

# ROS2 Pakete

[Roboter] Kompakte autonome Roboter, die auf Fuß- und Gehwegen operieren, bieten für logistische Herausforderungen, Unterstützungsdienste für älterer Menschen oder Wartungs- und Monitoring Aufgaben, Lösungsansätze für gesamtgesellschaftliche Fragen. Dabei steht aktuell die Frage nach Konzepten und Ansätzen für die rechtliche Genehmigung der Roboter und Szenarien im Raum, die in jedem Fall eine Remote-Kontrolle für schwierige Situationen vorsehen wird. Eine entsprechende Absicherung der Kommunikation mit den Robotern ist also zwingend erforderlich.

Parameter	Kursinformationen
Veranstaltung:	Robotik Projekt
Semester	Wintersemester 2024/25
Hochschule:	Technische Universität Freiberg
Inhalte:	Umsetzung von ROS Paketen Kommunikationsprinzipien
Link auf GitHub:	<a href="https://github.com/TUBAF-IfI-LiaScript/VL_SoftwareprojektRobotik/blob/master/04_ROS_Pakete/04_ROS_Pakete.md">https://github.com/TUBAF-IfI-LiaScript/VL_SoftwareprojektRobotik/blob/master/04_ROS_Pakete/04_ROS_Pakete.md</a>
Autoren	Sebastian Zug & Georg Jäger & Claud.ai



---

Leitfragen zur heutigen Veranstaltung

- Wie ist ein ROS2-Paket aufgebaut und welche Dateien sind zwingend erforderlich?
- Wie erstellt und strukturiert man eigene ROS2-Pakete für verschiedene Anwendungsfälle?
- Wie funktioniert die Kommunikation zwischen mehreren ROS2-Nodes auf unterschiedlichen Rechnern?
- Welche Rolle spielt die `ROS_DOMAIN_ID` beim Betrieb verteilter ROS2-Systeme?
- Wie werden Nachrichten, Services und Aktionen in ROS2-Paketen definiert und genutzt?
- Wie kann man eigene Nachrichten- und Servicetypen in einem ROS2-Paket anlegen?
- Welche typischen Fehlerquellen gibt es beim Arbeiten mit ROS2-Paketen und verteilten Systemen?
- Wie testet und debuggt man ROS2-Pakete im Netzwerkverbund praktisch?

## ROS2 auf mehreren Rechnern

Während die Organisation von Anwendungen über mehreren Rechnern unter ROS1 aufwändiger war, ist die Realisierung in ROS2 außerordentlich einfach.

<https://roboticsbackend.com/ros2-multiple-machines-including-raspberry-pi/>

Die `ROS_DOMAIN_ID` trennt einzelne ROS2 Umgebungen voneinander.

```
ps -ax | grep ros-domain
```



Damit bleibt

```
ROS_DOMAIN_ID=1 ros2 topic pub -r 1 /string_topic std_msgs/String "{data: \n from my 2ND domain\}"
```

"unsichtbar".

## Konzept

ROS1 und 2 sind in Pakten organisiert, diese kann man als Container für zusammengehörigen Code betrachten.

Softwareengineering Ziele:

- Alle Funktionalität sollte so strukturiert werden, dass Sie in einem anderen Kontext erweitert und wiederverwendet werden kann.
- Jede Komponente (Memberfunktion, Klasse, Package) sollte eine einzige Verantwortlichkeit definieren.
- Die Kopplung zwischen den Paketen sollte allein über den Datenaustausch realisiert werden.
- Verzicht auf unnötige Funktionalität - *you don't pay for what you don't use*

Wenn Sie Ihren Code installieren oder mit anderen teilen möchten, muss dieser in einem Paket organisiert sein.

```
InspectPackages
```

```
> ros2 pkg
usage: ros2 pkg [-h] Call `ros2 pkg <command> -h` for more detailed usage.

Various package related sub-commands

optional arguments:
-h, --help            show this help message and exit

Commands:
create      Create a new ROS2 package
executables Output a list of package specific executables
list        Output a list of available packages
prefix      Output the prefix path of a package

Call `ros2 pkg <command> -h` for more detailed usage.
> ros2 pkg list
action_msgs
action_tutorials
actionlib_msgs
ament_cmake
ament_cmake_auto
ament_cmake_copyright
```

Ein Paket umfasst aber nicht nur den eigentlichen Code sondern auch:

- eine Spezifikation der Abhängigkeiten
- die Konfiguration des Build-Systems
- die Definition der nutzerspezifischen Messages
- die `launch` Files für den Start der Anwendungen und deren Parametrisierung

Ein minimales Paket umfasst zwei Dateien:

- `package.xml` - Informationen über das Paket selbst (Autor, Version, ...)
- `CMakeLists.txt` file - beschreibt wie das Paket gebaut wird und welche Abhängigkeiten bestehen

Die Paketerstellung in ROS 2 verwendet `ament` als Build-System und `colcon` als Build-Tool.

Pakete können in gemeinsamen `Workspaces` angelegt werden.

```
workspace_folder/
  src/
    package_1/
      CMakeLists.txt
```

```
    package.xml  
  
    package_2/  
        setup.py  
        package.xml  
        resource/my_package  
    ...  
    package_n/  
        CMakeLists.txt  
        package.xml
```

Diese Struktur wird durch das jeweilige Build-System automatisch erweitert. `colcon` erstellt standardmäßig weitere Verzeichnisse in der Struktur des Projektes:

- Das `build`-Verzeichnis befindet sich dort, wo Zwischendateien gespeichert werden. Für jedes Paket wird ein Unterordner erstellt, in dem z.B. CMake aufgerufen wird.
- Das Installationsverzeichnis ist der Ort, an dem jedes Paket installiert wird. Standardmäßig wird jedes Paket in einem separaten Unterverzeichnis installiert.
- Das `log` Verzeichnis enthält verschiedene Protokollinformationen zu jedem Colcon-Aufruf.

## Compilieren eines Paketes / mehrerer Paketen

Warum ist ein Build-System erforderlich? Für eine einzelne Datei ist der Vorgang überschaubar

- Eingabe: Code und Konfigurationsdateien
- Ausgabe: Artifacts (Code, Daten, Dokumentation)
- Vorbereiten der Verwendung

Für ein ganzes Set von Paketen ist deutlich mehr Aufwand erforderlich:

- Voraussetzungen:
  - Verfügbarkeit von System-Abhängigkeiten (abhängige Pakete) [rosdep](#)
  - Setzen der notwendigen Umgebungsvariablen
- Eingabe: Code und Konfigurationsdateien der Pakete
- Build Prozess:
  - Berechnen der Abhängigkeiten und Festlegen einer Build-Reihenfolge
  - Bauen einzelner Pakete UND Installation, Update der Umgebung
- Ausgabe: Set von ausführbaren Knoten

**Merke:** ROS / ROS2 umfasst eine Fülle von Tools für die Organisation dieses Prozesses.

## Build-Tools

In ROS 2 gibt es zwei zentrale Komponenten:

- ament – das Buildsystem für einzelne Pakete
- colcon – das Build-Orchestrierungswerkzeug für ganze Workspaces

Aspekt	ament	colcon
Ebene	Paket-Buildsystem	Workspace-Build-Orchestrator
Aufgabe	Baut <i>ein einzelnes ROS-Paket</i> (Build, Install, Test)	Baut <i>alle Pakete</i> eines Workspaces in korrekter Reihenfolge
Rolle im ROS-Ökosystem	Definiert Buildregeln, Tests, Install-Skripte	Koordiniert Builds, löst Abhängigkeiten auf, steuert Logging
Eingaben	<code>package.xml</code> , <code>CMakeLists.txt</code> oder <code>setup.py</code>	Workspace mit mehreren Paketen ( <code>src/</code> )
Ausgaben	installierte Artefakte eines Pakets (Nodes, Libs, Header, Skripte)	kompletter Workspace: <code>build/</code> , <code>install/</code> , <code>log/</code>
Buildsprache / Technologie	CMake (amentcmake), Python (amentpython)	Python-basiertes CLI-Tool, generisch erweiterbar
Unterstützte Sprachen	C++, Python	alle Buildsysteme, u. a. ament, CMake, Python, Gradle

Hinweis: `colcon` ist nicht spezifisch für ROS 2 und kann verwendet werden, um beliebige Softwareprojekte zu bauen, die verschiedene Buildsysteme verwenden.

## Realisierung eines eigenen Paketes

Wir wollen die Funktionalität der `minimal_subscriber`/`minimal_publisher` Beispiel erweitern und einen neuen Knoten implementieren, der den Zählerwert nicht als Bestandteil eines strings kommuniziert sondern als separaten Zahlenwert.

Sie finden den Beispielcode im Repository dieses Kurses unter [Link](#)

### Stufe 1: Individuelles Msg-Format

```
> ros2 pkg create my_msg_package --build-type ament_cmake --dependencies rclcpp std_msgs
going to create a new package
package name: my_msg_package
destination directory: /home/zug/Desktop/SoftwareprojektRobotik/examples/07_ROS2Pakete/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['zug <Sebastian.Zug@informatik.tu-freiberg.de>']
licenses: ['TODO: License declaration']
build type: ament_cmake
dependencies: ['rclcpp', 'std_msgs']
creating folder ./my_msg_package
creating ./my_msg_package/package.xml
creating source and include folder
creating folder ./my_msg_package/src
creating folder ./my_msg_package/include/my_msg_package
creating ./my_msg_package/CMakeLists.txt
```

Nun ergänzen wir unsere eigentliche Beschreibungsdatei für die Definition der eigenen Nachricht. Dazu legen wir uns einen Ordner `msg` an und integrieren wir eine Datei `MyMsg.msg`.

```
int32 counter
string comment "Keine Evaluation des Wertes"
```

An diesem Beispiel wird deutlich, wie das Build System uns unterstützt. Unsere `json` Datei mit der Nachrichtenkonfiguration muss in entsprechende C++ Code transformiert werden. Hierfür ist das `rosidl_default_configurators` Tool verantwortlich. Folglich müssen wir CMake anweisen, dies als Abhängigkeit zu betrachten und einen entsprechenden Generierungsprozess zu starten.

Ergänzen wir also in `CMakeList.txt file` die Abhängigkeiten:

```
find_package(builtin_interfaces REQUIRED)
find_package(rosidl_default_generators REQUIRED)
```

Darüber hinaus müssen wir die Generierungsanweisung selbst integrieren:

```
rosidl_generate_interfaces(${PROJECT_NAME}
    "msg/MyMsg.msg"
    DEPENDENCIES ...)
```

```
    DEPENDENCIES std_msgs  
)
```

Um sicherzugehen, dass die Pakete auf unserem Rechner vorhanden sind, integrieren wir diese in die Paketbeschreibung `package.xml`:

```
<build_depend>builtin_interfaces</build_depend>  
<build_depend>rosidl_default_generators</build_depend>  
<exec_depend>builtin_interfaces</exec_depend>  
<exec_depend>rosidl_default_runtime</exec_depend>  
  
<member_of_group>rosidl_interface_packages</member_of_group>
```

Ready for take off? Die Kompilierung kann starten.

```
> colcon build --symlink-install  
Starting >>> my_msg_package  
Finished <<< my_msg_package [4.59s]  
  
Summary: 1 package finished [4.68s]
```

Lassen Sie uns das neue Paket zunächst auf der Kommandozeile testen. Dazu wird das neu definierte Work-Package in die Liste der verfügbaren Pakete aufgenommen.

```
> source install/setup.bash  
> ros2 interface list | grep my  
my_msg_package/msg/MyMsg  
> ros2 topic pub /tralla my_msg_package/msg/MyMsg "{counter: 8}"
```

```
> source install/setup.bash  
> ros2 topic echo /tralla
```

## Schritt 2: Integration einer Methode

Nunmehr wollen wir die neu definierte Nachricht auch in einem Node verwenden. Entsprechend nutzen wir den `minimal_publisher` Beispiel aus der vergangenen Vorlesung und ersetzen die `String` Message gegen unsere `My_Msg` Implementierung. Dafür muss für den Knoten eine Abhängigkeit zum Paket `my_msg_package` spezifiziert werden. Dies kann während der Initialisierung des Paketes oder im Anschluss anhand der 'package.xml' und 'CMakeList.txt' erfolgen. Schauen Sie sich noch mal die Definition der Abhängigkeiten in unserem `my_msg_package` an.

```
> ros2 pkg create my_tutorial_package --build-type ament_cmake --node-name  
    data_generator --dependencies rclcpp std_msgs my_msg_package  
going to create a new package  
package name: my_tutorial_package  
destination directory: /home/zug/Desktop/VL_SoftwareprojektRobotik/examples  
/07_PDS_Pakete/src
```

```
/usr/include/sil
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['zug <Sebastian.Zug@informatik.tu-freiberg.de>']
licenses: ['TODO: License declaration']
build type: ament_cmake
dependencies: ['rclcpp', 'my_tutorial_package']
node_name: data_generator
creating folder ./my_tutorial_package
creating ./my_tutorial_package/package.xml
creating source and include folder
creating folder ./my_tutorial_package/src
creating folder ./my_tutorial_package/include/my_tutorial_package
creating ./my_tutorial_package/CMakeLists.txt
creating ./my_tutorial_package/src/data_generator.cpp
```

Prüfen Sie in der `CMakeList.txt` des Paketes `my_tutorial_package`, dass die notwendigen Abhängigkeiten (in Abhängigkeit von den dependencies) vollständig integriert wurden.

```
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
find_package(my_msg_package REQUIRED)

add_executable(data_generator src/data_generator.cpp)
ament_target_dependencies(data_generator rclcpp std_msgs my_msg_package)
```

Analoge Ergänzungen sind für `package.xml` notwendig. Wiederum werden die Basispakete `rclcpp` und `std_msgs` referenziert. Dazu kommt unser Message-Paket. Mit der Vorgabe der Abhängigkeiten werden diese Einträge automatisch generiert.

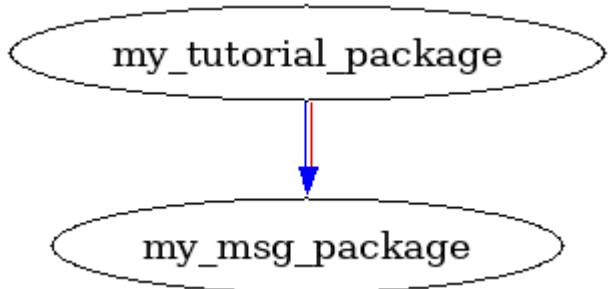
```
<depend>rclcpp</depend>
<depend>my_msg_package</depend>
```

Danach kann der übergreifende Build-Prozess gestartet werden.

```
colcon build --symlink-install
Starting >>> my_msg_package
Starting >>> my_tutorial_package
Finished <<< my_msg_package [0.59s]
Finished <<< my_tutorial_package [1.75s]

Summary: 2 packages finished [1.86s]
```

Für große Projekte kann es sinnvoll sein, die Abhängigkeiten grafisch zu analysieren. `colcon` stellt dafür mit `colcon graph --dot` eine Kommandozeilenoption bereit.



```

> colcon graph --dot
digraph graphname {
    "my_tutorial_package";
    "my_msg_package";
    "my_tutorial_package" -> "my_msg_package" [color="#0000ff:#ff0000"];
}

```

Bevor Sie eine der installierten ausführbaren Dateien oder Bibliotheken verwenden können, müssen Sie sie zu Ihrem Pfad und Bibliothekspfaden hinzufügen. colcon hat bash/bat-Dateien im Installationsverzeichnis generiert, um die Einrichtung der Umgebung zu erleichtern.

```

> ros2 run my_tutorial_package data_generator
hello world my_tutorial_package package

```

Soweit so gut. Nun müssen wir allerdings auch noch die Logik in die Anwendung integrieren.

## + data\_generator.cpp



```
#include <memory>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
#include "my_msg_package/msg/my_msg.hpp"

using std::placeholders::_1;

using namespace std::chrono_literals;

class MinimalPublisher : public rclcpp::Node
{
public:
    MinimalPublisher()
    : Node("minimal_publisher"), count_(0)
    {
        publisher_ = this->create_publisher<my_msg_package::msg::MyMsg>("topic");
        timer_ = this->create_wall_timer(
            500ms, std::bind(&MinimalPublisher::timer_callback, this));
    }

private:
    void timer_callback()
    {
        auto message = my_msg_package::msg::MyMsg();
        message.counter = count_++;
        RCLCPP_INFO(this->get_logger(), "Publishing: '%d, %s'", message.count,
                    message.comment.c_str());
        publisher_->publish(message);
    }
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<my_msg_package::msg::MyMsg>::SharedPtr publisher_;
    size_t count_;
};

int main(int argc, char * argv[])
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<MinimalPublisher>());
    rclcpp::shutdown();
    return 0;
}
```

Nach einem weiteren Build-Prozess können wir das Paket nun im erwarteten Funktionsumfang starten. Einen guten Überblick zur Behandlung von eigenen Datentypen im originären Paket oder aber in einem anderen bietet:

<https://index.ros.org/doc/ros2/Tutorials/Rosidl-Tutorial/>

### Schritt 3: Python-Paket mit Subscriber für eigenen Datentyp

Während das vorherige Beispiel einen C++-Publisher demonstriert, zeigen wir nun, wie man einen Subscriber in Python implementiert, der unseren eigenen Nachrichtentyp `MyMsg` empfängt.

Zunächst erstellen wir ein Python-Paket mit dem `ament_python` Build-Type:

```
> ros2 pkg create my_python_subscriber --build-type ament_python --dependencies rclpy std_msgs my_msg_package
going to create a new package
package name: my_python_subscriber
destination directory: /home/zug/Desktop/SoftwareprojektRobotik/examples/07_ROS_Pakete/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['zug <Sebastian.Zug@informatik.tu-freiberg.de>']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: ['rclpy', 'std_msgs', 'my_msg_package']
creating folder ./my_python_subscriber
creating ./my_python_subscriber/package.xml
creating source folder
creating folder ./my_python_subscriber/my_python_subscriber
creating ./my_python_subscriber/setup.py
creating ./my_python_subscriber/setup.cfg
creating ./my_python_subscriber/resource/my_python_subscriber
creating ./my_python_subscriber/my_python_subscriber/__init__.py
creating ./my_python_subscriber/test/test_copyright.py
creating ./my_python_subscriber/test/test_flake8.py
creating ./my_python_subscriber/test/test_pep257.py
```

Die Struktur eines Python-Pakets unterscheidet sich von einem C++-Paket:

```
my_python_subscriber/
├── package.xml
├── resource/
│   └── my_python_subscriber
├── setup.cfg
├── setup.py
└── my_python_subscriber/
    └── __init__.py
    ...
    └── test/
        ├── test_copyright.py
        └── test_flake8.py
```

```
└── test_package.py  
    └── test_pep257.py
```

Anstelle einer `CMakeLists.txt` verwendet ein Python-Paket die Dateien `setup.py` und `setup.cfg` für die Build-Konfiguration.

Nun erstellen wir unseren Subscriber-Node in der Datei `my_python_subscriber/data_receiver.py`:

### data\_receiver.py

```
import rclpy
from rclpy.node import Node
from my_msg_package.msg import MyMsg

class MinimalSubscriber(Node):

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            MyMsg,
            'topic',
            self.listener_callback,
            10)
        self.subscription # prevent unused variable warning
        self.get_logger().info('Subscriber initialized and listening on topic "topic"')

    def listener_callback(self, msg):
        self.get_logger().info('Received: counter=%d, comment="%s"' % (msg.counter, msg.comment))

def main(args=None):
    rclpy.init(args=args)
    minimal_subscriber = MinimalSubscriber()

    try:
        rclpy.spin(minimal_subscriber)
    except KeyboardInterrupt:
        pass
    finally:
        minimal_subscriber.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Als nächstes müssen wir die `setup.py` Datei bearbeiten, um unseren Node als ausführbare Datei zu registrieren:

## setup.py

```
from setuptools import setup

package_name = 'my_python_subscriber'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
         ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='zug',
    maintainer_email='Sebastian.Zug@informatik.tu-freiberg.de',
    description='Python subscriber for custom message type',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'data_receiver = my_python_subscriber.data_receiver:main',
        ],
    },
)
```

Der wichtige Teil ist der `entry_points` Abschnitt, der definiert, dass die Funktion `main` aus `data_receiver.py` als ausführbares Programm `data_receiver` verfügbar gemacht wird.

Prüfen Sie die Abhängigkeiten in der `package.xml`:

## package.xml

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
  schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>my_python_subscriber</name>
  <version>0.0.0</version>
  <description>Python subscriber for custom message type</description>
  <maintainer email="Sebastian.Zug@informatik.tu-freiberg.de">zug</maintainer>
  <license>TODO: License declaration</license>

  <depend>rclpy</depend>
  <depend>std_msgs</depend>
  <depend>my_msg_package</depend>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

Wichtig: Python-Pakete verwenden `<depend>` statt der separaten `<build_depend>` und `<exec_depend>` Tags, die in C++-Paketen üblich sind. Die Abhängigkeit zu unserem `my_msg_package` ist essentiell, damit Python auf die generierten Message-Definitionen zugreifen kann.

Nun können wir alle Pakete gemeinsam bauen:

```
> colcon build --symlink-install
Starting >>> my_msg_package
Starting >>> my_python_subscriber
Finished <<< my_msg_package [0.59s]
Finished <<< my_python_subscriber [0.82s]

Summary: 2 packages finished [1.12s]
```

Die Option `--symlink-install` ist besonders nützlich bei Python-Paketen, da sie symbolische Links erstellt anstatt die Dateien zu kopieren. Das bedeutet, dass Änderungen am Python-Code sofort wirksam werden, ohne dass ein erneuter Build notwendig ist.

Nach dem Sourcen der Umgebung können wir den Subscriber starten:

```
> source install/setup.bash
> ros2 run my_python_subscriber data_receiver
[INFO] [minimal_subscriber]: Subscriber initialized and listening on topic
```

```
[INFO] [minimal_publisher]: Subscriptor initialized and listening on topic  
"topic"
```

Testen Sie das Zusammenspiel von Publisher und Subscriber, indem Sie in separaten Terminals starten:

**Terminal 1 (Publisher):** `source install/setup.bash > ros2 run mytutorial/package data_generator [INFO]`  
[minimal\_publisher]: Publishing: '0, Keine Evaluation des Wertes' [INFO] [minimal\_publisher]: Publishing: '1, Keine Evaluation des Wertes' [INFO] [minimal\_publisher]: Publishing: '2, Keine Evaluation des Wertes'

**Terminal 2 (Subscriber):** `source install/setup.bash > ros2 run mypythonsubscriber data_receiver [INFO]`  
[minimal\_subscriber]: Subscriber initialized and listening on topic: "topic" [INFO] [minimal\_subscriber]: Received: counter=0, comment="Keine Evaluation des Wertes" [INFO] [minimal\_subscriber]: Received: counter=1, comment="Keine Evaluation des Wertes" [INFO] [minimal\_subscriber]: Received: counter=2, comment="Keine Evaluation des Wertes"

### Unterschiede zwischen C++ und Python Paketen:

Aspekt	C++ ( <code>amentcmake</code> ) / Python ( <code>amentpython</code> )	
Build-System	CMake	<code>setuptools</code>
Konfigurationsdatei	<code>CMakeLists.txt</code>	<code>setup.py, setup.cfg</code>
Quellcode-Ordner	<code>src/, include/</code>	<code>&lt;package_name&gt;/</code>
Node-Registrierung	<code>addexecutable() in CMakeLists.txt / entrypoints in setup.py</code>	
Abhängigkeiten	<code>find_package() in CMakeLists.txt</code>	<code>dependencies in setup.py</code>
Import	<code>#include &lt;package/msg/Type.hpp&gt;</code>	<code>from package.msg import Type</code>
Build-Geschwindigkeit	Langsamer (Kompilierung)	Schneller (keine Kompilierung)
<code>symlink-install</code>	Optional	Empfohlen (Änderungen sofort wirksam)

## Aufzeichnen des Prozesses

Die Aufzeichnung von ganzen Datensätzen ist einer der zentralen Debug-Techniken unter ROS. Das Mitschneiden von ausgetauschten Nachrichten:

- ermöglicht das automatisierte Testen von Algorithmen
- stellt eine Vergleichbarkeit der Evaluationen sicher
- macht reale Sensoren in einigen Situationen überflüssig.

Schauen wir uns das Ganze für einen Sensor an. Die Intel Realsense D435 eine RGB-D Kamera, mit einem ROS Interface. Der zugehörige Treiber findet sich unter folgendem [Link](#).

```
> ros2 run realsense_ros2_camera realsense_ros2_camera
> ros2 run image_tools showimage /image:=/camera/color/image_raw
```

`ros2 bag` ermöglicht die Aufzeichnung von allen Topics oder aber eine selektive Erfassung. Im Beispiel bedeutet dies:

```
> ros2 bag record -a
[INFO] [rosbag2_storage]: Opened database 'rosbag2_2019_12_06-20_43_24'.
[INFO] [rosbag2_transport]: Listening for topics...
[INFO] [rosbag2_transport]: Subscribed to topic '/rosout'
[INFO] [rosbag2_transport]: Subscribed to topic '/parameter_events'
[INFO] [rosbag2_transport]: Subscribed to topic '/camera/depth/color/points'
[INFO] [rosbag2_transport]: Subscribed to topic '/camera/infra2/image_rect_
[INFO] [rosbag2_transport]: Subscribed to topic '/camera/depth/camera_info'
...
> ros2 bag info rosbag2_2019_12_06-20_43_24/
Files:          rosbag2_2019_12_06-20_43_24.db3
Bag size:       1.0 GiB
Storage id:    sqlite3
Duration:      11.891s
Start:          Dec  6 2019 20:43:24.812 (1575661404.812)
End:            Dec  6 2019 20:43:36.704 (1575661416.704)
Messages:       954
Topic information: Topic: /camera/aligned_depth_to_color/camera_info | Type: sensor_msgs/msg/CameraInfo | Count: 88 | Serialization Format: cdr
Topic: /camera/aligned_depth_to_color/image_raw | Type: sensor_msgs/msg/Image | Count: 87 | Serialization Format: cdr
Topic: /camera/color/camera_info | Type: sensor_msgs/msg/CameraInfo | Count: 87 | Serialization Format: cdr
Topic: /camera/color/image_raw | Type: sensor_msgs/msg/Image | Count: 85 | Serialization Format: cdr
Topic: /camera/depth/camera_info | Type: sensor_msgs/msg/CameraInfo | Count: 89 | Serialization Format: cdr
Topic: /camera/depth/color/points | Type: sensor_msgs/msg/PointCloud2 | Count: 87 | Serialization Format: cdr
Topic: /camera/depth/image_rect_raw | Type: sensor_msgs/msg/Image | Count: 85 | Serialization Format: cdr
Topic: /camera/infra1/camera_info | Type: sensor_msgs/msg/CameraInfo | Count: 88 | Serialization Format: cdr
```

```
... /CameraInfo | Count: 88 | Serialization Format: cdr  
Topic: /camera/infra1/image_rect_raw | Type: sensor_msgs/  
     /Image | Count: 85 | Serialization Format: cdr  
Topic: /camera/infra2/camera_info | Type: sensor_msgs/ms  
     /CameraInfo | Count: 88 | Serialization Format: cdr  
Topic: /camera/infra2/image_rect_raw | Type: sensor_msgs/  
     /Image | Count: 85 | Serialization Format: cdr
```

Welche Informationen vermuten Sie hinter den einzelnen Einträgen, die hier publiziert werden? Wie gehen Sie vor, um deren Bedeutung zu ermitteln?

Mit `play` können Sie die Inhalte wieder abspielen.

```
> ros2 bag play rosbag2_2019_12_06-20_43_24.db3
```



Sehen Sie das Problem bei diesem Vorgehen, insbesondere im Zusammenhang mit RGB-D Daten? Die Messages wurden für etwa 11 Sekunden aufgenommen und trotzdem ist eine Datei mit einer Größe von einem GigaByte entstanden. Entsprechend ist es notwendig sich vor der Realisierung einer Aufzeichnung grundsätzlich Gedanken über die notwendigen Daten zu machen. Unter anderem sollten zwei Fehlkonfigurationen vermieden werden:

- Wenn zu wenige Daten aggregiert wurden, sinkt die Wiederverwendbarkeit des Datensatzes (vgl. `camera_info` Daten für overlays).
- Wenn zu viele Daten aggregiert wurden, wird die Performanz des Systems möglicherweise überstrapaziert. Die Bandbreite der Schreiboperationen auf dem Speichermedium muss die Datenrate entsprechend abdecken.

Ein Lösungsansatz ist die zeitliche Filterung der Informationen, in dem zum Beispiel nur jede 10te Nachricht gespeichert wird. Dies wiederum kann dann aber einen Einfluss auf das Verhalten des Algorithmus haben!

## Steuerung des Startprozesses

Sie sehen, dass wir immer weitere Konsolen öffnen, um einzelne Knoten zu starten und zu parametrisieren. Dieses Vorgehen ist für die Arbeit mit Anwendungen, die mehr als 3 Knoten umfassen ungeeignet. Dabei definiert der Entwickler ein Set von Anwendungshorizonten für den Einsatz des `launch` Systems:

- Komposition von Systemen in Systeme von Systemen zur Bewältigung der Komplexität
- Verwenden einer `include` semantic, um Fragmente wiederzuverwenden, anstatt jedes von Grund auf neu zu schreiben
- Definition von Gruppen, um Einstellungen (z.B. remappings) auf Sammlungen von Knoten/Prozessen/enthalteten Startdateien anzuwenden
- Verwendung von Gruppen mit Namensräumen, um Hierarchien zu bilden.
- Portabilität durch Abstraktion von Betriebssystemkonzepten, z.B. Umgebungsvariablen
- Dienstprogramme, um Dateien auf dem Dateisystem auf eine verschiebbare und portable Weise zu finden, z.B. `$(find <package_name>)`

ROS1 setzte dabei auf launch-Files, die eine XML Beschreibung der Konfiguration enthielten. Das zugehörige File für den Start eines Hokuyo Laserscanners hätte entsprechend folgende Struktur:

```
/launch/hokuyo_node.launch
```

```
<launch>
  <node name="hokuyo" pkg="hokuyo_node" type="hokuyo_node" respawn="false"
        = "screen">
    <param name="calibrate_time" type="bool" value="true"/>
    <param name="port" type="string" value="/dev/ttyACM0"/>
    <param name="intensity" type="bool" value="false"/>
    <param name="min_ang" value="-2.2689"/>
    <param name="max_ang" value="+2.2689"/>
    <param name="cluster" value="1"/>
    <param name="frame_id" value="hokuyo_laser_frame"/>
  </node>
</launch>
```

Üblicherweise wäre es in eine Hierarchie von launch-Skripts eingebunden, die es zum Beispiel für ein SLAM Projekt referenzieren.

```
<launch>
  <param name="pub_map_odom_transform" value="true"/>
  <param name="map_frame" value="map"/>
  <param name="base_frame" value="base_frame"/>
  <param name="odom_frame" value="odom"/>

  <include file="$(find hokuyo_node)/launch/hokuyo_node.launch"/>

  <node pkg="rviz" type="rviz" name="rviz" args="-d $(find hokuyo_hector_s1
    )/launch/rviz_cfg.rviz"/>

  <include file="$(find hokuyo_hector_slam)/launch/default_mapping.launch"/>
  <include file="$(find hector_geotiff)/launch/geotiff_mapper.launch"/>

</launch>
```

Dabei werden verschiedene Nachteile von ROS1-Launch beseitigt:

- Zuordnung von Nodes zu Prozessen
- Parameterdefinition
- ROS1 kannte kein deterministisches Verhalten während der Startup-Prozesse ... *roslaunch does not guarantee any particular order to the startup of nodes -- although this is a frequently requested feature, it is not one that has any particular meaning in the ROS architecture as there is no way to tell when a node is initialized.*
- "Dynamisierung" des Startup-Prozesses

Hierfür war es notwendig die bisherige XML basierte Notation der Launch-Files aufzugeben und durch eine Skript-Sprache zu ersetzen. Dabei lag es nahe, Python als Grundlage zu verwenden. Zur Illustration einiger Features wurde das Paket `my_tutorial_package` um einen minimalistischen Subscriber für unseren individuellen Message-Typ erweitert.

### launchNodes.launch.py

```
import launch
import launch.actions
import launch.substitutions
import launch_ros.actions

def generate_launch_description():
    return launch.LaunchDescription([
        launch.actions.DeclareLaunchArgument(
            'node_prefix',
            default_value=[launch.substitutions.EnvironmentVariable('USER')
                           ],
            description='Prefix for node names'
        ),
        launch_ros.actions.Node(
            package='my_tutorial_package', node_executable='data_generator',
            output='screen',
            node_name=[launch.substitutions.LaunchConfiguration('node_prefix')
                       'talker'],
            arguments = ['/topic:=/newtopic']
        )
    ])
```

Dann kann der Aufruf des Launch-Files wie folgt erfolgen:

```
# Mit Standardwert (z.B. "zug_")
ros2 launch my_tutorial_package launchNodes.launch.py

# Mit eigenem Wert
```

```
ros2 launch my_tutorial_package launchNodes.launch.py node_prefix:=robot_
```

Innerhalb eines C++ Paketes müssen launch-Files in der CMakeLists.txt spezifiziert werden, damit Sie in den Installationsprozess einfließen:

```
install(DIRECTORY
    launch
    DESTINATION share/${PROJECT_NAME}/
)
```

Welche Ergänzungen sind notwendig, um den Subscriber entsprechend abzubilden? Nutzen Sie das Launch-File, um das "System" zu starten.

```
ros2 launch my_tutorial_package launchNodes.launch.py
```

Warum empfängt der Subscriber die Nachrichten des Publishers nicht?

Weitergehende Möglichkeiten eröffnen zeitliche Trigger für die Aktivierung einzelner Knoten. Das nachfolgende Beispiel, aktiviert den Knoten `heart_beat` erst mit einer Verzögerung von 5 Sekunden.

### launchNodes.launch.py

```
import launch
import launch.actions
import launch.substitutions
import launch_ros.actions

def generate_launch_description():
    return launch.LaunchDescription([
        launch.actions.TimerAction(
            actions = [
                launch_ros.actions.Node( package='my_tutorial_package',
                    node_executable='heart_beat', output='screen')
            ], period = 5.0
        )
    ])
```

Die Implementierung kann aber auch ohne die Nutzung von `ros2 launch` als einfaches Python Skript erfolgen. Ein Beispiel findet sich unter folgendem [Link](#)

## Logging

ROS1 und ROS2 implementieren ein eigenes Log-System, dass sowohl über Kommandozeilentools, wie auch mittels Python/C++ API konfiguriert werden kann. Den Protokollmeldungen ist eine Schweregrad zugeordnet, der eine Aktivierung oberhalb eines bestimmten Levels erlaubt. `DEBUG`, `INFO`, `WARNING`,

`ERROR` oder `FATAL`, in aufsteigender Reihenfolge.

Jedem Knoten ist ein Logger zugeordnet, der automatisch den Namen und den Namensraum des Knotens enthält. Wenn der Name des Knotens extern auf etwas anderes umgestellt wird, als im Quellcode definiert ist, wird er im Loggernamen reflektiert (vgl. Launch-File Beispiel).

```
ros2 run demo_nodes_cpp listener __log_level:=debug  
export RCUTILS_COLORIZED_OUTPUT=0 # 1 for forcing it or, on Windows:  
# set "RCUTILS_COLORIZED_OUTPUT=0"
```



```
rcutils_logging_set_logger_level("logger_name", RCUTILS_LOG_SEVERITY_DEBUG)
```

Das Loggingsystem unter ROS2 bildet folgende Makros ab, die in der rclcpp API enthalten sind ([Link](#)). Dabei zielt die Neuimplementierung aber darauf ab, ein Interface zu definieren, das es erlaubt Logging-Bibliotheken allgemein einzubetten:

Makro	Signatur	Bedeutung
<code>RCLCPP_x</code>	logger	Die Ausgabe der Log-Nachricht erfolgt bei jedem Durchlauf.
<code>RCLCPP_x_ONCE</code>	logger	Die Ausgabe erfolgt nur einmalig.
<code>RCLCPP_x_SKIPFIRST</code>	logger	Beim ersten Durchlauf wird die Ausgabe unterdrückt, danach immer realisiert.
<code>RCLCPP_x_EXPRESS</code> <code>ION</code>	logger, expression	Die Ausgabe erfolgt nur für den Fall, dass der Ausdruck war ist.
<code>RCLCPP_x_FUNCTION</code>	logger, function	Die Ausgabe wird nur realisiert, wenn der Rückgabewert der Funktion wahr war.
<code>RCLCPP_x_THROTTLE</code>	logger, clock, duration	Die Ausgabe erfolgt nur mit einer entsprechenden Periodizität, die über <code>duration</code> definiert wird.

Anwendungsbeispiele:

## Logging.cpp



```
// This message will be logged only the first time this line is reached.  
RCLCPP_INFO_ONCE(this->get_logger(), "This message will appear only once")  
// Filter even counter values  
RCLCPP_DEBUG_EXPRESSION(  
    this->get_logger(), (counter % 2) == 0, "Count is even (expression eval  
    to true)");  
// Detect range overrun  
if (counter >= 2147483647) {  
    RCLCPP_FATAL(this->get_logger(), "Resetting count to 0");  
    counter = 0;  
}
```

Erweitern Sie die Ausgaben unseres Beispiels um eine vorgefilterte Ausgabe, so dass nur noch alle 10 Sekunden die entsprechenden Log-Nachrichten erscheinen.

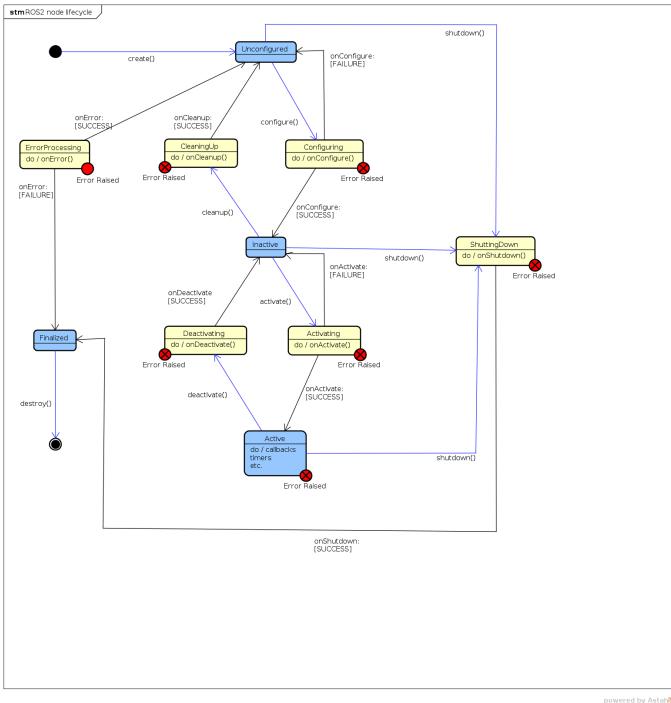
## Erweiterung des Knotenkonzepts

Ein verwalteter Lebenszyklus für Knoten (*managed nodes*) ermöglicht eine bessere Kontrolle über den Zustand des ROS-Systems. Es ermöglicht dem ros2 launch, sicherzustellen, dass alle Komponenten korrekt instantiiert wurden, bevor es einer Komponente erlaubt, mit der Ausführung ihres Verhaltens zu beginnen. Es ermöglicht auch, dass Knoten online neu gestartet oder ersetzt werden können.

Das wichtigste Konzept dieses Dokuments ist, dass ein verwalteter Knoten eine bekannte Schnittstelle darstellt, nach einem bekannten Lebenszyklus-Zustandsautomaten ausgeführt wird und ansonsten als Blackbox betrachtet werden kann.

ROS2 definiert vier Zustände `Unconfigured`, `Inactive`, `Active`, `Finalized` und insgesamt 7 Transitionen.

Autor: Geoffrey Biggs Tully Foote, [https://design.ros2.org/articles/node\\_lifecycle.html](https://design.ros2.org/articles/node_lifecycle.html)



Für die Interaktion mit einem *managed node* stehen Ihnen unterschiedlichen Möglichkeiten offen. Auf der Kommandozeile kann zwischen den States mittels

```
ros2 lifecycle set /nodename X #State Id
```



gewechselt werden. Komfortabler ist die Spezifikation in den launch-Files. Ein Beispiel für die entsprechend Realisierung findet sich unter folgendem [Link](#)