließend an den MoveCmdPublisher weitergeleitet. Dieser übersetzt die abstrahierten Joystick-Eingaben der Variablen `joy_x` für die Rotation und `joy_y` für die Translation in `geometry_msgs/Twist`-Nachrichten, die auf dem ROS-Topic `/cmd_vel` veröffentlicht werden. Das Differenzialantriebs-Plugin von Gazebo

konsumiert diese Nachrichten und setzt die gegebenen Geschwindigkeiten direkt in Bewegung um. Die Latenz der Befehlsübertragung hängt dabei vor allem von der Netzwerkübertragung (HTTP-Roundtrip) und dem Scheduling der ROS-Callback-Queue ab, während die Videolatenz zusätzlich durch die JPEG-Kodierung und das Rendering im Browser beeinflusst wird.

Die klare Trennung der Verantwortlichkeiten, von der HTTP-Schicht über die Frame-Generierung bis hin zum ROS-Publishing, erhöht die Wartbarkeit des Gesamtsystems. So lässt sich das Frontend beispielsweise problemlos durch andere Eingabeschnittstellen wie eine Gamepad-API oder eine Touch-Oberfläche ersetzen, ohne die Bewegungslogik anpassen zu müssen. Ebenso kann das System um zusätzliche Sensorkanäle wie Tiefenbilder erweitert werden. Darüber hinaus bietet die Architektur die Möglichkeit, den MoveCmdPublisher künftig mit einem Sicherheitslayer, etwa zur Kollisionsvermeidung vor der Publikation, oder mit einem Modusumschalter zwischen Teleoperation und Autonomie auszustatten. Auf diese Weise entsteht eine schlanke und zugleich erweiterbare Teleoperationspipeline, die konsequent auf Standardmechanismen von Flask und ROS 2 setzt und ohne proprietäre Protokolle auskommt.

3.3 Erstellung und Konfiguration der Simulation

Zum starten der Simulation wird eine launch-Datei verwendet. Diese ist in Listing 1 dargestellt. Jede Launch-Datei ist ähnlich strukturiert. Die ersten Zeilen sind Variablen-Deklarationen, die den Pfad zur URDF-Datei, Weltdefinition und RViz-Konfigurationsdatei enthalten. Anschließend wird die Gazebo-Launch-Datei eingebunden, die den Simulator mitsamt Weltdefinition startet. Daraufhin wird der robot_state_publisher gestartet, der die URDF-Beschreibung des Roboters über das Topic /robot_description bereitstellt. Das Skript spawn_entity.py konsumiert diese Beschreibung und instanziiert daraus das Robotermodell (simple_robo) in der laufenden Gazebo-Instanz.

```
1 <launch>
2   <let name="urdf_path" value="$(find-pkg-share␣
       telerobot_description)/urdf/simple_robot.urdf.xacro"/>
3   <let name="rviz_config_path" value="$(find-pkg-share␣
       telerobot_bringup)/rviz/urdf_config.rviz"/>
4
5   <include file="$(find-pkg-share␣
       gazebo_ros)/launch/gazebo.launch.py">
6     <arg name="world" value="$(find-pkg-share␣
         telerobot_bringup)/worlds/school_simple_robot.sdf"/>
```

```
7   </include>
8
9   <node pkg="robot_state_publisher"
        exec="robot_state_publisher">
10    <param name="robot_description" command="xacro␣$(var␣
        urdf_path)"/>
11  </node>
12
13  <node pkg="gazebo_ros" exec="spawn_entity.py"
14        args="-topic␣robot_description␣-entity␣
            simple_robo"/>
15
16  <node pkg="rviz2" exec="rviz2" output="screen"
17        args="-d␣$(var␣rviz_config_path)"/>
18 </launch>
```

Listing 1: ROS 2 Launch-Datei (Gazebo, URDF, RViz)

Abschließend wird RViz gestartet, um die Sensorik und Umgebung des Roboters zu visualisieren. Die unterschiedlichen Roboter haben jeweils eigene Launch-Dateien, die sich nur in der URDF-Datei und der Weltdefinition unterscheiden. Folgende Bezeichnungen werden in den Launch-Dateien verwendet:

- **simple_robot:** vereinfachtes Modell mit Ultraschallsensoren zur Hinderniserkennung
- **my_robot:** erweitertes Modell mit Lidar für die Implementierung von SLAM und Navigation
- **mosro:** Das Modell, das an den Mosro-Roboter angelehnt ist und den Eigenschaften des realen Roboters entspricht

3.4 Modellierung des Roboters

Die Modellierung des Roboters beginnt stets mit der Definition einer Roboterbasis. In dieser Basiskonfiguration werden die grundlegenden Elemente des Roboters festgelegt, wie beispielsweise die Antriebsräder und das Caster-Rad. Die Listing 2 zeigt einen Ausschnitt der Xacro-Datei, in der die Roboterbasis beschrieben wird. Dabei wird ein sogenannter TF (Transform)-Baum definiert, der die Beziehungen zwischen den einzelnen Links und Joints des Roboters abbildet.

Jede Base-Datei beginnt mit Variablen-Deklarationen, die die Abmessungen des Roboters festlegen. Anschließend wird ein `base_footprint`-

Link definiert, der als Referenzpunkt für die Position des Roboters dient. Dieser befindet sich auf Bodenhöhe zwischen den Antriebsrädern, also am unteren Mittelpunkt des Roboters. Darauf aufbauend wird der `base_link` definiert, der den Korpus des Roboters darstellt. In diesem Beispiel besteht er aus einem Quader mit den Abmessungen des Roboters. Damit das Modell in Gazebo verwendet werden kann, muss jeder Link mit Kollisionen und Trägheit versehen werden. Abschließend wird ein fixed Joint definiert, der den `base_link` mit dem `base_footprint` verbindet. Die feste Position dieses Joints wird über das origin-Tag festgelegt.

Damit sich der Roboter bewegen kann werden Räder benötigt. Damit sich diese drehen können werden Joints verwendet, die sich drehen können. Hierbei kommen *continuous* Joints zum Einsatz, die eine unendliche Rotation ermöglichen. Wichtig ist dabei, die korrekte Achse für die Rotation auszuwählen, um die gewünschte Bewegungsfreiheit zu gewährleisten.

In der geometry wird die Form des Links definiert. Hierbei können verschiedene geometrische Formen wie Box, Cylinder oder Sphere verwendet werden. In der Mosro Xacro-Datei wird zudem ein Mesh verwendet, um die komplexe Form des Roboters darzustellen. Der nächste Tag in der Link-Definition ist die collision. Diese definiert die physikalische Kollision des Links. Für Meshes wird in der Regel eine vereinfachte Form (z.B Zylinder, Box) verwendet, um die Berechnung zu erleichtern. Die letzte Komponente eines Links ist die *inertial*. Diese definiert die Trägheit des Links, die für die physikalische Simulation in Gazebo erforderlich ist.

In Listing 2 wird diese mithilfe eines Makros aus der common_properties-Datei (Listing 3) generiert. Als nächstes wird der Joint definiert, der den Link mit dem `base_link` verbindet. Hierbei wird der Name des Joints, der Typ (z.B. fixed, continuous), die Parent- und Child-Links sowie die Achse der Rotation festgelegt. Die Position des Joints wird ebenfalls über das origin-Tag definiert.

```
1  <?xml version="1.0"?>
2  <robot xmlns:xacro="http://www.ros.org/wiki/xacro">
3
4  <!-- Robot Base-->
5  <xacro:property name="base_length" value="0.6" />
6  <xacro:property name="wheel_length" value="0.05" />
7
8  <link name="base_footprint" />
9  <link name="base_link">
10     <visual>
11         <geometry>
12             <box size="${base_length} ${base_width} 
```

```
                ${base_height}" />
13          </geometry>
14          <origin xyz="0␣0␣0" rpy="0␣0␣0" />
15          <material name="blue" />
16      </visual>
17      <collision>
18          <geometry>
19              <box size="${base_length}␣${base_width}␣
                    ${base_height}" />
20          </geometry>
21          <origin xyz="0␣0␣0" rpy="0␣0␣0" />
22      </collision>
23      <origin xyz="0␣0␣${base_height␣/␣2.0}" rpy="0␣0␣0" />
24      <xacro:box_inertia m="5.0" r="${base_width/2}"
           h="${base_width}"
25          xyz="0␣0␣${base_width␣/␣2.0}" rpy="0␣0␣0" />
26  </link>
27
28  <link name="base_right_wheel_joint">
29          <visual>
30          </visual>
31          <collision>
32          <xacro:cylinder_inertia />
33      </link>
34  </xacro:macro>
35
36  <joint name="base_joint" type="fixed">
37      <parent link="base_footprint" />
38      <child link="base_link" />
39      <origin xyz="0␣0␣${wheel_radius}" rpy="0␣0␣0"/>
40  </joint>
41
42  <joint name="base_right_wheel_joint" type="continuous">
43      <parent link="base_link" />
44      <child link="right_wheel_link" />
45      <origin xyz="0␣0␣0" rpy="0␣0␣0" />
46      <axis xyz="0␣1␣0" />
47  </joint>
48  </robot>
```

Listing 2: Minimale Robotermodellierung zur Veranschaulichung

Zusätzlich zur Roboterdefinition wird eine common_properties-Datei erstellt, die allgemeine Eigenschaften wie Farben, Materialien und Maße enthält. Darüber hinaus können darin wiederverwendbare Makros definiert

werden, die in verschiedenen Robotermodellen zum Einsatz kommen. Dies vereinfacht die Modellierung und gewährleistet Konsistenz zwischen den Robotern, da URDF-Dateien ansonsten schnell unübersichtlich werden. Eine zentrale Aufgabe der common_properties-Datei ist unter anderem die automatische Generierung der Trägheitsmatrix. Diese kann auf Basis der Form, der Abmessungen und der Masse eines Links berechnet werden. In Listing 3 ist ein Beispiel für eine solche Datei dargestellt, die ein Material definiert und ein Makro zur Berechnung der Trägheitsmatrix eines Quaders enthält. Mithilfe des Materials kann hierbei die Farbe des Elements in Rviz bestimmt werden. Makros werden in Xacro mit dem Tag `<xacro:macro>` definiert, wobei Parameter übergeben werden können. Innerhalb des Makros lassen sich diese Parameter wie Variablen verwenden. Der Aufruf eines Makros erfolgt über das Tag `<xacro:makroname>`.

```
1  <?xml version="1.0"?>
2  <robot xmlns:xacro="http://www.ros.org/wiki/xacro">
3
4      <material name="blue">
5          <color rgba="0␣0␣0.5␣1" />
6      </material>
7
8      <xacro:macro name="box_inertia" params="m␣l␣w␣h␣xyz␣
           rpy">
9          <inertial>
10             <origin xyz="${xyz}" rpy="${rpy}" />
11             <mass value="${m}" />
12             <inertia ixx="${(m/12)␣*␣(h*h␣+␣l*l)}"
                   ixy="0" ixz="0"
13                       iyy="${(m/12)␣*␣(w*w␣+␣l*l)}"
                             iyz="0"
14                       izz="${(m/12)␣*␣(w*w␣+␣h*h)}" />
15         </inertial>
16     </xacro:macro>
17
18 </robot>
```

Listing 3: Beispiel einer common_properties-Datei

Um beide Dateien zu verbinden, wird in einer weiteren Xacro-Datei alle gemeinsamen Xacro-Dateien eingebunden. Diese wird dann in der Konfiguration der Launch-Datei verwendet, um die URDF-Beschreibung des Roboters zu generieren. Hierbei ist wichtig, dass ein für den Roboter geeigneter Name für die Datei gewählt wird, um Verwechslungen zu vermeiden. Die Listing 4 zeigt ein Beispiel, wie die common_properties und roboter Basis-Datei

eingebunden wird.

```
<?xml version="1.0"?>
<robot name="simple_robot"
   xmlns:xacro="http://www.ros.org/wiki/xacro">

    <!-- Robot Base-->
    <xacro:include filename="common_properties.xacro" />
    <xacro:include filename="robot_base.xacro" />

</robot>
```

Listing 4: Einbindung der common_properties-Datei

3.5 Integration virtueller Sensoren

Die Integration virtueller Sensoren in das Robotermodell erfolgt über Gazebo-Plugins, die in der URDF-Datei des Roboters eingebunden werden. Diese Plugins simulieren das Verhalten realer Sensoren und ermöglichen es, Sensordaten innerhalb der Simulationsumgebung zu generieren. Die Listing 5 zeigt ein Beispiel für die Einbindung eines Lidarsensors in die URDF-Datei. Hierbei wird zunächst ein Link namens `lidar_link` definiert, der den physischen Teil des Sensors darstellt. Anschließend wird das Gazebo-Plugin für den Lidar-Sensor eingebunden, das die Sensordaten generiert und über ein ROS-Topic veröffentlicht. Wichtig ist, dass der Link und das Plugin korrekt miteinander verbunden sind, damit der Sensor im Kontext des Roboters funktioniert. Die Parameter des Plugins, wie der Name des Topics, die Update-Rate und der Frame-Name, müssen entsprechend der Anwendung angepasst werden.

```
<link name="lidar_link">
    <visual></visual>
    <collision></collision>
</link>
<gazebo reference="lidar_link">
    <sensor name="lidar" type="ray">
        <always_on>true</always_on>
        <visualize>true</visualize>
        <update_rate>5</update_rate>
        <ray>
        <scan>
            <horizontal>
            <samples>360</samples>
            <resolution>1.000000</resolution>
```

```
15              <min_angle>0.000000</min_angle>
16              <max_angle>6.280000</max_angle>
17              </horizontal>
18          </scan>
19          <range>
20              <min>0.120000</min>
21              <max>3.5</max>
22              <resolution>0.015000</resolution>
23          </range>
24          <noise>
25              <type>gaussian</type>
26              <mean>0.0</mean>
27              <stddev>0.01</stddev>
28          </noise>
29          </ray>
30          <plugin filename="libgazebo_ros_ray_sensor.so"
               name="lidar_plugin">
31              <topic_name>scan</topic_name>
32              <output_type>sensor_msgs/LaserScan</output_type>
33              <frame_name>lidar_link</frame_name>
34          </plugin>
35      </sensor>
36  </gazebo>
```

Listing 5: Einbindung der common_properties-Datei

Jeder Gazebo Sensor benötigt einen Link, auf das referenziert werden kann. Innerhalb des sensor-Tags wird der Typ des Sensors (z.B. ray für Lidar) und dessen Parameter definiert. Im Plugin-Tag wird das Gazebo-Plugin eingebunden, das die Sensordaten generiert und über ein ROS-Topic veröffentlicht. Die Parameter des Plugins, wie der Name des Topics, die Update-Rate und der Frame-Name, müssen entsprechend der Anwendung angepasst werden.

3.6 Einbindung in die ROS 2-Architektur

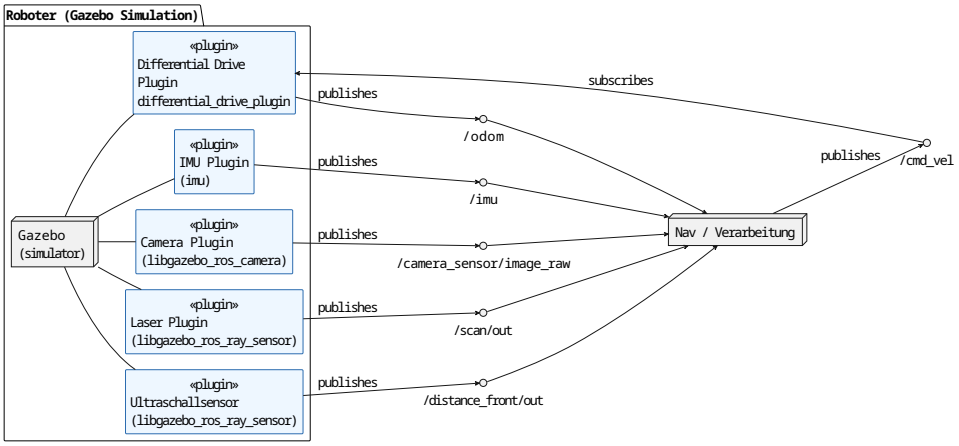


Figure 9: Übersicht der Einbindung der Sensoren in die ROS 2-Architektur

Um mit den Sensoren zu kommunizieren, publizieren diese ihre Daten über ROS 2-Topics. Andere Nodes können diese Topics abonnieren, um die Sensordaten weiterzuverarbeiten. Die Figure 10 veranschaulicht die Einbindung der Gazebo-Simulation in die ROS 2-Architektur.

Die Sensoren des Roboters übertragen ihre Daten über verschiedene ROS 2-Topics an andere Systemkomponenten:

- Die Kamera publiziert Bilder über das Topic `/camera/image_raw`.
- Die Ultraschallsensoren geben Abstände zu Hindernissen über die Topics `/front_distance/out` und `/back_distance/out` aus. Diese Informationen werden vom Controller genutzt, um Kollisionen zu vermeiden.
- Das Lidar stellt seine Messdaten über das Topic `/scan` bereit, die für SLAM und Navigation verwendet werden können.
- Zusätzlich wird zur Unterstützung von Navigation und SLAM eine IMU (Inertial Measurement Unit) benötigt. Diese wird über ein Gazebo-Plugin simuliert und publiziert ihre Daten auf dem Topic `/imu`.

Damit sich der Roboter fortbewegen kann, werden Steuerbefehle in Form von Nachrichten an das Topic `/cmd_vel` gesendet. Diese Nachrichten stammen entweder vom Nav2-Stack oder von dem im Rahmen dieser Arbeit en-

twickelten Controller. Es handelt sich dabei um **Twist**-Nachrichten, die jeweils zwei Vektoren enthalten: einen für die lineare und einen für die angulare Geschwindigkeit.

4 Implementierung der Funktionalitäten

Die Implementierung der Funktionalitäten des Telepräsenzroboters umfasst die Steuerung und Teleoperation. Beide nutzen Interfaces, die in `telerobot_interfaces` definiert sind. Diese Interfaces kapseln die Kommunikation mit den ROS2-Topics und bieten eine abstrahierte Schnittstelle für die Steuerung des Roboters. Dabei werden feste Typen verwendet, um die Konsistenz und Wartbarkeit des Codes zu gewährleisten. Die Steuerung des Roboters erfolgt über die Node `controller`, die die Bewegungsbefehle verarbeitet und an die Simulationsumgebung weiterleitet. Die Teleoperation wird über die Node `interface` realisiert, die eine Weboberfläche bereitstellt, über die der Roboter gesteuert werden kann. Beide Nodes sind in Python implementiert und nutzen die ROS2-Python-Bibliothek `rclpy`.

4.1 Teleoperation

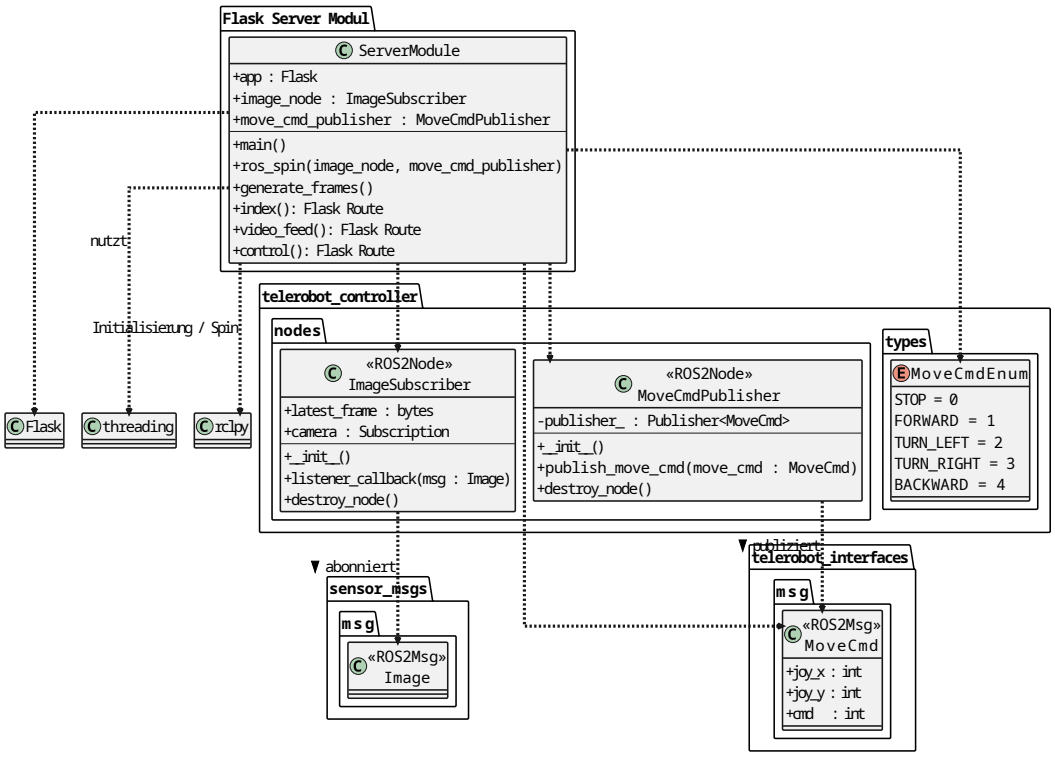


Figure 10: Zeigt ein Klassendiagramm der Teleoperationsschnittstelle

Im Mittelpunkt der Teleoperation-Node steht der *Flask*-Server, der die HTTP-Anfragen des Webinterfaces verarbeitet. Die Klasse `ImageSubscriber` abonniert das Kamera-Topic `/camera/image_raw` und speichert das zuletzt empfangene Bild als `latest_frame`. Die Klasse `MoveCmdPublisher` veröffentlicht Steuerbefehle über das Topic `/move_cmd`. Alle Komponenten werden in der Klasse `ServerModule` zusammengeführt, die zusätzlich die Flask-Routen für das Webinterface bereitstellt. Figure 10 zeigt das Klassendiagramm der Teleoperationsschnittstelle.

Da die Node zwei unterschiedliche Aufgaben erfüllt, müssen *Flask*-Server und ROS-Komponenten in separaten Threads ausgeführt werden. Dies übernimmt die Klasse `ServerModule`, die beide Instanzen initialisiert und parallel betreibt. Die Flask-Routen verarbeiten Bilddaten und Steuerbefehle und leiten diese an die jeweiligen Klassen weiter. Die Route `/video_feed` liefert einen MJPEG-Stream, der aus fortlaufend erzeugten JPEG-Einzelbildern

besteht. Diese werden serverseitig durch die Funktion `generate_frames()` erzeugt, die auf die globale Variable des `ImageSubscriber` zugreift, um das zuletzt empfangene Bild zu liefern. Die Steuerlogik der Bewegungen läuft hingegen über die Route `/control`. Hierbei werden sowohl die Befehle als auch die Wertebereiche der Variablen `joy_x` und `joy_y` validiert und anschließend an den `MoveCmdPublisher` weitergeleitet, der ebenfalls als globales Objekt bereitsteht.

Zu Beginn traten Probleme mit der Funktion `generate_frames()` auf, da diese zu viele Einzelbilder an den Stream übergab. Dies führt zu einer Überlastung des Browsers, der mit der großen Datenmenge nicht umgehen kann. Zur Lösung wird ein Vergleichs-Check implementiert, der ein neues Bild nur dann weitergibt, wenn es sich vom vorherigen unterscheidet. Dadurch reduzierte sich die Datenmenge erheblich, was zu einer stabileren Übertragung führt.

Ein ähnliches Problem bestand beim kontinuierlichen Publizieren der Steuerbefehle. Mithilfe eines clientseitigen Skripts in `JavaScript` wurde ein analoger Check eingebaut, sodass neue Befehle nur gesendet werden, wenn sich die Werte von `joy_x` oder `joy_y` tatsächlich geändert haben.

4.2 Steuerung

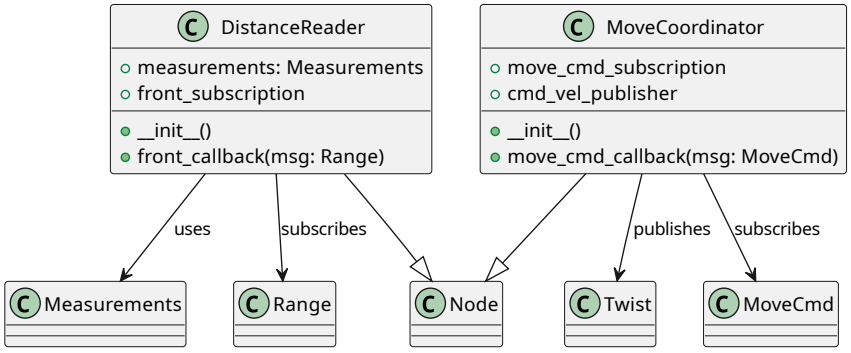


Figure 11: Zeigt ein Klassendiagramm der Steuerungskomponente

Die Figure 11 zeigt das Klassendiagramm der Steuerungskomponente. Die Steuerung ist als Mittelmann zwischen Teleoperation und Simulationsumgebung konzipiert. Sie empfängt Bewegungsbefehle von der Teleoperation und leitet sie je nach Zustand des Systems an den Roboter weiter. Stellt der Controller fest, dass ein Hindernis in der Nähe ist, werden die Be-

fehle entsprechend angepasst, um Kollisionen zu vermeiden. Die Klasse `DistanceReader` abonniert die Ultraschall-Topics `/front_distance/out` und `/back_distance/out` und speichert die zuletzt empfangenen Abstände in dem Attribut `measurements`.

Die Klasse `MoveCmdPublisher` ist für das Veröffentlichen von Bewegungsbefehlen zuständig. Sie empfängt die Befehle von der Teleoperation und leitet sie an den Roboter weiter.

4.3 SLAM-Integration

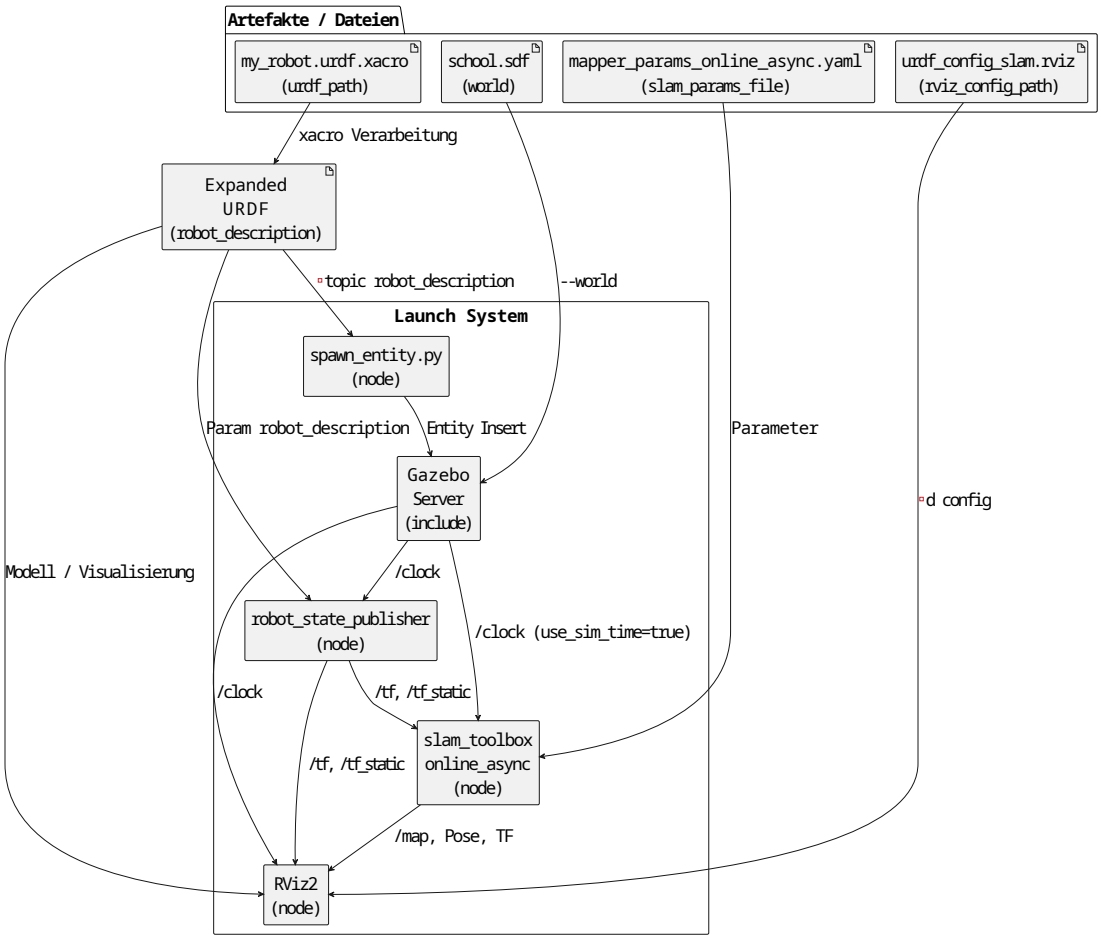


Figure 12: Zeigt ein Komponentendiagram der SLAM-Integration

Zur Integration von SLAM in die Simulationsumgebung wird die slam-toolbox verwendet. Sie bietet verschiedene Modi zur Kartenerstellung sowie zur Lokalisierung. Die zentralen Komponenten der SLAM-Integration sind in Figure 12 dargestellt.

Die Konfiguration erfolgt über die Datei `map_params_online_sync.yaml`, die für die Kartenerstellung im Online-Sync-Modus optimiert ist. Die Node `slam_toolbox` abonniert sowohl das Lidar-Topic `/scan` als auch die Odometrie-Daten vom Topic `/odom`. Auf Basis dieser Informationen wird eine Karte der Umgebung generiert und gleichzeitig die aktuelle Roboterposition bestimmt. Die resultierende Karte wird über das Topic `/map` veröffentlicht und kann in RViz visualisiert werden.

Die Lokalisierung des Roboters erfolgt durch die kontinuierliche Verarbeitung der Sensordaten in Kombination mit der Odometrie. Mit dem `map_saver` kann die Karte zusätzlich als Bilddatei gespeichert und für eine spätere Wiederverwendung exportiert werden. Eine gespeicherte Karte lässt sich anschließend in Nav2 einbinden, um eine autonome Navigation zu ermöglichen. Darüber hinaus stellt die Node `map_server` die erzeugte Karte über das Topic `/map` bereit, sodass andere Nodes darauf zugreifen können.

5 Fazit und Ausblick

Dieses Projekt hat gezeigt, dass sich mit `ROS 2 (Humble)` und `Gazebo Classic` eine modular erweiterbare Telepräsenz- und Experimentierplattform für hybride Lehrszenarien effizient aufbauen lässt. Kernbeiträge sind:

1. Ein klar strukturiertes Paketlayout zur Trennung von Bringup, Steuerlogik und Schnittstellen.
2. Drei aufeinander aufbauende Robotermodelle (*einfach*, *SLAM-fähig*, *referenznah zum Mosro-Design*).
3. Wiederverwendbare `Xacro`-Makros zur konsistenten Modellierung.
4. Eine Teleoperations-Architektur mit entkoppelter Video- und Steuerpipeline (`Flask` + `ROS 2 Topics`).
5. Die Integration von `slam_toolbox` für Online-Kartierung und Lokalisierung.
6. Dokumentation und Beispiele für eine Weiterentwicklung.

Die erzielte Struktur schafft eine belastbare Grundlage für spätere Erweiterungen wie autonome Navigation, Sicherheitsmechanismen oder zusätzliche Sensorik. Insgesamt wurde das Ziel einer flexiblen Simulationsbasis erreicht und technische Risiken für eine physische Umsetzung reduziert.

5.1 Perspektiven für Erweiterungen

Zukünftige Arbeiten könnten sich mit der Verbesserung der Simulation befassen, in der die neue Gazebo Harmonic Version verwendet wird. Zudem könnte das Docking Plugin verwendet werden um eine automatische Rückkehr zur Ladestation zu ermöglichen. Die Teleoperationsschnittstelle ließe sich durch alternative Eingabemethoden wie Gamepads ergänzen. Die Steuerlogik könnte um Sicherheitslayer zur Kollisionsvermeidung erweitert werden. Die Integration von Nav2 für autonome Navigation auf Basis gespeicherter Karten wäre ein weiterer Schritt. Schließlich könnte die Plattform um weitere Sensoren wie Tiefenkameras oder einem 3D Lidar ergänzt werden. Diese Erweiterungen würden die Funktionalität und Flexibilität des Systems weiter steigern und neue Anwendungsfälle erschließen.

Ebenso wäre die Implementierung der Software in ein echtes Robotersystem ein sinnvoller nächster Schritt. Hierbei könnten die im Rahmen dieser Arbeit entwickelten Konzepte und Strukturen als solide Basis dienen und an die spezifischen Anforderungen der Hardware angepasst werden. Das entstandene Repository bietet dabei alle nötigen Dokumentation um mit dem Umgang von ROS 2 und Gazebo vertraut zu werden.

References

Ayala, Angel, Francisco Cruz, Diego Campos, Rodrigo Rubio, Bruno Fernandes, and Richard Dazeley. 2020. "A Comparison of Humanoid Robot Simulators: A Quantitative Approach." `https://arxiv.org/abs/2008.04627`.

Macenski, Steven, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. 2022. "Robot Operating System 2: Design, architecture, and uses in the wild." *Science Robotics* 7 (66). `10.1126/scirobotics.abm6074`.

McNamee, Ty and Austin Horn. 2023. "Faculty Development at Community Colleges in U.S. Rural Contexts." *Journal of Education Human Resources* 42. `10.3138/jehr-2023-0051`.