Simple ROS Configurator

BACHELORARBEIT

Hägele Tobias

Eingereicht am 28. Februar 2022

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Betreuer:
Prof. Dr. Dirk Riehle, M.B.A.

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 28. Februar 2022

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see https://creativecommons.org/licenses/by/4.0/

Erlangen, 28. Februar 2022

Abstract

Robot Operating System (ROS) is a framework for robots used in industrial and private contexts. It is constantly being further developed by a large and active community. In order to use the framework to develop a robot system that is tailored to individual needs, you have to create your own projects. The creation, further development and maintenance of a project are not easy due to the large number of functions and the complexity of robot systems. Within the scope of this bachelor thesis, a program collection and graphical user interface are developed under the name ros2_ui, which is intended to facilitate the creation, extension and use of projects within the ROS framework. The concept of the software is explained. Furthermore it is shown which tasks usually occur within a ROS project and how they are facilitated by ros2_ui. The functions are demonstrated with an example.

Zusammenfassung

ROS ist ein Framework für im industriellen und privaten Kontext genutzte Robotersysteme. Es wird von einer großen und aktiven Gemeinschaft stetig weiterentwickelt. Um mithilfe des Frameworks ein Robotersystem zu entwickeln, das auf die individuellen Bedürfnisse zugeschnitten, ist müssen eigene Projekte erstellt werden. Die Erstellung, Weiterentwicklung und Pflege eines Projektes sind aufgrund der großen Fülle an Funktionen und der Komplexität von Robotersystemen nicht einfach. Im Rahmen der vorliegenden Bachelorarbeit wurden unter dem Namen ros2_ui eine Programmsammlung und grafische Oberfläche entwickelt, die das Erstellen, Erweitern und Nutzen von Projekten innerhalb des ROS-Frameworks erleichtert. Das Konzept der Software wird erläutert. Darüber hinaus wird aufgezeigt, welche Aufgaben im Rahmen eines ROS-Projektes üblicherweise anfallen und wie diese durch ros2_ui erleichtert werden. Die Funktionen werden an einem Beispiel demonstriert.

Inhaltsverzeichnis

1 Einleitung **1**

2 Problemstellung **3**
2.1 Komplizierte Projektverwaltung 3
2.2 Fehlende Übersicht in komplexen Projekten 3
2.3 Nichtexistenz ähnlicher Projekte für ROS2 4

3 Ziel **5**
3.1 Software, die den Umgang mit ROS2 vereinfacht 5
3.2 Grafische Oberfläche 5

4 Lösungskonzept **7**
4.1 Funktionsbeschreibung der Komponenten 7
4.1.1 Programmsammlung 7
4.1.2 Graphical User Interface (GUI) 7
4.2 Installation 8
4.3 Technologien und Sprachen 8
4.3.1 Python 3 8
4.3.2 Shellskript 8
4.3.3 HTML, CSS, JavaScript 9
4.3.4 WebSocket 9
4.3.5 Webserver (Flask) 9
4.3.6 JavaScript Object Notation (JSON) 9
4.4 Code-Struktur: Clean Architecture 9

5 Implementierung **11**
5.1 Projektverwaltung 11
5.1.1 ROS2: Arbeitsverzeichnis 12
5.1.2 ros2_ui: Arbeitsverzeichnis 13
5.1.3 ROS2: *Paket* anlegen 14
5.1.4 ros2_ui: *Projekt* anlegen 14
5.1.5 ROS2: *Paket*daten bearbeiten 14

5.1.6 ros2_ui: *Projekt*daten bearbeiten 15
5.1.7 ROS2: *Paket* laden und speichern 15
5.1.8 ros2_ui: *Projekt* laden und speichern 15
5.1.9 ROS2: *Paket*inhalt bearbeiten 15
5.1.10 ros2_ui: *Projekt*inhalt bearbeiten 16
5.1.11 ROS2: *Paket* bauen 16
5.1.12 ros2_ui: *Projekt* bauen 16
5.2 Publish-Subscribe 16
5.2.1 ROS2: *Knoten* 17
5.2.2 ROS2: Publish-Subscribe-Mechanismus 18
5.2.3 ros2_ui: *Knoten* 19
5.2.4 ros2_ui: *Publisher* 19
5.2.5 ros2_ui: *Subscriber* 20
5.2.6 ros2_ui: *Topic* 20
5.2.7 ros2_ui: *Publisher* & *Subscriber* 21
5.3 Start-Stop-Mechanismus 21
5.3.1 ROS2: *Paket* ausführen 21
5.3.2 ros2_ui: Projekt ausführen 21
5.4 Eigene *Knoten* 22
5.5 Vorgefertigte *Knoten* 22
5.5.1 *Subscriber*: „-led-“ 23
5.5.2 *Subscriber*: „-stdout-“ 23
5.5.3 *Publisher*: „-button-“ 23

6 Demonstration: Haustür 25
6.1 Geschichte 25
6.2 Akteure 25
6.2.1 Klingelknopf 26
6.2.2 Klingel 26
6.2.3 Alice 26
6.2.4 Tür 26
6.3 *Projekt*-Verwaltung 27
6.3.1 *Projekt* anlegen 27
6.3.2 „Klingelknopf“ hinzufügen 27
6.3.3 „Klingel-LED“ hinzufügen 27
6.3.4 „Tür“ hinzufügen 28
6.3.5 *Projekt* speichern 28
6.3.6 *Projekt* bauen 28
6.3.7 *Projekt* ausführen 28
6.4 Programmablauf 29

7 Bewertung 33

8 Schluss **35**

Appendices **37**
A Entwicklungsumgebung 39
A.1 Hardware 39
A.2 Software 39
B Browserkompatibilität 39
C Nachrichtentypen 39
D Code aus der Demonstration 40
E Aufgabenstellung der Bachelorarbeit 43
F Veröffentlichung des Codes 45

Literaturverzeichnis **47**

Abbildungsverzeichnis

5.1 Bildschirmfoto - *Projekt*-Seite 12
5.2 Bildschirmfoto - Dialog zur Erstellung eines *Projekts* 14
5.3 Bildschirmfoto - Maske zur Bearbeitung der Metadaten eines *Projekts* 15
5.4 Bildschirmfoto - Dropdown für das Laden eines *Projekts* 15
5.5 Bildschirmfoto - Knöpfe zum Speichern, Bauen und Ausführen . . 15
5.6 Bildschirmfoto - Bau-Log in der ros2_ui 17
5.7 Node-Modell 18
5.8 Publish-Subscribe-Mechanismus 18
5.9 Bildschirmfoto - Dialog zur Erstellung eines *Publishers* 19
5.10 Bildschirmfoto - Dialog zur Erstellung eines *Publishers* 20
5.11 Bildschirmfoto - *Publisher* und *Subscriber* in ros2_ui 21
5.12 Bildschirmfoto - Ausführungslog in der ros2_ui 22

6.1 Bild von RaspberryPi und Steckplatine 25
6.2 *Publisher* für den „Klingelknopf“ hinzufügen 27
6.3 *Publisher* für den „Klingelknopf“ hinzufügen 27
6.4 *Publisher* für den „Klingelknopf“ hinzufügen 28
6.5 *Subscriber* für die „Klingel-LED“ hinzufügen 29
6.6 *Subscriber* für die „Tür“ hinzufügen 30
6.7 Fertiges *Projekt* 30
6.8 Vor der Initialisierung - Alle *Knoten* wurden gestartet. 31
6.9 Nach der Initialisierung - Die *Knoten* haben sich beim ROS2-Framework für das Thema „bell“ registriert. 31
6.10 Nach Knopfdruck - Der *Publisher* „pushbutton“ versendet die Nachricht „pressed“. 31
6.11 ROS2 verteilt die Nachricht „pressed“ an die Subscriber. 31
6.12 Nach dem Empfang - Die *Knoten* arbeiten ihre Programme ab und warten auf weitere Nachrichten. 31

Tabellenverzeichnis

6.1 Zustände der „Tür“ als ASCII-Art 26

Abkürzungsverzeichnung

ROS Robot Operating System

ROS2 Robot Operating System - Version 2

OSRF Open Source Robotics Foundation

ASCII-Art American Standard Code for Information Interchange

UI User Interface (Nutzerschnittstelle)

GUI Graphical User Interface

JSON JavaScript Object Notation

1 Einleitung

Roboter werden in unserer modernen Lebenswelt immer wichtiger. Auf der einen Seite wachsen die industriellen Anwendungsfelder, auf der anderen Seite erfreuen sie sich auch in privaten Haushalten zunehmender Beliebtheit. Dabei decken sie ein breites Spektrum von Funktionalitäten ab und erfüllen unterschiedlich komplexe Anforderungen.

Die verschiedenen Funktionen werden über entsprechende Software gesteuert. Diese muss, zugeschnitten für den jeweiligen Roboter, das Zusammenspiel zwischen ihm und der Umwelt ermöglichen und die einzelnen Komponenten untereinander koordinieren.

ROS ist ein quelloffenes Framework, das genau solche Software beinhaltet. Es ist anwendbar für persönliche Roboter und Industrieroboter. Es wird von der gemeinnützigen Organisation Open Source Robotics Foundation (OSRF) koordiniert, gepflegt und weiterentwickelt. ROS bietet Hardwareabstraktion, Gerätetreiber, Nachrichtenvermittlung, Implementierungen häufig verwendeter Komponenten und eine Paketverwaltung.

Eine besondere Rolle spielt dabei die Nachrichtenvermittlung. Diese koordiniert die systemübergreifende Kommunikation verschiedener Komponenten in einem Roboter-System über ein gemeinsames Nachrichten-System.

Von ROS existieren zwei fast unabhängige Projekte, ROS (bzw. ROS1) und dessen Nachfolger ROS2. ROS2 ist im weitesten Sinne der Nachfolger von ROS1, implementiert aber einige Funktionalitäten schon von Grund auf, die im Design von ROS1 nicht vorgesehen waren, So ist ROS2 beispielsweise grundsätzlich echtzeitfähig. ROS1 wurde im Jahr 2007 veröffentlicht und seit der ersten Version von ROS2 im Jahr 2017 koexistieren die beiden Projekte. Im Jahr 2020 wurde die letzte Hauptversion von ROS1 veröffentlicht, die als noch bis 2025 mit Aktualisierungen versorgt wird. Diese Arbeit bezieht sich wegen der absehbaren Beendigung der Unterstützung von ROS1 ausschließlich auf ROS2.

Das Ziel dieser Arbeit ist die Entwicklung einer grafischen Benutzeroberfläche, mit deren Hilfe man ROS-basierte Projekte einfach anlegen, verwalten, bearbeiten,

beobachten, starten und beenden kann. Die entwickelte grafische Oberfläche, und auch das Programm dahinter tragen den Namen ros2_ui.

2 Problemstellung

Obwohl ROS in verschiedensten Bereichen verwendet wird, gibt es eine Reihe von Problemen. Auf einige davon wird im Folgenden kurz eingegangen.

2.1 Komplizierte Projektverwaltung

ROS verwaltet Projekte in Workspaces. In einem Workspace können viele Projekte liegen. Projekte können in einem Workspace gebaut und ausgeführt werden.

Da ROS ein breites Spektrum von Anwendungsfällen abdeckt, können verschiedenste Konfigurationen verwendet werden um ein Projekt innerhalb eines solchen Workspaces zu beschreiben. Dies hat den Nachteil, dass für neue Projekte entweder manuell Konfigurationsdateien geschrieben oder lange Konfigurationsbefehle ausgeführt werden müssen.

Die ros2_ui soll diese Arbeiten durch eine grafische Oberfläche erleichtern, die es ermöglicht die Konfigurationsdateien automatisiert zu erstellen.

2.2 Fehlende Übersicht in komplexen Projekten

Es ist schwierig in komplexen Robotersystemen den Überblick zu behalten. Dabei spielt die Interaktion vieler verschiedener Komponenten eine große Rolle.

Ein Werkzeug, das es ermöglicht Beziehungen zwischen einzelnen Modulen sichtbar und intuitiv verständlich zu machen, ist daher sinnvoll. Damit ist es einfacher neue Komponenten einzupflegen und das bestehende System bei auftretenden Fehlern zu reparieren. Die Darstellung in einer grafischen Oberfläche vereinfacht diesen Prozess gegenüber einer textuellen Darstellung weiter.

2.3 Nichtexistenz ähnlicher Projekte für ROS2

Für den älteren Entwicklungszweig ROS1 gibt es ein Projekt, das eine Nutzerschnittstelle anbietet und damit dieser Bachelorarbeit ähnelt (Manschitz, 2020).

Dieses beschränkt sich jedoch auf die Darstellung von bestehenden Systemen und bietet keine Möglichkeiten für Veränderungen an den Komponenten oder Konfigurationen.

Da sich die Programmierschnittstelle von ROS1 zu ROS2 stark verändert hat, ist eine simple Anpassung dieses Projekts auf ROS2 leider nicht möglich. Daher ist es sinnvoll, die Implementierung der Nutzerschnittstelle von Grund auf neu zu beginnen. Die neue Implementierung sollte die Möglichkeit bieten, auch Komponenten und Konfigurationen zu ändern.

3 Ziel

Die Problemstellungen in Kapitel 2 zeigen den Bedarf an einem User Interface für ROS2. Dies sollte den Umgang mit ROS2 vereinfachen und verschiedene Funktionen ermöglichen. Dabei ist eine Ausführung als Kommandozeilentool oder grafische Oberfläche denkbar.

3.1 Software, die den Umgang mit ROS2 vereinfacht

Ziel der Arbeit ist ein vereinfachter Umgang mit ROS2. Hierfür soll eine Software geschrieben werden, die es ermöglicht, grundlegende Aktivitäten ohne großen Aufwand durchzuführen.

In einem solchen Programm soll Folgendes möglich sein:

- Neue Projekte anlegen
- Akteure aus einer Liste verfügbarer Komponenten auswählen und zum Projekt hinzufügen
- Akteure individuell konfigurieren und löschen
- Projekte starten und beenden
- Parameter für die Kommunikation über die ROS2-Middleware festlegen

3.2 Grafische Oberfläche

Wie bereits beschrieben vereinfacht eine grafische Darstellung den Umgang mit den häufig komplexen Roboterprogrammen. Daher soll für die hier geschriebene Software eine grafische Oberfläche erstellt werden, die die Ziele aus Abschnitt 3.1 erfüllt.

4 Lösungskonzept

Im Rahmen dieser Bachelorarbeit wurde die Software ros2_ui geschrieben, die als Nutzerschnittstelle den Umgang mit ROS vereinfacht. Dieses Kapitel gibt einen Überblick über die Komponenten des entstandenen Programms, deren Funktionen sowie verwendete Technologien und Sprachen. Auch wird darauf eingegangen, welche Möglichkeiten zur Installation bereitgestellt werden.

4.1 Funktionsbeschreibung der Komponenten

ros2_ui besteht aus einer Sammlung von Programmen. Diese könnte über eine GUI oder als interaktives Konsolenprogramm verwendet werden. Im Rahmen dieser Arbeit wurde nur die GUI implementiert. Die einzelnen Komponenten werden im Folgenden näher beschrieben.

4.1.1 Programmsammlung

Die Programmsammlung besteht aus vielen Klassen und kleinen Skripten, die im Zusammenspiel miteinander die an sie gestellten Aufgaben verrichten. Dazu gehören beispielsweise

- Starten des Backends der GUI
- Laden und Speichern von Projekten
- Kommunikation mit ROS2-Dienstprogrammen
- Schreiben von Konfigurationen

Das Programm wird aus der Kommandozeile gestartet.

4.1.2 GUI

Mit der GUI kann der Nutzer mit der Programmsammlung interagieren ohne textuelle Befehle ausführen zu müssen. Einen Einblick in die Funktionen der Oberfläche liefert das Kapitel 5.

4.2 Installation

Das Programm-Paket kann auf verschiedenen Wegen installiert werden.

1. **Als versioniertes Python-Paket:**
   Bereitgestellt wird ein betriebssystemunhabhängiges Installationspaket. Dieses unterliegt einer Versionskontrolle womit sichergestellt werden soll, dass ein lauffähiges Programm vorliegt.

2. **Als entwicklungsaktuelle Version:**
   Angeboten wird ein betriebssystemunhabhängiges Installationsskript. Dieses stellt den tagesaktuellen Entwicklungsstand dar.

Bei beiden werden relevante Python-Pakete als Abhängigkeiten gelistet und automatisch installiert. Drittpakete müssen vorab separat installiert werden; dafür gibt es ein automatisiertes Skript.

4.3 Technologien und Sprachen

Im Rahmen dieses Projekts werden verschiedene Technolgien und Sprachen verwendet um ros2_ui zu programmieren. Neben den Programmiersprachen Python 3, Shellskript, HTML, CSS und JavaScript werden auch Flask, Websocket und JSON verwendet.

4.3.1 Python 3

Da ROS2 Programmierschnittstellen für Python und C++ anbietet, kommen diese Sprachen für die Programmierung der Software in Frage. Python ist, verglichen mit C++, deutlich einsteigerfreundlicher und breiter akzeptiert in der Open-Source-Community. Da es sich bei Python um eine Skriptsprache handelt müssen Programme nicht für jede Systemarchitektur einzeln kompiliert werden, sondern werden vom passenden Python-Interpreter ausgeführt. Zusammenfassend finden sich viele Gründe für die Verwendung von Python, weswegen dieses auch für die Programmierung der ros2_ui genutzt wird.

4.3.2 Shellskript

Für einige Interaktionen mit Betriebssystem oder Drittprogrammen existiert keine direkten Programmierschnittstellen. Diese Befehle werden als Shell-Programme ausgeführt und die Eingaben und Ausgaben entsprechend von der ros2_ui bereitgestellt bzw. verarbeitet.

4.3.3 HTML, CSS, JavaScript

Die Oberfläche wird in Form einer Webseite bereitgestellt, die über den Browser angezeigt wird. Aktionen des Nutzers werden per Netzwerk an das Backend übermittel übermittelt und dort die passenden Skripte aus der Programmsammlung ausgeführt, anders herum meldet das Backend der Oberfläche Statusänderungen zurück. Für die Beschreibung der GUI werden HTML und CSS verwendet. Um Inhalte nachzuladen und Befehle an das Backend abzusetzen wird JavaScript genutzt.

4.3.4 WebSocket

Um bidirektionale Kommunikation zwischen der GUI im Browser des Nutzers und dem Backend zu ermöglichen werden WebSockets eingesetzt. Sie erlauben es dem Backend selbstständig Nachrichten (beispielsweise mit Updates zum aktuell laufenden Bau-Prozess eines Projektes) auszuliefern, ohne dass der Client dazu eine Anfrage stellen muss. WebSockets sind als Erweiterung für Flask verfügbar.

4.3.5 Webserver (Flask)

Flask (Pallets Projects, 2021) ist ein Webserver-Framework. Weil Flask selbst in Python geschrieben ist entsteht hier nicht die Notwendigkeit für eine weitere Programmiersprache. Die Nutzung eines Webserver-Frameworks in Python hat den Vorteil, dass die Programmsammlung der ros2_ui direkt genutzt werden kann. Flask stellt die Oberfläche für die Anzeige im Browser des Benutzers und die WebSockets bereit.

4.3.6 JSON

JSON wird als Datenformat für Konfigurationsdateien und die Datenübertragung zwischen Webserver und Benutzeroberfläche genutzt. JSON ist ein weit verbreitetes Datenformat für ebensolche Anwendungen.

4.4 Code-Struktur: Clean Architecture

Je größer das Projekt, desto wichtiger ist es sich an bekannte Strukturen zu halten. So kann gewährleistet werden, dass auch in einem Projekt mit mehreren Programmierern der Code übersichtlich und verständlich bleibt. Diese Übersichtlichkeit wird unterstützt durch die klare Trennung zwischen einzelnen Schichten. Eine geeignete Struktur ist die „Clean Architecture“ (Martin, 2018), die für dieses Projekt verwendet wird.

Diese Strukturmethode dient nicht ausschließlich der Klarheit des Codes sondern stellt sicher, dass sich Modifikationen und Anpassungen schnell und einfach umsetzen lassen. Auch wird die Fehleranzahl reduziert und Probleme lassen sich gut finden und beheben. So beschreibt Martin (2018) dass es bei der „Clean Architecture“ zu minimalem Aufwand bei maximaler Funktionalität und Flexibilität kommt.

5 Implementierung

In diesem Kapitel wird die im Rahmen dieser Arbeit implementierte Software vorgestellt.

5.1 Projektverwaltung

ROS2 nennt die vom Nutzer zusammengefassten Sammlungen von Programm-Dateien *packages*. ros2_ui hingegen verwaltet *projects*. Jedes *project* resultiert nach dem Bauen in einem von ROS2 ausführbaren *package*. Die Umbenennung sorgt für eine klarere Trennung zwischen den Kontexten von ROS2 und ros2_ui. Um den deutschen Sprachfluss der Arbeit nicht zu stören wird ab jetzt von *Projekten* (=*projects*) bzw. *Paketen* (=*packages*) gesprochen.

Im Folgenden werden die einzelnen Interaktionsmöglichkeiten für *Pakete* bzw. *Projekte* erläutert. Dabei wird jeweils darauf eingegangen, wie die Aufgaben in ROS2 zu erledigen sind und wie die ros2_ui diese Aufgaben für den Nutzer darstellt. Die Befehle für ROS2 sind dabei stark gekürzt, hier wird stattdessen auf die ausführliche Dokumentation (OSRF, 2020b) verwiesen.

Abbildung 5.1 zeigt ein Beispielprojekt in der grafischen Oberfläche von ros2_ui.

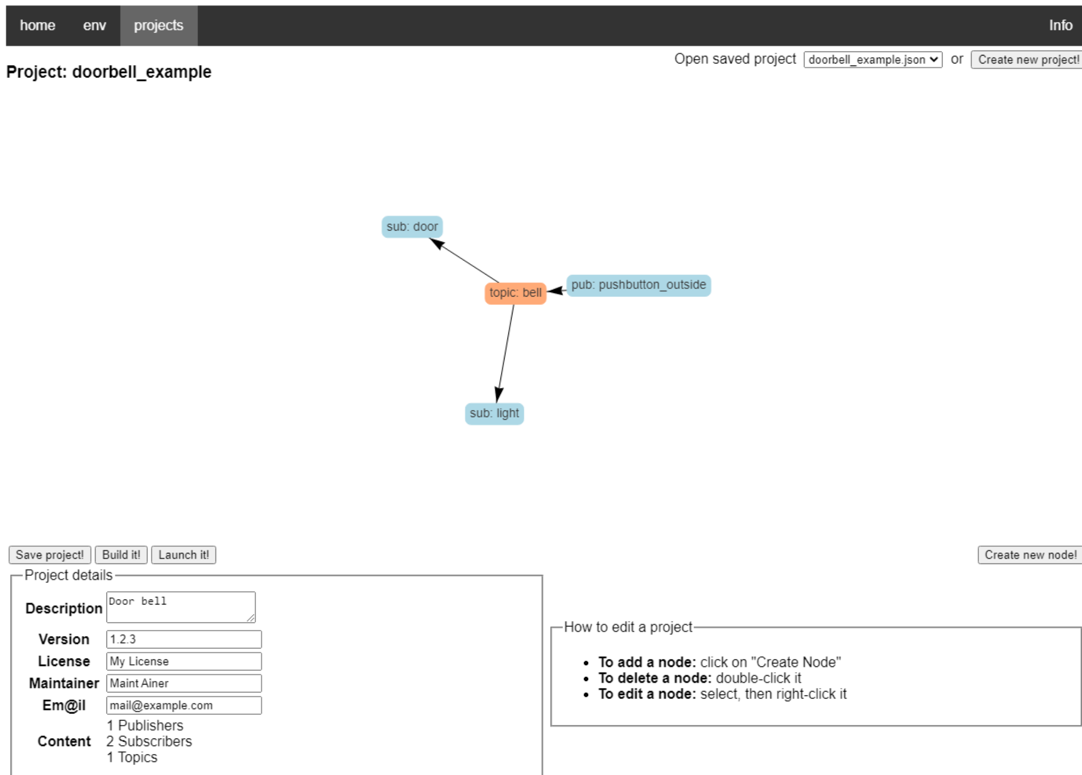


Abbildung 5.1: Bildschirmfoto - *Projekt*-Seite

5.1.1 ROS2: Arbeitsverzeichnis

Um ein Arbeitsverzeichnis für ROS2 anzulegen muss zunächst eine Orderstruktur angelegt werden. Es braucht einen Ordner für das Arbeitsverzeichnis und darin einen Ordner namens „src“ für die *Pakete*. Anschließend wird das Arbeitsverzeichnis mit dem Befehl „rosdep update“ initialisiert. Die Ordnerstruktur eines ROS2-Arbeitsverzeichnisses stellt dich wie folgt dar:

```
ros2_ws
+---build
¦
+---install
¦
+---log
¦
+---src
    +---doorbell_example
        ¦   package.xml
        ¦   setup.cfg
        ¦   setup.py
        ¦
        +---doorbell_example
        ¦       door_sub.py
        ¦       door_sub_runner.py
        ¦       light_sub.py
        ¦       light_sub_runner.py
        ¦       pushbutton_outside_pub.py
        ¦       pushbutton_outside_runner_button.py
        ¦       __init__.py
        ¦
        +---launch
        ¦       doorbell_example.launch.py
        ¦
        +---resource
        ¦       doorbell_example
        ¦
        +---test
```

Die Ordner *build*, *install* und *log* werden beim Bau von Paketen automatisch erzeugt und müssen nicht manuell angelegt werden.

5.1.2 ros2_ui: Arbeitsverzeichnis

Die Struktur für die Projektdateien der ros2_ui wird automatisch im home-Verzeichnis des Benutzers erzeugt. Im Arbeitsverzeichnis von ros2_ui liegt auch ein ROS2-Arbeitsverzeichnis, das wie oben beschrieben aufgebaut wird. Die Ordnerstruktur eines ros2_ui-Arbeitsverzeichnisses stellt dich wie folgt dar:

```
.ros2_ui
¦   LOG.log
¦
+---projects
¦       doorbell_example.json
¦
+---ros2_ws
```

Beim Ordner *ros2_ ws* handelt es sich um das ROS2-Arbeitsverzeichnis, dass die ros2_ui intern nutzt. Es ist aufgebaut, wie oben in Unterabschnitt 5.1.1 beschrieben.

5.1.3 ROS2: *Paket* anlegen

Zum Anlegen eines neuen *Pakets* im aktuellen Arbeitsverzeichnis wird entweder ein bestehendes *Paket* in den „src“ Ordner im Arbeitsverzeichnis kopiert oder ein neues leeres Projekt mithilfe des Befehls „ros2 pkg create“ erstellt. Dabei können viele Parameter in der Kommandozeile mit übergeben werden, um sie später nicht manuell anpassen zu müssen.

5.1.4 ros2_ui: *Projekt* anlegen

Das Anlegen eines neuen *Projektes* in der ros2_ui funktioniert auf Knopfdruck. Es muss lediglich ein Name vergeben werden, der von ROS2 vorgegebenen Kriterien genügt.

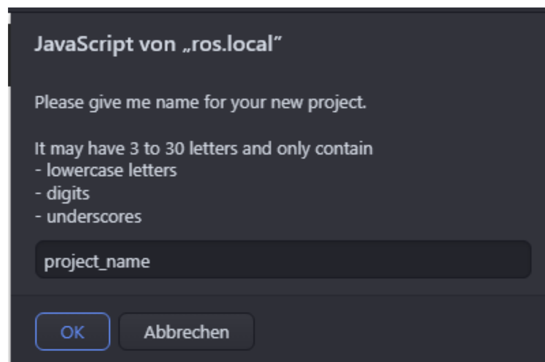


Abbildung 5.2: Bildschirmfoto - Dialog zur Erstellung eines *Projekts*

5.1.5 ROS2: *Paket*daten bearbeiten

Zum nachträglichen Bearbeiten von *Paket*daten müssen oft mehrere Konfigurationdateien (package.xml und setup.py) im „src“-Ordner des jeweiligen *Pakets* mithilfe eines Texteditors angepasst werden.

5.1.6 ros2_ui: *Projekt*daten bearbeiten

Um das Bearbeiten von *Projekt*daten zu erleichtern wird eine Eingabemaske angeboten. Nach Änderungen muss das Projekt gespeichert werden.

Save project! Build it! Launch it!
Project details
Description: Door bell
Version: 1.2.3
License: My License
Maintainer: Maint Ainer
Em@il: mail@example.com
Content: 1 Publishers, 2 Subscribers, 1 Topics

Abbildung 5.3: Bildschirmfoto - Maske zur Bearbeitung der Metadaten eines *Projekts*

5.1.7 ROS2: *Paket* laden und speichern

Das manuelle Laden und Speichern von *Paketen* entfällt in ROS2-Arbeitsverzeichnissen, weil direkt in Dateien gearbeitet wird, die nach der Bearbeitung ihrerseits gespeichert werden.

5.1.8 ros2_ui: *Projekt* laden und speichern

Um ein *Projekt* zu speichern muss ein Knopf betätigt werden; um ein anderes zu laden kann aus einem Dropdown-Menü das passende *Projekt* ausgewählt werden.

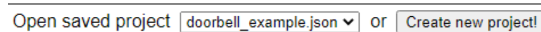


Abbildung 5.4: Bildschirmfoto - Dropdown für das Laden eines *Projekts*

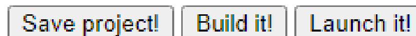


Abbildung 5.5: Bildschirmfoto - Knöpfe zum Speichern, Bauen und Ausführen

5.1.9 ROS2: *Paket*inhalt bearbeiten

Der Inhalt eines *Pakets* wird in ROS2 bearbeitet indem man die entsprechenden Programmdateien im *Paket* ändert.

5.1.10 ros2_ui: *Projekt*inhalt bearbeiten

In ros2_ui können neue Inhalte von *Projekten* interaktiv hinzugefügt, bearbeitet und gelöscht werden. Ein Klick auf „Create new Node“ öffnet beispielsweise eine Eingabemaske für die Erstellung eines neuen *Knotens* (siehe Abbildung 5.9 und Abbildung 5.10), ein Rechtsklick auf einen vorhandenen Knoten öffnet dessen Einstellungen und ein Doppelklick löscht ihn. Das *Projekt* muss anschließend per Klick auf „Save project!“ (siehe Abbildung 5.5) gespeichert werden.

5.1.11 ROS2: *Paket* bauen

Vor dem Bauen des *Pakets* müssen zunächst durch den Aufruf von „rosdep install“ alle Abhängigkeiten bereitgestellt werden. Das Bauen eines *Pakets* wird durch einen Aufruf von „colcon build“ ausgelöst.

5.1.12 ros2_ui: *Projekt* bauen

Das Bauen des Projektes in der ros2_ui erfolgt auf Knopfdruck (siehe Abbildung 5.5). Im Hintergrund erledigt das Backend dann folgende Aufgaben:

1. Falls es noch nicht vorhanden ist, wird das ROS2-Arbeitsverzeichnis angelegt.
2. Veraltete Daten vom aktuellen *Projekt* bzw. *Paket* werden bereinigt.
3. Es wird ein leeres ROS2-*Paket* erzeugt.
4. Die vom Nutzer eingegebenen Daten werden in Vorlage eingetragen und diese im ROS2-*Paket* abgelegt.
5. Die Konfigurationsdateien im *Paket* werden angepasst.
6. Für das Projekt wird ein Start-Skript angelegt.
7. Das *Paket* wird wie oben beschrieben gebaut.

Der Nutzer sieht in der Oberfläche im Browser dabei die Ausgaben der genutzten Dienstprogramme von ROS2 (siehe Abbildung 5.6).

5.2 Publish-Subscribe

Der *Publish-Subscribe-Mechanismus* ist eine zentrale Möglichkeit, wie verschiedene Komponenten in einem Robotersystem über ROS2 miteinander kommunizieren können. Im Folgenden wird beschrieben, wie der Mechanismus funktioniert und wie er in ros2_ui umgesetzt ist.

Build-Log

```
09:00:07 INFO - Existing package deleted.
09:00:08 INFO - STDOUT of 'ros2 pkg create...':
| going to create a new package
| package name: doorbell_example
| destination directory: /home/ubuntu/.ros2_ui/ros2_ws/src
| package format: 3
| version: 0.0.0
| description: Door bell
| maintainer: ['Maint Ainer ']
| licenses: ['My License']
| build type: ament_python
| dependencies: ['rclpy', 'std_msgs']
| creating folder /home/ubuntu/.ros2_ui/ros2_ws/src/doorbell_example
| creating /home/ubuntu/.ros2_ui/ros2_ws/src/doorbell_example/package.xml
| creating source folder
| creating folder /home/ubuntu/.ros2_ui/ros2_ws/src/doorbell_example/doorbell_example
| creating /home/ubuntu/.ros2_ui/ros2_ws/src/doorbell_example/setup.py
| creating /home/ubuntu/.ros2_ui/ros2_ws/src/doorbell_example/setup.cfg
| creating folder /home/ubuntu/.ros2_ui/ros2_ws/src/doorbell_example/resource
| creating /home/ubuntu/.ros2_ui/ros2_ws/src/doorbell_example/resource/doorbell_example
| creating /home/ubuntu/.ros2_ui/ros2_ws/src/doorbell_example/doorbell_example/__init__.py
| creating folder /home/ubuntu/.ros2_ui/ros2_ws/src/doorbell_example/test
| creating /home/ubuntu/.ros2_ui/ros2_ws/src/doorbell_example/test/test_copyright.py
| creating /home/ubuntu/.ros2_ui/ros2_ws/src/doorbell_example/test/test_flake8.py
| creating /home/ubuntu/.ros2_ui/ros2_ws/src/doorbell_example/test/test_pep257.py
|
09:00:08 INFO - ros2 pkg create returned code 0
09:00:08 INFO - callback is -led-
09:00:08 INFO - callback is -user-supplied-
09:00:08 INFO - registering entryPoint 'pushbutton_outside_runner_button =
doorbell_example.pushbutton_outside_runner_button:main'
09:00:08 INFO - registering entryPoint 'light_sub = doorbell_example.light_sub:main'
09:00:08 INFO - registering entryPoint 'door_sub = doorbell_example.door_sub:main'
09:00:08 INFO - registering launch file
09:00:08 INFO - Creating project 'doorbell_example'... DONE!
09:00:08 INFO - Resolving and installing dependencies...
```

Close!

Abbildung 5.6: Bildschirmfoto - Bau-Log in der ros2_ui

5.2.1 ROS2: *Knoten*

Knoten sind die Akteure im ROS2-Netzwerk. Sie können beispielsweise:

- Eingaben annehmen
- Ausgaben tätigen
- Hardware ansteuern
- Daten auswerten
- Befehle versenden

Knoten werden vom Nutzer erstellt und führen von ihm vorgegebenen Code aus. Sie kommunizieren über das ROS2-Netzwerk miteinander und tauschen Daten und Befehle aus.

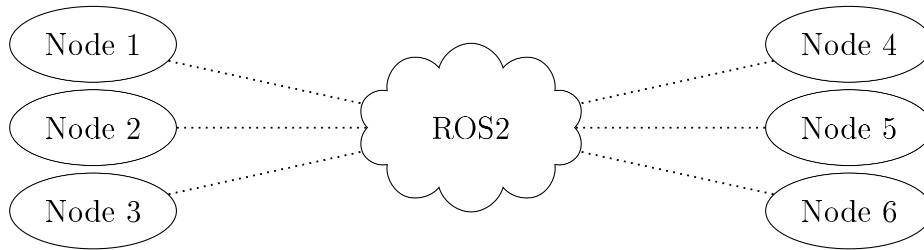
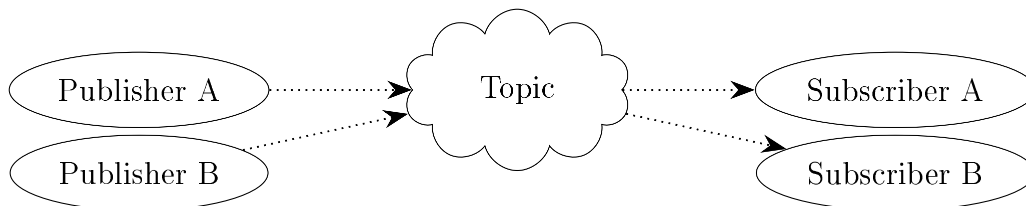


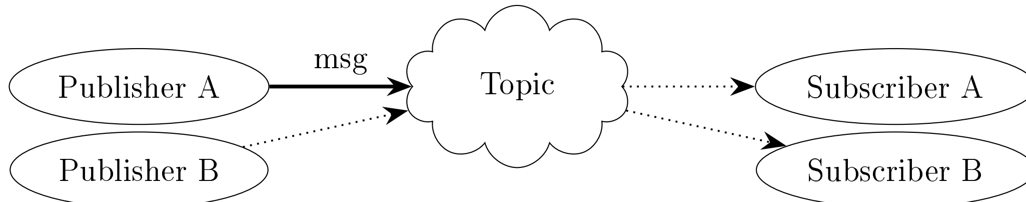
Abbildung 5.7: Node-Modell

5.2.2 ROS2: Publish-Subscribe-Mechanismus

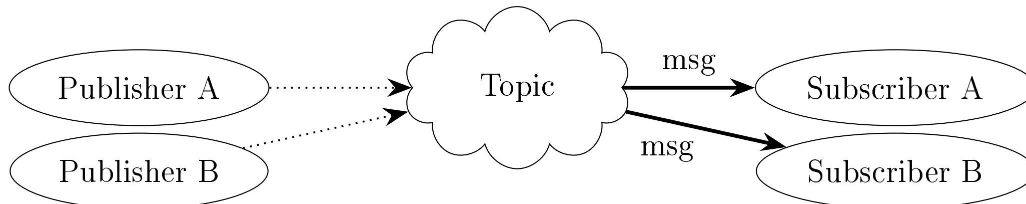
Der *Publish-Subscribe-Mechanismus* ist eine Möglichkeit im ROS2-Framework zu kommunizieren. Dafür registrieren sich ROS2-*Knoten* entweder als *Publisher* oder als *Subscriber* beim ROS-Framework. Dann können *Publisher* Nachrichten zu einem *Topic* veröffentlichen, die dann durch ROS2 an alle *Subscriber* verteilt werden, die dieses *Topic* abonniert haben. Der Ablauf ist in Abbildung 5.8 abgebildet.



Vorher: Einige Publisher und Subscriber haben sich beim ROS2-Framework für das Thema *Topic* registriert.



Schritt 1: *Publisher A* veröffentlicht die Nachricht *msg* zum Thema *Topic*.



Schritt 2: Das ROS2-Framework verteilt die Nachricht *msg* an die Subscriber.

Abbildung 5.8: Publish-Subscribe-Mechanismus

5.2.3 ros2_ui: *Knoten*

Publisher und *Subscriber* sind *Knoten*. Zur Erstellung eines *Knotens* müssen folgende Daten vorliegen:

- Knoten-Name (*node_name*): Wie heißt der *Knoten*? Der Name muss innerhalb eines Pakets eindeutig sein.
- Thema (*topic*): Zu welchem Thema sollen Nachrichten abonniert werden?
- Datentyp (*msg_type*): Welchen Datentyp haben die Nachrichten? Eine Liste der unterstützten Nachrichtentypen findet sich im Anhang im Abschnitt C.
- Textausgabe (*uses_stdout*): Gibt der *Knoten* Text aus?
- Texteingabe (*needs_tty*): Braucht der *Knoten* Texteingaben?

5.2.4 ros2_ui: *Publisher*

Zur Erstellung eines *Publishers* müssen zusätzlich zu den bei 5.2.3 aufgelisteten Daten, folgende weitere Informationen vorliegen:

- Quelle (*src*): Woher kommen die Nachrichten, die versendet werden? Es gibt Beispiele für den RaspberryPi (siehe Abschnitt 5.5), sodass beispielsweise bei Knopfdruck eine Nachricht veröffentlicht wird.

Wie die Eingabemaske für die Erstellung eines *Publishers* aussieht ist in Abbildung 5.9 dargestellt.

Abbildung 5.9: Bildschirmfoto - Dialog zur Erstellung eines *Publishers*

5.2.5 ros2_ui: *Subscriber*

Zur Erstellung eines *Subscribers* müssen zusätzlich zu den bei 5.2.3 aufgelisteten Daten, folgende weitere Informationen vorliegen:

- Callback (*callback*): Wohin sollen eintreffende Nachrichten weitergeleitet werden? Es gibt ein paar Beispiele für den RaspberryPi (siehe Abschnitt 5.5), andere müssen selbst implementiert werden. Wenn hier „-user-supplied-“ ausgewählt wird, dann kann zusätzlich Code im Feld *user_code* angegeben werden.
- Nutzer-Code (*user_code*): Was soll mit eintreffenden Nachrichten passieren? Hier kann frei Python Code angegeben werden.

Wie die Eingabemaske für die Erstellung eines *Subscribers* aussieht ist in Abbildung 5.10 dargestellt.

Abbildung 5.10: Bildschirmfoto - Dialog zur Erstellung eines *Publishers*

5.2.6 ros2_ui: *Topic*

Topics werden in ros2_ui, wie auch in ROS2, implizit erstellt. Ein *Topic* wird angelegt, indem man einen *Subscriber* oder *Publisher* für ein *Topic* anlegt.

5.2.7 ros2_ui: ***Publisher* & *Subscriber***

Angelegte *Publisher* und *Subscriber* werden, wie in Abbildung 5.11 gezeigt, als gerichteter Graph in der ros2_ui angezeigt.

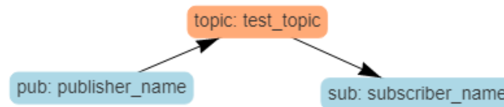


Abbildung 5.11: Bildschirmfoto - *Publisher* und *Subscriber* in ros2_ui

5.3 Start-Stop-Mechanismus

Die *Pakete* bzw. *Projekte* müssen gestartet werden um ihre Arbeit aufzunehmen. Im Folgenden ist beschrieben, wie der Start in ros2 bzw. ros2_ui jeweils abläuft.

5.3.1 ROS2: ***Paket*** ausführen

Um ein *Paket* auszuführen, müssen nach dem Bauen des *Pakets* zunächst die nötigen Umgebungsvariablen geladen werden. Anschließend können mit „ros2 run" einzelne *Knoten* gestartet werden.

Alternativ können auch mehrere *Knoten* auf einmal gestartet werden. Dafür muss vor dem Bau des Pakets eine „launch-Datei" erstellt werden. Anschließend kann das *Paket* - wie in Unterabschnitt 5.1.11 beschrieben - gebaut, die Umgebungsvariablen mittels „source" geladen und das *Paket* mit „ros2 launch" gestartet werden.

In der Ausgabe sieht der Nutzer dann die Ausgaben der gestarteten *Knoten* und Statusmeldungen des ROS2-Frameworks.

5.3.2 ros2_ui: Projekt ausführen

Zur Ausführung eines *Projektes* in der ros2_ui genügt ein Klick auf den „Launch it!"-Knopf. Nach der Auswahl der zu startenden *Knoten* werden diese - wie in Unterabschnitt 5.3.1 beschrieben - vom Backend gestartet. Der Nutzer sieht dann die Ausgaben, die das ROS2-Framework bereitstellt, als Log (siehe Abbildung 5.12). Ausgaben der *Knoten* werden im Log mit dargestellt.

Close

Launch Project

```
launch-request sent...
waiting...
[INFO] [launch]: All log files can be found below /home/ubuntu/.ros/log/2022-02-23-09-32-04-429371-ubuntu-18237
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [door_sub-3]: process started with pid [18243]
[INFO] [pushbutton_outside_runner_button-1]: process started with pid [18239]
[INFO] [light_sub-2]: process started with pid [18241]
[door_sub-3]
[door_sub-3]
[door_sub-3]
[door_sub-3]
[door_sub-3]
[door_sub-3]
[door_sub-3]
[door_sub-3]
[door_sub-3]  _______________
[door_sub-3] |  ___________  |
[door_sub-3] | |    _ _ _    | |
[door_sub-3] | |   |     |   | |
[door_sub-3] | |   |     |   | |
[door_sub-3] | |   |_ _ _|   | |
[door_sub-3] | |             | |
[door_sub-3] | |          () | |
[door_sub-3] | |          || | |
[door_sub-3] | |          () | |
[door_sub-3] | |             | |
[door_sub-3] | |             | |
[door_sub-3] | |             | |
[door_sub-3] |_|_____________|_|
```

Stop | Stop & Close

Abbildung 5.12: Bildschirmfoto - Ausführungslog in der ros2_ui

5.4 Eigene *Knoten*

Wird bei einem *Subscriber-Knoten* als Ausgabe-Weg („callback“) die Option „-user-supplied-“ gewählt, dann hat der Nutzer die Möglichkeit eigenen Code einzufügen. Der eigene Code kann eine Funktion mit der Signatur „init()“ definieren, die dann beim Start des *Knotens* aufgerufen wird. Der eigene Code muss eine Funktion mit der Signatur „callback(msg)“ anbieten, die bei eingehenden Nachrichten aufgerufen wird. Bei „msg“ handelt es sich um eine Instanz des unter „msg-type“ ausgewählten Datentyps. Die Eingabemaske ist in Abbildung 5.10 zu sehen.

5.5 Vorgefertigte *Knoten*

Für den Raspberry Pi gibt es einige vorgefertigte *Knoten*, die in der ros2_ui zur Auswahl stehen.

5.5.1 *Subscriber*: „-led-“

Beim Aufruf von „init()“ werden die Objekte angelegt, die die Ansteuerung der LED ermöglichen und die LED wird zunächst ausgeschaltet. Wird eine Nachricht vom Typ std_msgs.msg.String empfangen und lautet der Text darin „pressed“, dann wird die LED für einige Sekunden eingeschaltet.

5.5.2 *Subscriber*: „-stdout-“

Wird eine Nachricht vom Typ std_msgs.msg.String empfangen, dann wird sie auf „stdout“ ausgegeben.

5.5.3 *Publisher*: „-button-“

Beim Aufruf der „main()“-Funktion werden die Objekte angelegt, die die Überwachung des Druckknopfes ermöglichen und der aktuelle Status des Knopfes wird einmal versandt. Anschließend wird auf weitere Ereignisse gewartet. Wird der Knopf gedrückt, so wird die Nachricht „pressed“ versandt; beim Loslassen lautet sie „released“. Die Nachrichten haben den Typ std_msgs.msg.String.

6 Demonstration: Haustür

Mit einem Beispiel wird die Bedienung und Funktionalität des ros2_ui demonstriert.

6.1 Geschichte

Alice möchte Bob besuchen. Sie klingelt an seiner Tür, die sich nach einigen Sekunden öffnet. Nun hat Alice ein paar Sekunden Zeit um durch die Tür zu treten, bevor die Tür sich wieder schließt. Anschließend kann erneut geklingelt werden.

6.2 Akteure

Für die Demonstration werden verschiedene Hard- und Softwarekomponenten verwendet. Die Hardwarekomponenten werden in Abbildung 6.1 gezeigt. Die ros2_ui und alle beteiligten ROS2-Knoten laufen auf einem RaspberryPi, an den eine Steckplatine angeschlossen ist. Technische Spezifikationen des hier verwendeten Systems finden sich im Anhang in Abschnitt A. Im Folgenden werden die einzelnen Akteure erläutert.

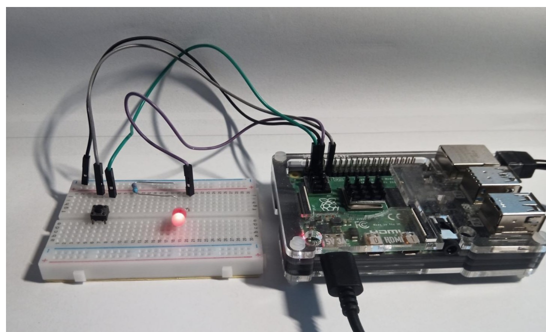


Abbildung 6.1: Bild von RaspberryPi und Steckplatine

6.2.1 Klingelknopf

Der „Klingelknopf“ wird durch einen Druckknopf repräsentiert, der auf der Steckplatine montiert ist. Der Knopf ist in Abbildung 6.1 links zu sehen.

6.2.2 Klingel

Das Klingeln der „Klingel“ wird auf zwei Arten dargestellt. Auf der Steckplatine ist eine LED angebracht, die aufleuchtet um das Klingeln anzuzeigen. Zu sehen ist die LED in Abbildung 6.1. Darüber hinaus wird das Klingeln auch über den Akteur „Tür“ dargestellt; dort erscheint dann das Wort „RING“ über der „Tür“.

6.2.3 Alice

Alice wird durch den Referenten gespielt; er drückt den „Klingelknopf“.

6.2.4 Tür

Nach einigen Sekunden öffnet sich die „Tür“. Sie ist als ASCII-Art (Wikipedia, 2021) angelegt. Die verschiedenen Zustände der „Tür“ sind in Tabelle 6.1 dargestellt.

geschlossen	geschlossen mit Klingeln	geöffnet

geschlossen:

```
 ---------------
|   ___________  |
| |    _ _ _   | |
| |   |     |  | |
| |   |     |  | |
| |   |_ _ _|  | |
| |            | |
| |         () | |
| |         || | |
| |         () | |
| |            | |
| |            | |
| |            | |
|_|____________|_|
```

geschlossen mit Klingeln:

```
                    /~~~~~~~~~\
                  < RING RING >
                    \~~~~~~~~~/


 ---------------
|   ___________  |
| |    _ _ _   | |
| |   |     |  | |
| |   |     |  | |
| |   |_ _ _|  | |
| |            | |
| |         () | |
| |         || | |
| |         () | |
| |            | |
| |            | |
| |            | |
|_|____________|_|
```

geöffnet:

```
 ---------------
|   ___________  |
| |   ,' | |   | |
| | |    | |   | |
| | |    | |   | |
| | |  ,'  |   | |
| | |,' o  |   | |
| |      H |   | |
| |      o |   | |
| |        |   | |
| |        |   | |
| |        ,'  | |
| |    ,'      | |
|_|,'__________|_|
```

Tabelle 6.1: Zustände der „Tür“ als ASCII-Art

6.3 *Projekt*-Verwaltung

Zunächst wird die Oberfläche von ros2_ui im Browser aufgerufen und in den „projects“-Tab gewechselt. Das Anlegen des *Projektes* und der *Knoten* ist in den folgenden Unterkapiteln beschrieben.

6.3.1 *Projekt* anlegen

Dort angelangt wird, wie in Abbildung 6.2 gezeigt, ein neues *Projekt* mit dem Namen „demo_door“ angelegt. Da leere Projekt wird geöffnet und steht zum Bearbeiten bereit. Nun können die Details des *Projektes* angepasst werden. Dieser Vorgang ist in Abbildung 6.3 dargestellt. Nun werden beginnend mit dem *Publisher-Knoten* für den „Klingelknopf“ alle Akteure hinzugefügt.

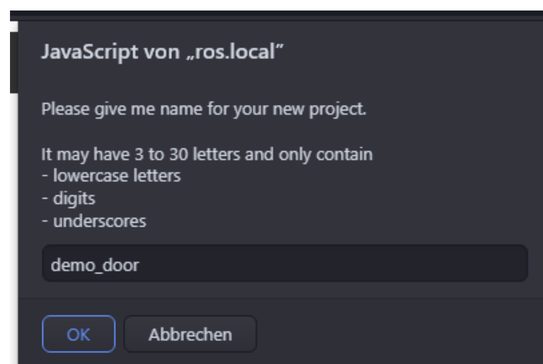


Abbildung 6.2: *Publisher* für den „Klingelknopf“ hinzufügen

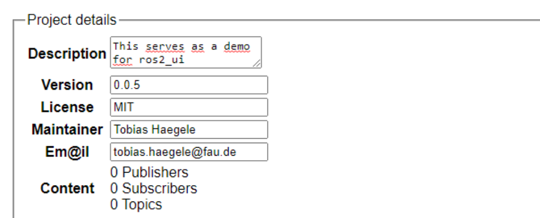


Abbildung 6.3: *Publisher* für den „Klingelknopf“ hinzufügen

6.3.2 „Klingelknopf“ hinzufügen

Wie in Abbildung 6.4 gezeigt wird der *Publisher-Knoten* für den „Klingelknopf“ erstellt und mit einem Klick auf „Save node“ gespeichert.

6.3.3 „Klingel-LED“ hinzufügen

Wie in Abbildung 6.5 gezeigt wird der *Subscriber-Knoten* für die „Klingel-LED“ erstellt und mit einem Klick auf „Save node“ gespeichert.

Abbildung 6.4: *Publisher* für den „Klingelknopf“ hinzufügen

6.3.4 „Tür“ hinzufügen

Wie in Abbildung 6.6 gezeigt wird der *Subscriber-Knoten* für die „Tür“ erstellt und mit einem Klick auf „Save node“ gespeichert.

Der vollständigen Programmcode des Python-Codes im Feld „user supplied code“ findet sich im Anhang in Abschnitt D.

6.3.5 *Projekt* speichern

Um die geänderten Projektdaten und die neu erstellten Knoten an ros2_ui zu übertragen muss das Projekt nun gespeichert werden. Der aktuelle Zustand des Projektes ist in Abbildung 6.7 ersichtlich.

6.3.6 *Projekt* bauen

Mit einem Klick auf „Build it!“ wird das Projekt gebaut. Der laufende Log weist dabei auf auftretende Fehler hin.

6.3.7 *Projekt* ausführen

Nachdem das Projekt erfolgreich gebaut wurde kann es gestartet werden. Dazu wird auf „Launch it!"geklickt. Auch hier läuft nun ein Log im Browser mit. Hier sind die Ausgaben der oben erstellten *Knoten* ersichtlich.

Close

Create / Edit node

All form fields are required.

Type of node: Subscriber

node_name	bell_light
topic	bell
msg_type	string
uses_stdout	false
needs_tty	false
callback	-led-
user supplied code	

```
# this code will only be used if you specify -user-supplied- as callback (see above)

def init():
    pass

# this example will just print the data to stdout
def callback(value):
    print(value.data)
```

Reset Save node

```
Available message sources:
-button- This will send std_msgs.msg.String messages whenever button (GPIO2) is pressed/released

Available message callbacks:
-stdout- This will just print the messages to stdout. Will be displayed in the browser.
-led- This will light up an LED (GPIO3) upon receiving the message "yes".
-user-supplied- Will enable an additional field. Supply your own code there. Lines must have length ≤ 120 chars.
```

Abbildung 6.5: *Subscriber* für die „Klingel-LED“ hinzufügen

6.4 Programmablauf

Nach dem Start initialisieren sich alle *Knoten* zunächst selbst. Hier werden beispielsweise alle referenzierten Module geladen und die Verbindung zum ROS2-Framework hergestellt. Die *Subsriber* abonnieren das *Thema* „bell“ und der *Publisher* registriert seine Absicht, Nachrichten für das Thema zu veröffentlichen. Diese Phase wird in Abbildung 6.8 dargestellt.

In Abbildung 6.9 sind dann alle *Knoten* mit ROS2 verbunden und warten auf Ereignisse.

Drückt der Nutzer nun den „Klingelknopf“, so sendet der *Publisher*, der dessen Status überwacht, die Nachricht „pressed“ ab. Dies ist in Abbildung 6.10 dargestellt.

Daraufhin leitet ROS2 die *Nachricht* an die *Subscriber* weiter. Dieser Schritt ist in Abbildung 6.11 zu sehen.

Im Anschluss führen die *Subscriber* ihren für die Nachricht hinterlegten Code aus. „door“ zeigt die ASCII-Art-Sequenz einer Tür an der es zunächst klingelt und die ein paar Sekunden später geöffnet wird. „light“ lässt die LED kurz aufleuchten. Anschließend warten alle *Knoten* wieder auf neue Ereignisse. Dieser Teil ist in Abbildung 6.12 dargestellt.

Close

Create / Edit node

All form fields are required.

Type of node: Subscriber

node_name	door
topic	bell
msg_type	string
uses_stdout	true
needs_tty	true
callback	-user-supplied-
user supplied code	

```
import sys
import threading
import random
import time
from std_msgs.msg import String

door_closed = """
```

Reset | Save node

```
Available message sources:
-button- This will send std_msgs.msg.String messages whenever button (GPIO2) is pressed/released

Available message callbacks:
-stdout- This will just print the messages to stdout. Will be displayed in the browser.
-led- This will light up an LED (GPIO3) upon receiving the message "yes".
-user-supplied- Will enable an additional field. Supply your own code there. Lines must have length ≤ 120 chars.
```

Abbildung 6.6: *Subscriber* für die „Tür“ hinzufügen

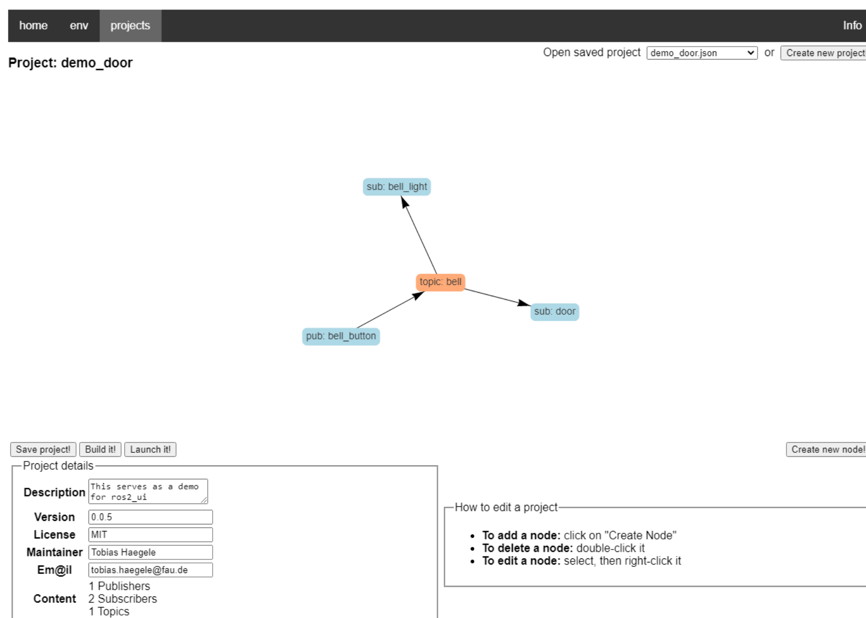


Abbildung 6.7: Fertiges *Projekt*

Abbildung 6.8: Vor der Initialisierung - Alle *Knoten* wurden gestartet.

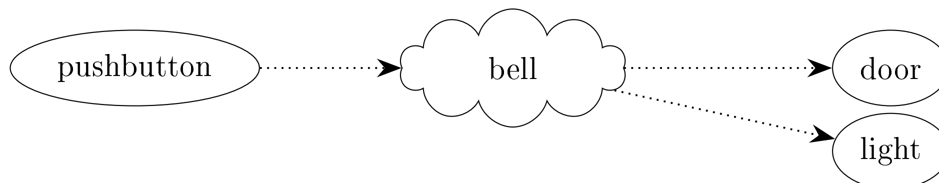


Abbildung 6.9: Nach der Initialisierung - Die *Knoten* haben sich beim ROS2-Framework für das Thema „bell“ registriert.

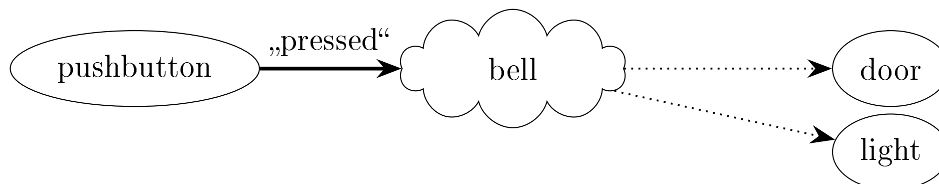


Abbildung 6.10: Nach Knopfdruck - Der *Publisher* „pushbutton“ versendet die Nachricht „pressed“.

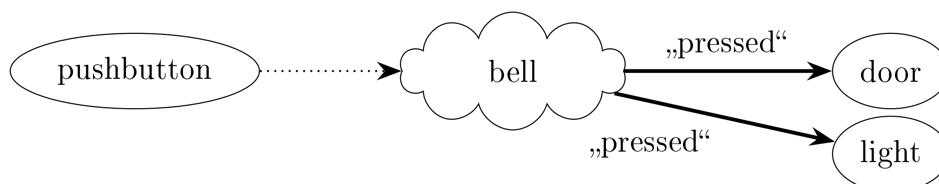


Abbildung 6.11: ROS2 verteilt die Nachricht „pressed“ an die Subscriber.

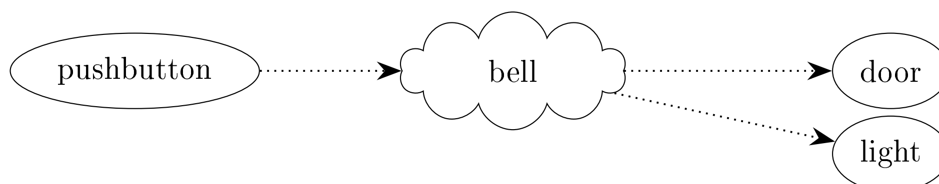


Abbildung 6.12: Nach dem Empfang - Die *Knoten* arbeiten ihre Programme ab und warten auf weitere Nachrichten.

7 Bewertung

Im Rahmen dieser Bachelorarbeit wurde ros2_ui konzipiert und erstellt. Die Programmsammlung und die für den Nutzer sichtbare GUI wurden in den vorausgegangenen Kapiteln vorgestellt. Nun soll bewertet werden, in wieweit das Programm eine Verbesserung der Benutzerfreundlichkeit darstellt.

Wie bereits in Kapitel 2 beschrieben wurde ist die Verwaltung von *Projekten* in ROS2 kompliziert. Konfigurationsdateien müssen manuell geschrieben und lange Befehle ausgeführt werden. Mithilfe der grafischen Oberfläche von ros2_ui können sie mit wenigen Mausklicks erzeugt werden. Ein Beispiel dafür ist das Anlegen von *Projekten*, der hier exemplarisch beschrieben wird. In ROS2 wird zum Anlegen des neuen *Pakets* folgender Befehl verwendet:

```
ros2 pkg create --build-type ament_python \
--destination-directory "PFAD/ZUM/ZIELORDNER" \
--description "PROJEKTBESCHREIBUNG" \
--license "LIZENZ" --dependencies "ABHÄNGIGKEITEN" \
--maintainer-email "EMAILADRESSE-DES-ERSTELLERS" \
--maintainer-name "NAME-DES-ERSTELLERS" PAKETNAME
```

Alternativ können im Befehl die übergebenen Parameter auch manuell in den Dateien „package.xml“ und „setup.py“ hinterlegt werden. Mithilfe der ros2_ui wird dies vereinfacht, denn die nötigen Werte werden über Eingabemasken abgefragt und der für die Erstellung des ROS2-*Pakets* nötige Befehl auf Knopfdruck abgesetzt.

Ein weiteres Problem von ROS2 ist die fehlende Übersicht über die einzelnen Komponenten in einem *Paket*. In einem ROS2-*Paket* erfolgt kann die Struktur eingesehen werden indem man die einzelnen Code-Dateien im *Paket* öffnet und ihren Inhalt betrachtet. Außerdem lässt sich in der Konfigurationsdatei „setup.py“ nachvollziehen welche der Code-Dateien von ROS2 ausgeführt werden könnten. Über die grafische Oberfläche von ros2_ui gelingt all das leichter, denn die *Knoten* und ihre Beziehungen werden als gerichteter Graph angezeigt. Welche *Knoten* gestartet werden kann beim Start entscheiden werden.

Bisher gibt es kein der ros2_ui ähnliches Projekt für ROS2. Daher stellt dieses Programm eine erhebliche Verbesserung der Benutzbarkeit dar, auch wenn aktuell nur eine begrenzte Auswahl möglicher Funktionalitäten umgesetzt wurde.

8 Schluss

Die Implementierung von ros2_ui ist ein Teilerfolg.

Die Programmsammlung und auch die Oberfläche von ros2_ui erfüllen alle im Abschnitt 3.1 definierten Ziele: ros2_ui ermöglicht die Ausführung grundlegender Aktivitäten. Dazu zählen das Anlegen, Entfernen, Starten und Beenden von Projekten. Auch ist es möglich Akteure hinzuzufügen, zu bearbeiten und zu löschen. Parameter für die Kommunikation können festgelegt werden.

Alle genannten Aufgaben können einfach mithilfe einer grafischen Oberfläche erledigt werden.

Was der Nutzer selbst an Konfiguration vornehmen muss beschränkt sich auf die Installation von ROS2 und der ros2_ui, sowie den Start der ros2_ui. Im Betrieb muss nichts rund um ROS2 manuell angepasst werden.

Damit ist eine erhebliche Verbesserung im Vergleich zur Nutzung von ROS2 ohne ros2_ui erreicht: Anstatt umfangreiche komplexe Konfigurationsdateien zu schreiben oder Konfigurationsbefehle auszuführen kann mithilfe der GUI die gewünschte Konfiguration über Masken angelegt werden.

Leider gibt es einige Limitierungen, die für eine nicht ausschließlich positive Bewertung sorgen. ROS2 ist ein sehr umfangreiches Framework mit einer Vielzahl von Anwendungsmöglichkeiten und Funktionen. Im Rahmen dieser Bachelorarbeit konnte nur ein kleiner Bereich davon abgebildet werden. Von den verschiedenen Varianten der Kommunikation wurde ausschließlich der *Publish-Subscribe-Mechanismus* abgedeckt. Eine Erweiterung auf andere Kommunikationsarten, wie beispielsweise Services, ist denkbar. Auch wurden nur einige der verfügbaren Funktionalitäten umgesetzt, darunter das Starten und Stoppen sowie das Bauen von Projekten.

Bei den Datentypen gibt es ebenfalls einige Einschränkungen. ROS2 in C++ verwendet andere Datentypen als ROS2 in Python 3, was durch die Fokussierung auf Python 3 in dieser Arbeit nicht berücksichtigt wird. Hinzu kommen zahlreiche weitere Datentypen wie beispielsweise Arrays, Koordinaten und Farben. Umgesetzt werden jedoch nur die im Anhang in Abschnitt C genannten Typen.

Darüber hinaus ist das aktuelle Design des *Publisher*-Knotens nicht echtzeitfähig. Stattdessen überprüft dieser nur in bestimmten Zeitintervallen ob neue Nachrichten für den Versand vorliegen. Das Design müsste an dieser Stelle entsprechend angepasst werden um die Echtzeitfähigkeit zu erreichen.

Außerdem werden aktuell keine über mehrere Maschinen verteilte Systeme innerhalb einer Oberfläche unterstützt. Eine Erweiterung um diese Funktionalität ist denkbar. Hierfür müsste ros2_ui auf allen beteiligten Geräten laufen und diese über ein Netzwerk miteinander verbunden sein. Eines der Systeme müsste dann die Koordinationsrolle übernehmen und die Eingaben des Nutzers an alle anderen weiterleiten. Der hierfür notwendige Programmieraufwand übersteigt jedoch den Rahmen dieser Bachelorarbeit.

Die Benutzeroberfläche an sich könnte moderner aussehen. Eine Möglichkeit dafür wäre die Nutzung eines modernen Web-Frameworks wie Angular oder VueJS, wobei sich die Rolle von Flask dann auf die Bereitstellung einer API reduzieren würde.

Der Autor der Arbeit plant, das Projekt ros2_ui in Zukunft zu pflegen und weiter auszubauen.

Appendices

A Entwicklungsumgebung

A.1 Hardware

Entwickelt und getestet wurde das für diese Arbeit entwickelte Programmpaket auf einem „Raspberry Pi 4 Model B Rev 1.2“ mit 4 GB Arbeitsspeicher. Als Betriebssystem wurde die Linux-Distribution „Ubuntu 20.04 - Focal Fossa“ genutzt.

A.2 Software

Verwendet wurde zuletzt Python in der Version 3.8.10. Als Kommandozeile wurde bash genutzt.

Diese Arbeit basiert auf ROS2 in der Version „Foxy Fitzroy - Patch Release 6.1“ für arm64-Systeme (Perron, 2021). Da Robot Operating System - Version 2 (ROS2) erfreulicherweise immer entwickelt wird, diese Arbeit aber eine einmalige Sache ist kann der Autor nicht dafür garantieren, dass getroffene Annahmen und Aussagen auch für die neueren Versionen gelten. Zum Zeitpunkt der Veröffentlichung dieser Arbeit ist bereits der Nachfolger „Galactic Geochelone“ veröffentlicht (Logan, 2021).

B Browserkompatibilität

ros2_ui nutzt für die Dialoge den relativ neuen HTML-Tag *dialog*. Zur Kompatibilität wird auf (Mozilla Developer Center, 2021) verwiesen. Firefox wird den Tag *dialog* erst ab Version 98 vollständig unterstützen. Im Browser muss JavaScript aktiviert sein.

C Nachrichtentypen

ROS2 stellt im Modul *std_msgs*[1] (OSRF, 2020a) verschiedene Datentypen bereit. ros2_ui bietet davon folgende Datentypen an:

- Bool
- Byte
- Char
- Empty
- Float
- Int
- String
- Time

[1] Code: https://github.com/ros2/common_interfaces/tree/master/std_msgs

D Code aus der Demonstration

Der „Tür“ *Subscriber* enthält folgenden Python3-Code.

```
import sys
import threading
import time
from std_msgs.msg import String

door_closed = """
 ---------------
|  ___________  |
| |   _ _ _   | |
| |  |     |  | |
| |  |     |  | |
| |  |_ _ _|  | |
| |           | |
| |        () | |
| |        || | |
| |        () | |
| |           | |
| |           | |
| |           | |
|_|___________|_|
"""

door_open = """
 ---------------
|  ___________  |
| |  ,' | |   | |
| | |   | |   | |
| | |   | |   | |
| | |  ,' |   | |
| | |,' o |   | |
| |     H |   | |
| |     o |   | |
| |       |   | |
| |       |   | |
| |      ,'   | |
| |   ,'      | |
|_|,'_________|_|
"""
```

```
door_ring = """
                 /~~~~~~~~~\\
                < RING RING >
                 \\~~~~~~~~~/


 ---------------
|  ___________  |
| |   _ _ _   | |
| |  |     |  | |
| |  |     |  | |
| |  |_ _ _|  | |
| |           | |
| |        () | |
| |        || | |
| |        () | |
| |           | |
| |           | |
| |           | |
|_|___________|_|
"""


def door_print(door: str):
    print("\n" * 50)
    print(door)


def init():
    door_print(door_closed)


t_open: threading.Thread


def callback(out: String):
    global t_open

    # if thread already ran, or is currently running: join it
    try:
        if t_open and t_open.is_alive():
            t_open.join()
    except NameError:
        pass
```

```
    # assemble new thread and run it
    t_open = threading.Thread(target=go_open_the_door, args=())
    if out.data == "yes":
        t_open.start()


def go_open_the_door():
    # get random time-to-open
    # sleep_time = random.randint(9, 15)
    sleep_time = 10

    # show ringing door
    door_print(door_ring)
    time.sleep(1)
    door_print(door_closed)
    time.sleep(1)
    door_print(door_ring)
    time.sleep(1)

    # show closed door (waiting for opening)
    door_print(door_closed)
    for i in range(sleep_time):
        time.sleep(1)
        print(".", end="")
        sys.stdout.flush()

    # show open door
    door_print(door_open)
    time.sleep(3)

    # show closed-again door
    door_print(door_closed)


if __name__ == '__main__':
    init()
    s = String()
    s.data = "yes"
    callback(s)
```

E Aufgabenstellung der Bachelorarbeit

In diesem Kapitel wird die Aufgabenstellung dieser Bachelorarbeit und Änderungen daran dokumentiert. Die folgende Seite zeigt die ursprüngliche Ausschreibung der Bachelorarbeit.

Im initialen Gespräch vom 07.04.2021 wurde festgelegt, dass die Arbeit die Struktur einer „Design science“-Arbeit haben wird. Darüber hinaus wurde festgelegt, dass für die Erstellung des zugehörigen Softwareprojektes freie Technologiewahl gilt und dass auch auf existierende Software aufgebaut werde darf.

Bachelor / Master Thesis Description [for optional], status: open [assigned | closed], language: [DE | EN]

Simple ROS Configurator

Summary

The goal of this thesis is to develop a (simple) ROS configuration tool to make configuring, launching, and shutting down of ROS-based systems easy. The main functions of the configurator are: Selection of components for a system, instance-specific configuration of components, modeling and configuration of component communication, creation of launch configuration, and startup and shutdown function. These functions shall be available through a graphical user interface, hiding the various configuration files from the user. The configurator should be able to handle a simple example application of at least three different components.

Work Results

The structure of the resulting thesis shall be as follows:

1. Literature review
2. (Final) requirements definition
    - The initial requirements are functions to:
        - Model the system's architecture by
            - Selecting components from a set of available ROS components
            - Placing those components of instances into a system model
            - Configuring each component instance independently
            - Defining communication through the ROS middleware
        - Generate configurations from the model
        - Be able to startup and shutdown the system
    - Definition of evaluation criteria
3. Architecture and design
4. Implementation
5. Evaluation against final requirements

Students should be aware that requirements may change over the course of the project, however, any such changes from initial to final requirements shall be documented in an appendix of the thesis.

Supervisor

Prof. Dr. Dirk Riehle, dirk.riehle@fau.de

Open Source Research Group
Computer Science Department
Friedrich-Alexander University

More information: https://oss.cs.fau.de/theses

F Veröffentlichung des Codes

Das für diese Arbeit erstellte Programmpaket wird vorerst unter der MIT License veröffentlicht. Wenn abschließend analysiert ist, inwieweit die verwendeten Frameworks und weiteren eingebundenen Softwareprojekte dies erlauben soll ein Wechsel zu einer Creative-Commons-Lizenz erfolgen.

Für das Programmpaket steht ein GitHub-Repository[2] bereit. Die zum Zeitpunkt der Abgabe dieser Bachelorarbeit aktuelle Version 0.1.0 steht dort[3] zum Herunterladen bereit.

Im oben genannten Repository auf GitHub steht auch eine kurze Anleitung zur Installation und Benutzung von ros2_ui zur Verfügung.

[2]Link: https://github.com/haegelix/ros2_ui

[3]Link: https://github.com/haegelix/ros2_ui/releases/tag/0.1.0

Literaturverzeichnis

OSRF. (2020a). *About ROS 2 interfaces*. https://docs.ros.org/en/foxy/Concepts/About-ROS-Interfaces.html#field-types

OSRF. (2020b). *ROS 2 Documentation: Foxy documentation*. Verfügbar 12. Februar 2022 unter https://docs.ros.org/en/foxy/

Logan, S. ((2021, 23. Mai). *ROS 2 Galactic Geochelone Released*. Verfügbar 8. Dezember 2021 unter https://discourse.ros.org/t/ros-2-galactic-geochelone-released/20559

Manschitz, S. (2020, 2. November). *ros_web_gui*. Verfügbar 2. Mai 2021 unter https://github.com/smanschi/ros_web_gui

Martin, R. C. (2018, 23. Februar). *Clean Architecture: Das Praxis-Handbuch für professionelles Softwaredesign - Regeln und Paradigmen für effiziente Softwarestrukturierung*. MITP. Verfügbar 20. Februar 2022 unter https://www.ebook.de/de/product/31865005/robert_c_martin_clean_architecture.html

Mozilla Developer Center. (2021, 3. Oktober). *⟨dialog⟩: The Dialog element*. Verfügbar 18. Februar 2022 unter https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dialog#browser_compatibility

Pallets Projects. (2021, 21. Mai). *Flask 2.0.1*. Verfügbar 25. Juni 2021 unter https://flask.palletsprojects.com/en/2.0.x/

Perron, J. (2021, 13. Oktober). *ROS 2 Foxy Fitzroy - Patch Release 6.1*. Verfügbar 10. November 2021 unter https://github.com/ros2/ros2/releases/tag/release-foxy-20211013

Wikipedia. (2021). *ASCII-Art — Wikipedia, die freie Enzyklopädie*. Verfügbar 18. Februar 2022 unter https://de.wikipedia.org/w/index.php?title=ASCII-Art&oldid=212022546