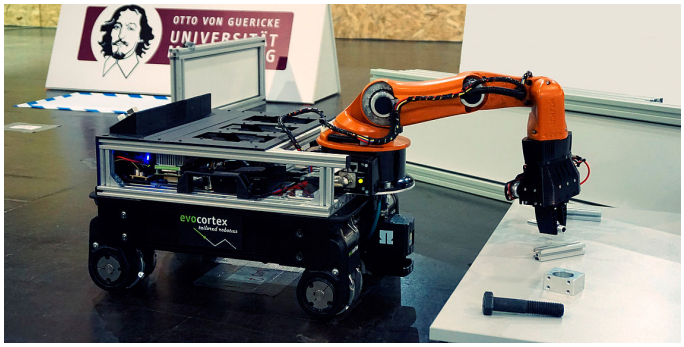
Einführung in ROS2

Parameter	Kursinformationen
Veranstaltung:	`Softwareprojekt Robotik`
Semester	`Wintersemester 2023/24`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Motivation für die Verwendung von ROS`
Link auf GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/06_EinfuehrungROS.md
Autoren	Sebastian Zug & Georg Jäger

Zielstellung der heutigen Veranstaltung

- Spezifische Anforderungen an Frameworks für die Arbeit mit Robotern
- Basiskonzepte von ROS
- Unterschiede von ROS1 und ROS2

Welche Herausforderungen stecken in der Programmierung eines Roboters?



Roboter des RoboCupTeams aus Nürnberg [1]

[1] : TDP des Teams AutonOhm, 2019

Comic auf der Webseite der Firma Willow Garage, das die individuellen Frameworks für die Robotikentwicklung adressiert. [2]

[2] : Willow Garage, http://www.willowgarage.com/blog/2010/04/27/reinventing-wheel, 2010

Vergleich von Frameworks

1. Hardwareunterstützung und Laufzeitumgebung

- **Betriebssystem** Eine Robotikentwicklungsumgebung sollte mehrere Betriebssysteme und dabei eine möglichst umfangreiche Abdeckung für häufig genutzte Bibliotheken, Komponenten oder Hardwaretreiber bieten.
- **Unterstützung ressourcenbeschränkter Systeme** Die Interaktion und Kooperation mit ressourcenbeschränkten Systemen ist u.a. in Bereichen gefragt, in denen z.B. die Effizienz und Kosten der eingesetzten Komponenten eine tragende Rolle spielen. Für ein breites Anwendungsspektrum des jeweiligen Frameworks ist eine entsprechende Unterstützung solcher Systeme wünschenswert.
- **Echtzeitfähigkeit** Robotikanwendungen umfassen häufig Anwendungselemente, die harte Echtzeitanforderungen an die Verarbeitung stellen. Regelschleifen zur Ansteuerung von Manipulatoren benötigen, um stabil zu laufen, ein deterministisches Zeitverhalten.
- **Treiberintegration** Ein Framework sollte nicht nur eine die Vielfalt an Sensoren und Aktoren, sondern auch konkrete Robotersysteme spezifischer Hersteller, die auf diesen Komponenten aufsetzen, unterstützen.

2. Kommunikation

- **Kommunikationsmiddleware** Damit Anwendungen verteilt über mehrere Rechnerknoten laufen können und somit eine Ortsunabhängigkeit gewährleisten, sind entsprechende Mechanismen erforderlich.
- **Kommunikationsparadigmen** Im Kontext einer Anwendung ist die Abdeckung unterschiedlicher Formen für den Nachrichtenaustausch zwischen den Komponenten wünschenswert. Als Interaktionsmuster sind die *Client-Server-Beziehung* und das *Publish-Subscribe-Modell* denkbar.
- **Echtzeitfähigkeit** Anknüpfend an die Echtzeitfähigkeit der Laufzeitumgebung ist das deterministische Verhalten der Kommunikationsverbindungen Voraussetzung für die Entwicklung zeitkritischer Systeme.

3. Programmierung

- **Unterstützte Programmiersprachen** Bei der Anwendungsentwicklung sollte dem Entwickler möglichst die Wahl gelassen werden, in welcher Programmiersprache entwickelt wird. Eine domain-spezifische Frage zielt dabei auf die Möglichkeit der grafischen Programmierung.
- **Unterstützungsbibliotheken** Vordefinierte Komponenten z.B. für Pfadplanung, Verhaltensauswahl und Lokalisierung erleichtern den Entwicklungsprozess und fördern die Wiederverwendung von Software-Modulen, wobei gegebenenfalls entsprechende Anpassungen erforderlich sind.
- **Erweiterbarkeit** Erweiterbarkeit bedeutet hier die Unterstützung für das Hinzufügen neuer Software-Module und neuer Hardware-Komponenten in das bestehende Rahmenwerk.
- **Lizenzmodell** Der Typ der Lizenz der Frameworks bestimmt insbesondere im Fall der kommerziellen Nutzung über deren generelle Anwendbarkeit. Durch das gewählte Lizenzmodell wird die Breite der Entwicklungs-Community zumindest mitbestimmt. Eine aktive Community erleichtert die Entwicklungsarbeit und bieten in Wikis oder Foren eine Vielzahl von Antworten, Anregungen und Beispielcode.

4. Test und Debugging

- **Monitoring** Die Überwachung der einzelnen Komponenten und deren Beziehungen zueinander muss in einem übergreifenden Ansatz möglich sein, um komfortabel Aussagen über den Status des Robotersystemes. Eine grafische Schnittstelle, die die Visualisierung einzelner Komponenten, des Gesamtsystems oder einzelner Parameter übernimmt, vereinfacht die Entwicklung erheblich.
- **Logging** Das Logging der Anwendungsoperation unterstützt einerseits das Debugging und ermöglicht andererseits eine Wiederholung dieser Anwendungsausführung im Sinne eines Wiederabspielens einer Aufzeichnung. Somit wird eine Offline-Analyse der implementierten Funktionalitäten möglich, sodass auch Aussagen über die Performance dieser bzw. des Gesamtsystems getroffen werden können.
- **Simulation** Die Simulation der realen Welt ermöglicht es den Entwicklern, ihre Anwendungen zu testen, ohne die entsprechende Hardware besitzen zu müssen, indem diese geeignet modelliert wird. Die Simulatoren können dabei in Form von „einfachen“ zweidimensionalen bis hin zu komplexen 3-D-Umsetzungen mit realistischen physikalischen Gegebenheiten vorliegen.

ROS, was ist das?

Robot Operating System (ROS) ist ein Framework für die Entwicklung von Robotern. Dabei ist ROS kein Betriebssystem sondern eine Middleware mit aufgesetzter Paketstruktur. 2020 listeten die Statistiken mehrere tausend Repositories und Pakete. ROS wird unter der BSD-Lizenz veröffentlicht und ist somit der Open-Source-Szene zuzuordnen.

Die Entwicklung begann 2007 am Stanford Artificial Intelligence Laboratory im Rahmen des Stanford-AI-Robot-Projektes (STAIR) und wurde ab 2009 hauptsächlich am Robotikinstitut Willow Garage weiterentwickelt. Seit April 2012 wird ROS von der neu gegründeten, gemeinnützigen Organisation Open Source Robotics Foundation (OSRF) unterstützt und seit Beendigung der operativen Tätigkeit von Willow Garage 2013 von dieser koordiniert, gepflegt und weiterentwickelt.

Die Hauptbestandteile und -aufgaben von ROS sind:

- Hardwareabstraktion
- Gerätetreiber
- oft wiederverwendete Funktionalität
- Nachrichtenaustausch zwischen Programmen bzw. Programmteilen
- Paketverwaltung
- Programmbibliotheken zum Verwalten und Betreiben der Software auf mehreren Computern

Die Webseite von ROS findet sich unter ROS.

Seit 2013 beschäftigt sich das ROS Industrial Consortium mit der Förderung und Unterstützung von ROS für Anwendungen in der Industrierobotik. In Europa koordiniert das Fraunhofer IPA (Stuttgart) die Aktivitäten des ROS Industrial Consortium Europe.

Seit Beginn der Entwicklung von ROS 2.0 wird zwischen ROS 1 und ROS 2 unterschieden. Die beiden Hauptversionen sind nicht miteinander kompatibel, jedoch interoperabel und können parallel ausgeführt werden.

Die aktuellen Versionen sind

- ROS1 - *Noetic Ninjemys* (Noetische Ninjemys Oweni) veröffentlicht im Mai 2020 und
- ROS2 - *Humble Hawksbill* (bescheidene Echte Karettschildkröte) Veröffentlichung im Mai 2021 (LTS Version).
- ROS2 - *Iron Irwini* (eiserne Irwins Schildkröte) Veröffentlichung im Mai 2023.

> In der Fachwelt für das autonome Fahren werden auch gerne zumindest Teile von ROS eingesetzt. In der Robotik nutzen mittlerweile nahezu alle Forschungsgruppen zumindest teilweise ROS. Viele Forschungsgruppen besitzen gar kein eigenes Softwareframework mehr, sondern konzentrieren sich voll auf ROS. *golem.de Beitrag - Für wen ist ROS?*

Das erste Paper, in dem die Basiskonzepte beschrieben wurden, ist unter Link zu finden.

ROS 1 vs. ROS 2

Merkmal	ROS1	ROS2
Betriebssysteme	Linux (Ubuntu, Gentoo, Debian), OS X	Linux (Ubuntu), OS X, Windows 10
Programmiersprachen C	C++03	C++11 (einzelne Konzepte von C++14)
Programmiersprachen Python	Python 2	Python 3.5
Middleware	eigene Konzepte für den Datenaustausch	abstraktes Middlewarekonzept offen für spezifische Lösungen (aktuell DDS als Default-Implementierung), mit einem geeigneten Middleware-Ansatz sind Echtzeitanwendungen umsetzbar
Build System	CMake Projekte mit catkin	CMake Projekte (mit colcon), andere Systeme können integriert werden
Paketstruktur	Integrierende Pakete möglich	aus Gründen der automatisierten Auflösung von Abhängigkeiten nur isolierte Pakete
Message/Service Deklaration	Messageformatdefinitionen auf der Basis von Grundtypen,	einheitliche Namen in Python und C++, default Werte, separate Namespaces für Services und Messages, einheitliche Zeittypen für die APIs
Systemkonfiguration	XML Beschreibungen	Python Code für den Systemstart mit komplexen Logiken

Node vs. Nodelet	unterschiedliche APIs für beide Konzepte	Implementierungen zielen auf eine Shared Library - es wird zur Laufzeit entschieden, ob diese als separater Prozess oder innerhalb eines Prozesses ausgeführt wird.

Einen Überblick zu den genannten Features gibt die Webseite Link

Ersetzt ROS2 als ROS1 vollständig?

Blick zurück ...

> *The quick answer is: Yes but with drawbacks. So ROS2 Crystal has worked with the new Gazebo with ROS2 support, so you have access to creating your own simulations using only ROS2. You have access to the main ROS packages like tf, navigation, image_transport, rqt, and the big RVIZ. So it would seem that 90% of the current ROS users would be satisfied. ...*

> *But the reality is that a huge amount of packages don't come out of the box working for ROS2 or are installed through debians.* [Ref]

[Ref]: The Construct, "ROS2 vs ROS1? Or more like ROS2 + ROS1?" https://www.theconstructsim.com/ros2-vs-ros1/, 2018

> Akutell sollten neue Projekte immer auf ROS2 aufsetzen!

Falls es Probleme bei der Umsetzung einzelner Pakete gibt, bietet das Paket `ros1_bridge` Hilfe, das die Kommunikation zwischen einem ROS1 und einen ROS2 Graphen sicherstellt.

Wie kann man sich in ROS einarbeiten?

1. Das offizielle ROS-Tutorial-Website ist sehr umfangreich und in mehreren Sprachen verfügbar. Es enthält Details zur ROS-Installation, Dokumentation von ROS, etc. und ist völlig kostenlos. Dabei lauern aber einige Fallstricke:
    - Achten Sie immer, wenn Sie sich in ein Beispiel einlesen auf die zugehörige ROS-Versionsnummer!
    - Prüfen Sie Abhängigkeiten und die Aktualität der Bibliotheken.
    - Informieren Sie sich darüber in wie weit an dem Paket aktuell noch gearbeitet wird. Letzte Commits vor einigen Monaten sind immer ein schlechtes Zeichen 🙂

ROS2	ROS1	Hintergrund
https://index.ros.org/doc/ros2/	http://wiki.ros.org/	Hauptseite des Projektes OSF
https://discourse.ros.org/	https://answers.ros.org/questions/	ROS Forum
https://index.ros.org/doc/ros2/Tutorials/#tutorials	http://wiki.ros.org/ROS/Tutorials	ROS Tutorials

2. Es existiert eine Vielzahl von Tutorials in Form von Videos, die einen Überblick versprechen oder einzelne Themen individuell adressieren.

Titel	Inhalt	Link
ROS tutorial #1	Installation, erste Schritte	Link
Programming for Robotics	5 Kurse als Einführung in ROS1 der ETH Zürich	Link
ROS2 Tutorials	Tutorial des kommerziell orientierten Kursanbieters "The Construct"	Link

3. Verschiedene Hochschulen und Institutionen bieten Kurse und Summer Schools an. Achtung, diese sind teilweise kostenpflichtig!
4. Zu empfehlen ist das Buch von Newmann "A Systematic Approach to Learning Robot Programming with ROS" oder aber von Kane "A Gentle Introduction to ROS". Letzteres ist online unter Link zu finden. Beide beziehen sich aber auf ROS 1.

Basiskonzepte

Pakete - Pakete kapseln einzelne Algorithmen und realisieren deren Abhängigkeiten. Letztendlich wird damit die Wiederverwendbarkeit einer Implementierung gewährleistet.

https://fkromer.github.io/awesome-ros2/

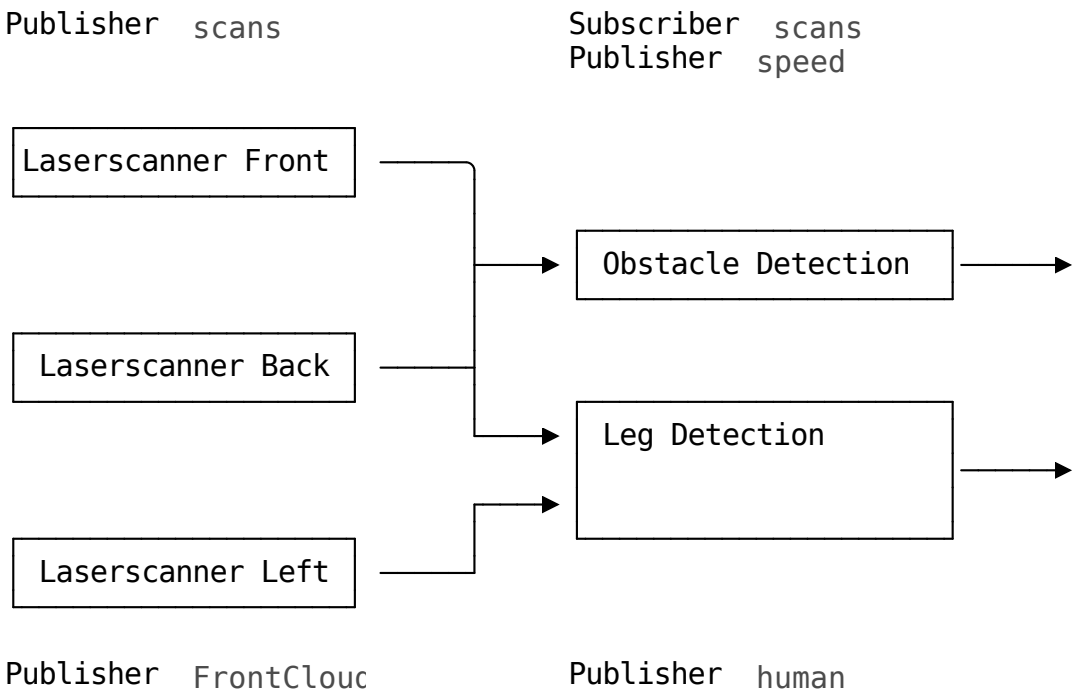
Node - Ein Knoten ist Teilnehmer im ROS-Graphen. ROS-Knoten verwenden eine ROS-Clientbibliothek, um mit anderen Knoten zu kommunizieren. Knoten können ein *Subject* veröffentlichen oder abonnieren. *Nodes* können auch einen Dienst bereitstellen oder verwenden. Einem Knoten sind konfigurierbare Parameter zugeordnet. Verbindungen zwischen Knoten werden durch einen verteilten Erkennungsprozess hergestellt. Knoten können sich im selben Prozess, in unterschiedlichen Prozessen oder auf unterschiedlichen Rechnern befinden.

> *Screenshot der Knoten eines umfangreicheren Projektes. Die Ellipsen repräsentieren die Knoten, die rechteckigen Boxen die "Datenkanäle" dazwischen.*

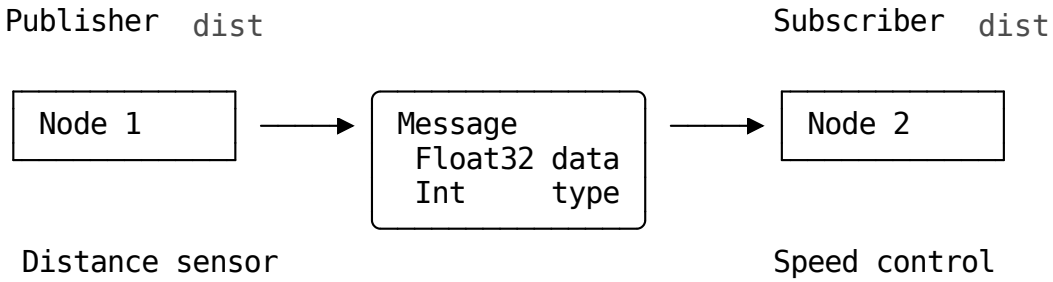
Discovery - Die Erkennung von *Nodes* erfolgt automatisch über die zugrunde liegende Middleware von ROS2. Dafür sind folgende Punkte zu beachten

- Wenn ein Knoten gestartet wird, kündigt er seine Anwesenheit anderen Knoten im Netzwerk mit derselben ROS-Domäne an (festgelegt mit der Umgebungsvariablen `ROS_DOMAIN_ID`). Knoten antworten auf diese Ankündigung mit Informationen über sich selbst, so dass die entsprechenden Verbindungen hergestellt werden können und die Knoten kommunizieren können.
- Knoten informieren regelmäßig über ihre Anwesenheit, damit Verbindungen zu neu erscheinenden Entitäten hergestellt werden können, die während des eigenen Starts noch nicht aktiv waren.
- Knoten stellen nur dann Verbindungen zu anderen Knoten her, wenn diese über kompatible Quality of Service-Einstellungen verfügen.

Topics - Topics repräsentieren den Inhalt einer Nachricht und erlauben damit die Entkopplung von Informationsquelle und Informationssenke. Die Knoten brauchen nicht zu wissen, mit wem sie kommunizieren, allein das "Label" oder "Thema" genügt. *Topics* sind für die unidirektionale, streamende Kommunikation gedacht. *Nodes*, die *Remote Procedure Calls* durchführen müssen, d.h. eine Antwort auf eine Anfrage erhalten, sollten stattdessen *Services* verwenden.



Messages - Um die Kommunikation von Datenpaketen zwischen den Knoten zu ermöglichen, muss deren Aufbau und inhaltliches Format spezifiziert werden. Welche Datenformate werden verwendet, wo befindet sich der versendende Sensor, welche Maßeinheiten bilden die Informationen ab? ROS definiert dafür abstrakte Message-Typen.



Die ROS2 Message-Spezifikation integriert verschiedene Konfigurationsmöglichkeiten. Auf der obersten Ebene sind dies einfache Namens- und Typzuweisungen. Dabei wird zwischen sogenannten Built-in Typen und nutzerspezifischen Typen unterschieden. Feldnamen müssen klein geschriebene alphanumerische Zeichen mit Unterstrichen zur Trennung von Wörtern sein. Die Typdefinitionen der Basistypen erfolgen anhand "C++ naher" Bezeichner (`bool` , `char` , `float32` usw.)

Komplexe Typen werden wie folgt spezifiziert

Index	ROS2 msg Type	C++
0	zB. `float32`	`float`
1	`string`	`std::string`
2	static array	`std::array<T, N>`
3	unbounded dynamic array	`std::vector<T>`
4	bounded dynamic array	custom_class<T,N>
5	bounded string	`std::string`

Im folgenden sollen Beispiele die

```
# Basic format: fieldtype1 fieldname1
# Type 0, 1 examples:
int32 my_int
string my_string

# Type 2
int32[5] five_integers_array
# Type 3
int32[] unbounded_integer_array
# Type 4
int32[<=5] up_to_five_integers_array

# Type 5
string<=10 up_to_ten_characters_string
string[<=5] up_to_five_unbounded_strings
string<=10[] unbounded_array_of_string_up_to_ten_characters each
string<=10[<=5] up_to_five_strings_up_to_ten_characters_each
```

Eine weitere Neuerung besteht in der Möglichkeit Default-Werte und Konstanten zu definieren.

```
# Field default values
uint8 x 42
int16 y -2000
string full_name "John Doe"
int32[] samples [-200, -100, 0, 100, 200]

# Constant values with "="
int32 X=123
string FOO="foo"
```

Eigene Messagetypen umfassen üblicherweise eine Hierarchie von Messages und Sub-Messages. Untersuchen Sie zum Beispiel das Standard-Laserscanner Nachrichtenformat:

```
> ros2 msg show sensor_msgs/msg/LaserScan
```

Knoten, die einen Wert publizieren lassen sich neben den Programmen auch auf der Kommandozeile erzeugen. Damit besteht für Tests eigener Subscriber die Möglichkeit diese sehr einfach mit spezifischen Nachrichten zu triggern.

```
ros2 topic pub /test s
```

Worin unterscheiden sich ROS1 und ROS2 in Bezug auf diese Konzepte?

Einen Überblick bietet die Webseite unter folgendem Link

Parameter	ROS2 node	ROS1 node
Zweck	Ausführbares Programm im ROS1 Kontext, das in der Lage ist mit anderen Knoten zu kommunizieren	Ausführbares Programm im ROS1 Kontext, das in der Lage ist mit anderen Knoten zu kommunizieren
Discovery	Verteilte Discovery-Mechanismen (die nicht von einem einzelnen Knoten abhängen)	ROS Master als zentrale Verwaltungsinstanz der Kommunikation
Client Bibliotheken	`rclcpp` = C++ client Library, `rclpy` = Python client library C++	`roscpp` = C++ client Library, `rospy` = Python client library

Einführungsbeispiele

Arbeit auf der Konsole

Die Exploration und Untersuchung eines ROS2 Systems erfolgt mittels des Tools "ros2". Mit diesem können die folgenden Konzepte adressiert werden. Dazu bietet das Tool folgendende API:

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

options:
  -h, --help            show this help message and exit
  --use-python-default-buffering
```

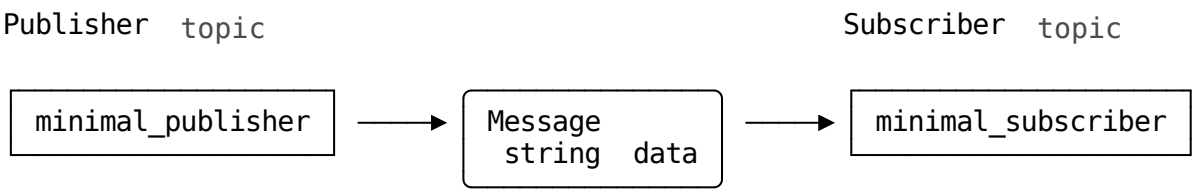
```
                Do not force line buffering in stdout and instead u
                the python default buffering, which might be affect
                by PYTHONUNBUFFERED/-u and depends on whatever stdo
                is interactive or not

Commands:
  action     Various action related sub-commands
  bag        Various rosbag related sub-commands
  component  Various component related sub-commands
  daemon     Various daemon related sub-commands
  doctor     Check ROS setup and other potential issues
  interface  Show information about ROS interfaces
  launch     Run a launch file
  lifecycle  Various lifecycle related sub-commands
  multicast  Various multicast related sub-commands
  node       Various node related sub-commands
  param      Various param related sub-commands
  pkg        Various package related sub-commands
  run        Run a package specific executable
  security   Various security related sub-commands
  service    Various service related sub-commands
  topic      Various topic related sub-commands
  wtf        Use `wtf` as alias to `doctor`

  Call `ros2 <command> -h` for more detailed usage.
```

Hello-World Implementierung

Wir versuchen das "Hello World"-Beispiel der ROS Community nachzuvollziehen, dass zwei einfache Knoten - "minimal publisher" und "minimal subscriber" - definiert.



Eine entsprechende Kommentierung eines ähnlichen Codes findet sich auf der ROS2 Webseite.

Publisher.cpp

```cpp
#include <chrono>
#include <memory>

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

using namespace std::chrono_literals;

class MinimalPublisher : public rclcpp::Node
{
  public:
    MinimalPublisher()
    : Node("minimal_publisher"), count_(0)
    {
      publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 1
      timer_ = this->create_wall_timer(500ms,
                                       std::bind(&MinimalPublisher::timer_callback, thi
    }

  private:
    void timer_callback()
    {
      auto message = std_msgs::msg::String();
      message.data = "Hello, world! " + std::to_string(count_++);
      RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_st
      publisher_->publish(message);
    }
    rclcpp::TimerBase::SharedPtr timer_;
    rclcpp::Publisher<std_msgs::msg::String>::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;
}
```

Zeile	Bedeutung
1-2	Die `chrono`-Bibliothek ist eine Sammlung von flexiblen Typen, die das Benutzen von Zeiten, Zeiträumen und Zeitpunkten mit unterschiedlichen Präzisionsgraden ermöglicht (Link). `memory` wird für die Verwendung von der Smart Pointer benötigt (Link).
4-5	Integration der ROS2 spezifischen API-Definitionen für C++. Der Header `rclcpp.hpp` muss in jedem Fall eingebettet werden. Der avisierte Message Type `String` wird zudem inkludiert.
9	Definition einer individuellen Knotenklasse, die von der API-Basisklasse `rclcpp::Node` erbt.
12	Der zugehörige Konstruktor der Klasse `MinimalPublisher()` ruft den Konstruktor der Basisklasse `Node` auf und initialisiert die Member `node_name` und eine eigene Variable `count_`. Diese dient als variabler Dateninhalt unserer Kommunikation.
14	Wir erzeugen einen Publisher, der über die Memebervariable `publisher_` referenziert wird, der Nachrichten vom Typ `std_msgs::msg::String` versendet (Templateparameter), das entsprechende Topic lautet "topic" (Konstruktorparameter).
15	Ein Timer-Objekt wird erzeugt und initialisiert. Zum einen spezifizieren wir die Periodendauer und zum anderen eine aufzurufende Funktion. In diesem Fall ist das unsere private Methode `timer_callback()` die als impliziten Parameter `this` übergeben bekommt.
20	In der Funktion `callback()` wird die eigentliche Funktionalität des Publishers, das versenden einer Nachricht, realisiert. Dazu wird zunächst ein entsprechender String befüllt und mittels des Makros `RCLCPP_INFO` auf der Konsole ausgegeben. Anschließend wird die Methode `publish()` mit unserer Nachricht als Parameter aufgerufen.
32	`init` aktiviert die abstrakte Middelwareschnittstelle von ROS2 für einen Knoten. Dabei können unterschiedliche Parameter für die Konfiguration (zum Beispiel Logging-Levels, Kommunikationsparameter, usw.) übergeben werden.
33	Aktivierung des Knotens über den einmaligen Aufruf von `spin()`.

Diese Funktion sollte der Knoten nur im Fehlerfall verlassen.

Subscriber.cpp

```
#include <memory>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
using std::placeholders::_1;

class MinimalSubscriber : public rclcpp::Node
{
  public:
    MinimalSubscriber()
    : Node("minimal_subscriber")
    {
      subscription_ = this->create_subscription<std_msgs::msg::String>(
      "topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
    }

  private:
    void topic_callback(const std_msgs::msg::String::SharedPtr msg) const
    {
      RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
    }
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};

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

Ok, wie lassen sich diese beiden Knoten nun starten. In dieser Veranstaltung wollen wir uns allein auf die vorinstallierten Beispiele, in diesem Fall die Pakete `examples_rclcppp_minimal_subscriber` und `examples_rclcppp_minimal_publisher` konzentrieren. Dazu starten wir diese jeweils mit

```
#ros2 run <package_name> <node>
ros2 run examples_rclcppp_minimal_subscriber subscriber_member_function
ros2 run examples_rclcppp_minimal_publisher publisher_member_function
```

Lassen Sie uns diese Konfiguration systematisch untersuchen:

1. Welche Nachrichten werden in unserem bescheidenen ROS2 System ausgetauscht?

```
>ros2 topic list
/parameter_events
/rosout
/topic
>ros2 topic info /topic
Topic: /topic
Publisher count: 1
Subscriber count: 1
>ros2 topic hz /topic
average rate: 2.011
  min: 0.489s max: 0.500s std dev: 0.00474s window: 4
...
```

2. Wie lassen sich mehrere Instanzen ein und des selben Knoten integrieren?

Es soll nochmals darauf hingewiesen werden, `topic` ist ein willkürlich gewählter Name für unseren Kanal. Um beim Testen von verschiedenen Nodes eine schnelle Umbennenung zu ermöglichen können wir mittels Remapping die Topic und Nodenamen anpassen.

```
> ros2 run examples_rclcpp_minimal_publisher publisher_member_function /top
  :=/topic2
```

```
>ros2 topic info /topic
Topic: /topic
Publisher count: 1
Subscriber count: 2
```

Screenshot des Tools `rgt_graph`

Natürlich können Sie auch den Topic-Namen aus der Kommandozeile anpassen. Damit entsteht ein neuer Kanal, der keine Subcriber hat.

```
> ros2 run examples_rclcpp_minimal_publisher publisher_member_function --ro
  --remap /topic:=/topic2
```

Screenshot des Tools `rgt_graph`

4. Kann ich auch einen Publisher in der Konsole erzeugen?

Natürlich, dies ist ein wichtiges Element des Debugging. Starten Sie also zum Beispiel den Subscriber mit den bereits bekannten Kommandos und führen Sie dann in einer anderen Konsole den nachfolgenden Befehl aus.

```
ros2 topic pub /topic std_msgs/String  "data: Glück Auf" -n TUBAF
```

Informieren Sie sich zudem über die weiteren Parameter von `ros2 topic pub`. Sie können die Taktrate und die Qualitätskritieren der Übertragung definieren.

Turtlebot

Das "turtlebot" Beispiel soll die verschiedenen Mechanismen der Kommunikation unter ROS verdeutlichen. Dabei wird unter anderem eine Publish-Subscribe Kommunikation zwischen einem Node für die Nutzereingaben und einer grafischen Ausgabe realisiert.

```
ros2 run turtlesim turtle_teleop_key
ros2 run turtlesim turtlesim_node
```

Screenshot des TurtleSim-Knotens

Wir wollen wiederum das System inspizieren und nutzen dafür ein grafisches Inspektionssystem, das in ROS2 integriert ist. Hier werden die Methoden, die `ros2` auf der Kommandozeile bereithält in einer GUI abgebildet.

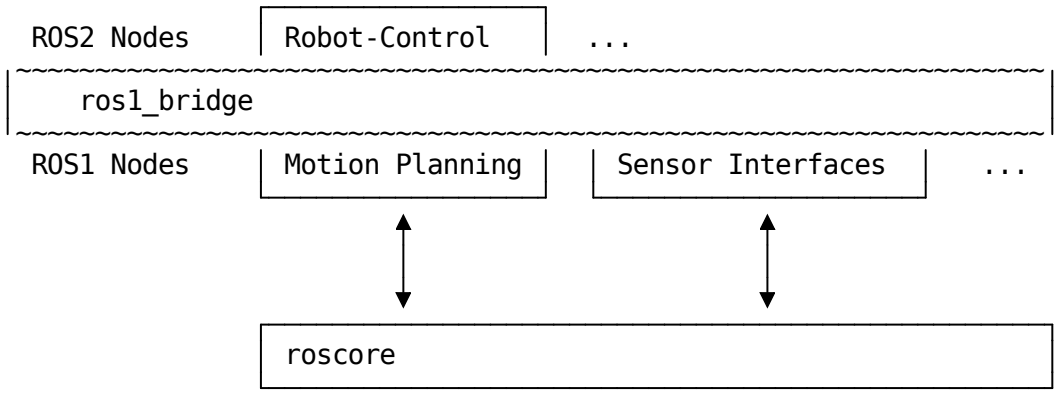
```
rqt
```

Screenshot des TurtleSim-Knotens

ros1_bridge

`ros1_bridge` ermöglicht die Kombination beider ROS Konzepte in übergreifenden Anwendungen. Dabei treffen die unterschiedlichen Kommunikationskonzepte aufeinander, so dass die in ROS2 etablierten Kommunikations-Qualitätskritieren nicht übergreifend realisiert werden können.

Allerdings ist diese Möglichkeit nur unter Linux und MacOS gegeben. In wieweit eine Interaktion zwischen verteilten Systemen möglich ist, wurde nicht evaluiert.



Eine gute Beschreibung findet sich unter Projektordner ros1_bridge. Das nachfolgende Beispiel greift diese Idee auf und liest auf der ROS1 Seite Bilddaten einer Kamera ein, detektiert Gesichter und gibt ein entsprechend angereichertes Bild unter ROS2 in rqt aus.

Wichtig ist, dass Sie die verschiedenen `setup.x` Aufrufe korrekt ausführen.

ConsoleA

```
# ROS1_Shell mit roscore
> . /opt/ros/noetic/setup.bash
> roscore
```

ConsoleB

```
# ROS1_bridge
> . /opt/ros/noetic/setup.bash
> . /opt/ros/foxy/setup.bash
> export ROS_MASTER_URI=http://localhost:11311
> ros2 run ros1_bridge dynamic_bridge
```

Auf der ROS2 Seite nutzen wir nun das image_tools Paket um unser Videosignal einzulesen.

ConsoleC

```
# ROS2_Shell die ein Kamerasignal erfasst
> . /opt/ros/foxy/setup.bash
> ros2 run image_tools cam2image
#ros2 run image_tools showimage   # zum testen
```

Im folgenden wird ein Knoten aus dem Paket opencv_app aktiviert, dass für die Erkennung der Gesichter verantwortlich ist. Diese ist bisher nicht für ROS2 verfügbar.

ConsoleD

```
# ROS1_Shell die ein Kamerasignal erfasst
> . /opt/ros/noetic/setup.bash
> rosrun opencv_apps face_detection image:=image
```

Nun können wir uns die Ausgabe auf der ROS1 Seite anschauen. Leider gibt es noch einen Fehler beim Zurückleiten des Bildes aus ROS1 nach ROS2, was das Bild komplett gemacht hätte. Der Typ des Bildes aus ROS1 war für ROS2 unbekannt. Damit verbleiben wir hier für die Anzeige auf der Seite von ROS1.

ConsoleE

```
# ROS1_Shell die die Ausgabe koordiniert
. /opt/ros/noetic/setup.bash
> rostopic list
/face_detection_1575145081472799322/face_image
/face_detection_1575145081472799322/faces
/face_detection_1575145081472799322/image
/face_detection_1575145081472799322/parameter_descriptions
/face_detection_1575145081472799322/parameter_updates
> rosrun image_view image_view image:=/face_detection_1575145081472799322/
```

Beispielhafte Ausgabe der erkannten Gesichter

Installation

1. ... auf einem Linux System ... folgen Sie den Anweisungen der Dokumentation
2. ... auf einem Windows 10 System nativ ... schwierig
3. ... auf einem Windows 10 System mit einer WSL2 Ubuntu-Installation

Aufgabe der Woche

1. Beschäftigen Sie sich mit den Kommandozeilentools Ihres Betriebssystems. Machen Sie sich mit dem Anlegen, Löschen, Rechtemanagement usw. von Dateien und Ordnern vertraut.

2. Machen Sie sich mit den Möglichkeiten des Kommandozeilen-Tools `ros2` vertraut. Wie können Sie Topics anzeigen, generieren, visualisieren? Eine Hilfe dazu kann zum Beispiel https://ubuntu.com/blog/ros-2-command-line-interface sein.