



# **WORKSHOP-TERMIN 2**

## Drohnenprogrammierung und Automatisiertes Fliegen

10.05.2022

*Patrik Golec*

# Organisatorisches I

---

- » 12.04.: VM & Python Basics
- » **10.05.: ROS & ROS Python**
- » 14.06.: CrazySwarm & Drohnenlabor
- » 06.07.: Automatisiertes Fliegen

# Agenda

- » Interaktive ROS-Einführung
- » ROS-Beispiel „turtlesim“

- » ROS Architektur
  - ROS Master, Nodes, Topics, Messages
  - Konsolenbefehle
  - CMake, Catkin workspace, build system
  - Launch-files
- » ROS packages
- » ROS Python Clientbibliothek (rospy)
- » ROS Subscribers & Publishers
- » ROS Parameters
- » ROS Services & Actions

# Was ist ROS?

- » Robot Operating System
- » Kein natives Betriebssystem wie etwa Linux
- » „Meta-Betriebssystem“, benötigt OS
- » Vielzahl von Middleware für Roboter und weitere Werkzeuge
  - „Unterbau“
  - bietet Kommunikationsschicht

# Was ist ROS?

- » grob 4 Komponenten
- » „**Unterbau**“
  - Prozessmanagement, Interprozesskommunikation, Treiber
- » **Werkzeuge, Tools**
  - Simulation, Visualisierung, Logging
- » **Algorithmen**
  - Control, Planning, Mapping
- » **Ecosystem**
  - Packageorganisation, Build Tools
  - Dokumentation, Tutorials

- » 2007 von Stanford AI Lab entwickelt
- » Seit 2013 von OSRF verwaltet
- » Standard für Roboterprogrammierung
  - breite Verwendung in Universitäten & Wirtschaft
  - “The Rise of ROS: Nearly 55% of total commercial robots shipped in 2024 Will Have at Least One Robot Operating System package Installed”
    - » <https://www.businesswire.com/news/home/20190516005135/en/The-Rise-of-ROS-Nearly-55-of-total-commercial-robots-shipped-in-2024-Will-Have-at-Least-One-Robot-Operating-System-package-Installed>

## » **Peer to peer**

- ROS-Programme kommunizieren über API (ROS messages, services, etc.)

## » **Verteilt (Verteiltes System)**

- ROS-Programme können über mehrere Maschinen verteilt laufen

## » **Multi-lingual**

- ROS Module können in beliebigen Programmiersprachen entwickelt werden, solange es eine Clientbibliothek gibt (C++, Python, Java, MATLAB, etc.)

## » **Leichtgewichtig**

- Wrapper für Bibliotheken

## » **Kostenlos und open-source**

- die meiste ROS-Software ist frei und quelloffen



- » bietet Namensgebungs- und Registrierungsservices für Nodes (usw.)
- » macht Kommunikation zwischen Nodes (Prozesse) möglich
  - hilft Nodes, einander zu lokalisieren und Verbindung aufzubauen
- » Kommunikation via XMLRPC
  - zustandloses, auf HTTP basierendes Protokoll
- » jede Node registriert sich beim Start im Master Node

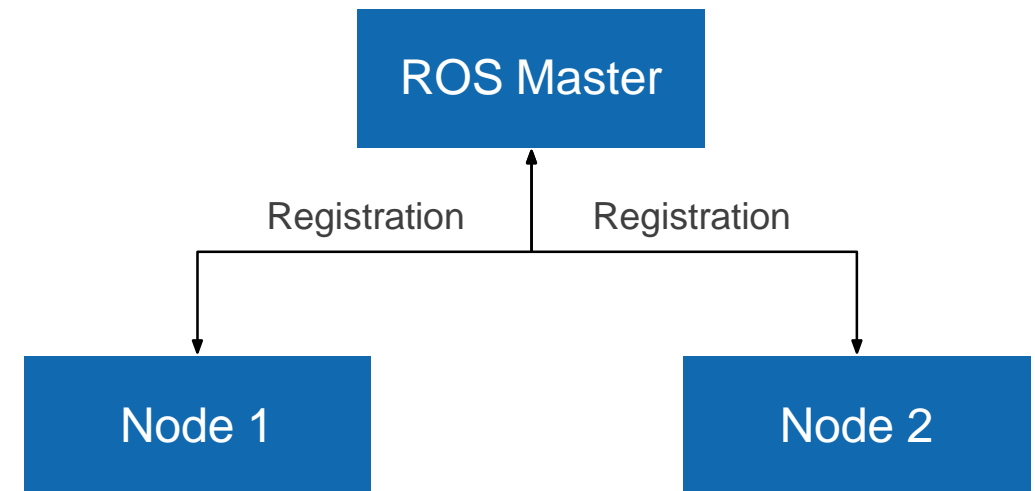
ROS Master

Einen master starten mit

```
> roscore
```

# ROS Nodes

- » ausführbares Programm, eigener Prozess, führt Berechnungen durch
- » hat einen einzigen Nutzen (Separation of Concerns)
- » unabhängig von anderen Nodes kompiliert, ausgeführt und verwaltet
- » werden in sog. Packages organisiert
- » jede Node hat eine URI
- » Kommunikation mit Master
  - XMLRPC-Protokoll
- » Inter-Node-Kommunikation peer-to-peer
  - TCP bzw. TCPROS



# ROS Nodes

Eine Node ausführen

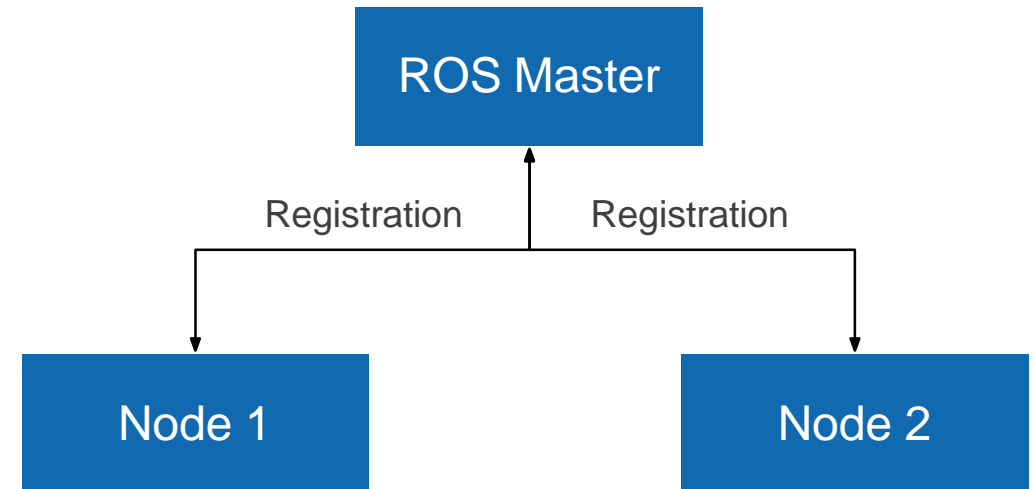
```
> rosrun package_name node_name
```

Liste aktiver Nodes einsehen

```
> rosnodet list
```

Informationen über eine Node abrufen

```
> rosnodet info node_name
```



- » Nodes kommunizieren über Topics
  - können Topic subscriben oder publishen
  - für gewöhnlich 1 Publisher, mehrere Subscriber
- » Topic ist ein kontinuierlicher Datenstrom:

Aktive Topics auflisten

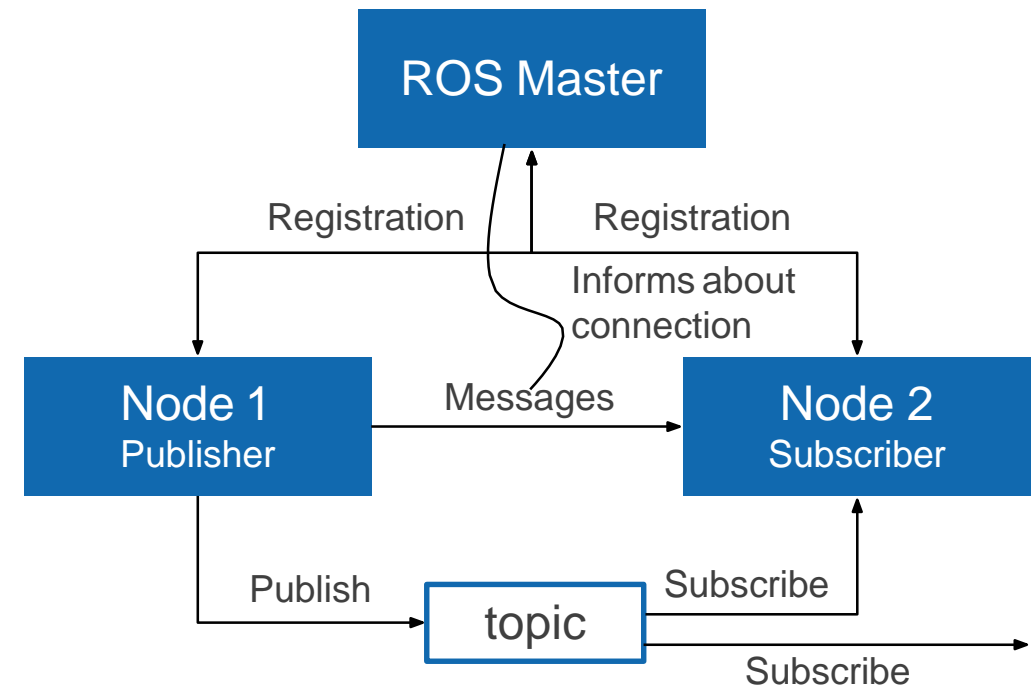
```
> rostopic list
```

Den Inhalt eines Topics ‚subscriben‘ und ausgeben

```
> rostopic echo /topic
```

Informationen über ein Topic ausgeben

```
> rostopic info /topic
```



# ROS Messages

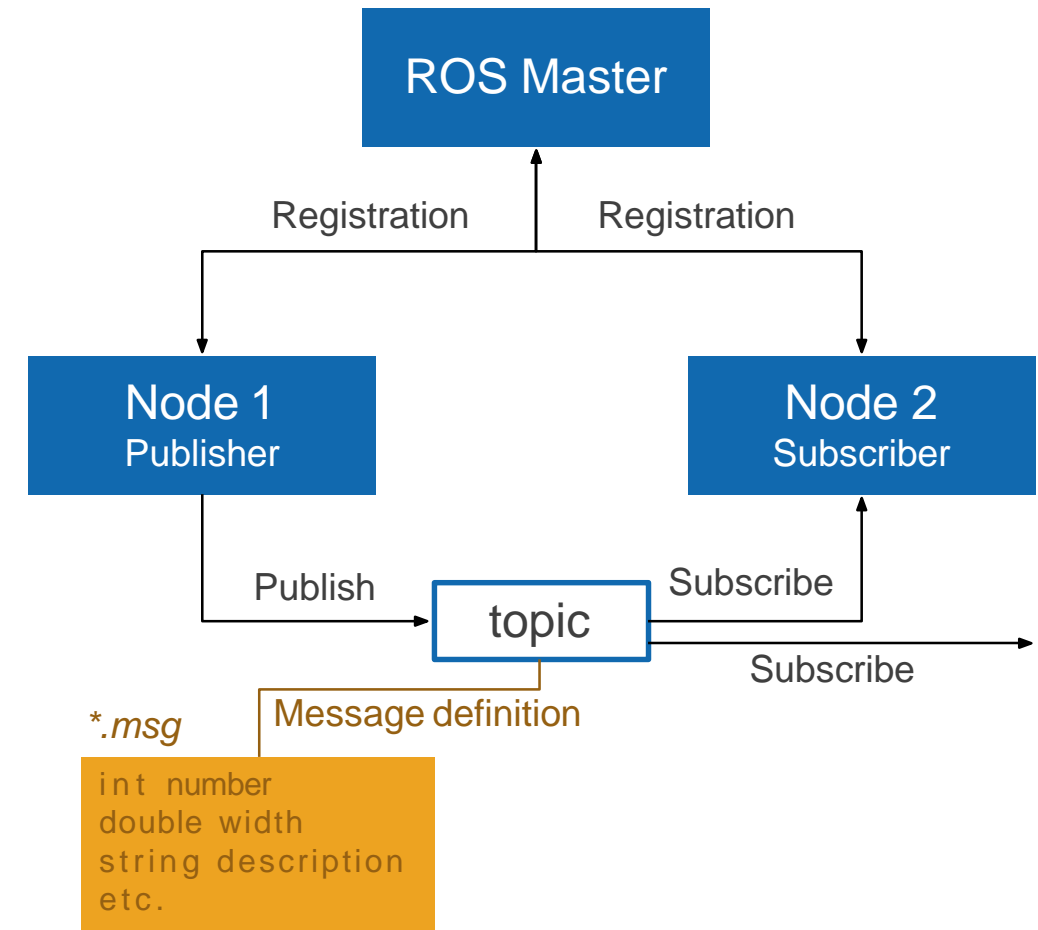
- » werden innerhalb von Topics von Nodes untereinander versendet
- » Datenstruktur definiert Typ eines Topics
- » Verschachtelte Strukturen von Integern, Floats, Booleans, Strings usw.
- » definiert über *\*.msg* Dateien

Typ eines Topics sehen

```
> rostopic type /topic
```

Eine Message in einem Topic veröffentlichen

```
> rostopic pub /topic type data
```



# ROS Messages

## PoseStamped Beispiel

[geometry\\_msgs/Point.msg](#)

```
float64 x  
float64 y  
float64 z
```

[sensor\\_msgs/Image.msg](#)

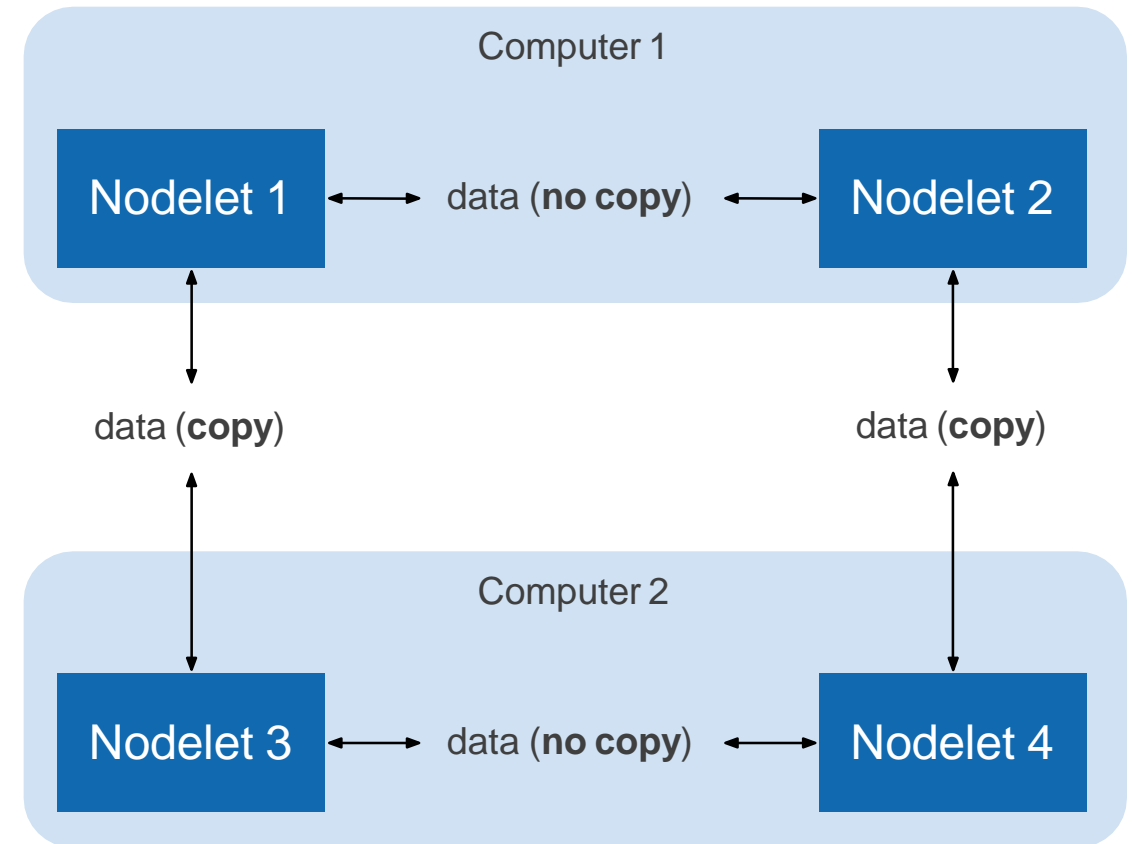
```
std_msgs/Header header  
  uint32 seq  
  time stamp  
  string frame_id  
uint32 height  
uint32 width  
string encoding  
uint8 is_bigendian  
uint32 step  
uint8[] data
```

[geometry\\_msgs/PoseStamped.msg](#)

```
std_msgs/Header header  
uint32 seq  
time stamp  
string frame_id  
geometry_msgs/Pose pose  
  geometry_msgs/Point position  
    float64 x  
    float64 y  
    float64 z  
  geometry_msgs/Quaternion orientation  
    float64 x  
    float64 y  
    float64 z  
    float64 w
```

# ROS Nodelets

- » Gleiches Konzept wie ROS-nodes
- » Reduzieren den Kommunikations-Overhead, wenn sie auf demselben Rechner laufen
- » ROS-Nodes bevorzugen, weil Nodelets komplizierter zu implementieren



# Beispiel

## Konsole – *roscore* starten

Roscore starten mit

```
> roscore
```

```
drohne@drohne-VirtualBox:~$ roscore
... logging to /home/drohne/.ros/log/3ed04f30-caae-11ec-abd8-cb1ba44872f6
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://drohne-VirtualBox:34235/
ros_comm version 1.15.14

SUMMARY
=====

PARAMETERS
* /roscdistro: noetic
* /rosversion: 1.15.14

NODES

auto-starting new master
process[master]: started with pid [2975]
ROS_MASTER_URI=http://drohne-VirtualBox:11311/

setting /run_id to 3ed04f30-caae-11ec-abd8-cb1ba44872f6
process[rosout-1]: started with pid [2986]
started core service [/rosout]
```



# Beispiel

## Konsole – *talker* node starten

Eine talker demo node starten

```
> rosrun roscpp_tutorials talker
```

```
drohne@drohne-VirtualBox:~$ rosrun roscpp_tutorials talker
[ INFO] [1651561128.274825605]: hello world 0
[ INFO] [1651561128.375905989]: hello world 1
[ INFO] [1651561128.475303927]: hello world 2
[ INFO] [1651561128.575356591]: hello world 3
[ INFO] [1651561128.676556631]: hello world 4
[ INFO] [1651561128.776673786]: hello world 5
[ INFO] [1651561128.875935481]: hello world 6
[ INFO] [1651561128.975894268]: hello world 7
[ INFO] [1651561129.074905002]: hello world 8
[ INFO] [1651561129.175551523]: hello world 9
[ INFO] [1651561129.276780081]: hello world 10
[ INFO] [1651561129.376652581]: hello world 11
[ INFO] [1651561129.475940848]: hello world 12
[ INFO] [1651561129.575399622]: hello world 13
[ INFO] [1651561129.676350217]: hello world 14
[ INFO] [1651561129.777578286]: hello world 15
[ INFO] [1651561129.875524695]: hello world 16
[ INFO] [1651561129.976112974]: hello world 17
[ INFO] [1651561130.075571905]: hello world 18
[ INFO] [1651561130.176532261]: hello world 19
[ INFO] [1651561130.275824251]: hello world 20
[ INFO] [1651561130.376943124]: hello world 21
[ INFO] [1651561130.478528725]: hello world 22
[ INFO] [1651561130.576835832]: hello world 23
```

# Beispiel

## Konsole – *talker* node analysieren

Liste aktiver Nodes einsehen

```
> rosnod list
```

```
drohne@drohne-VirtualBox:~$ rosnod list
/rosout
/talker
drohne@drohne-VirtualBox:~$
```

Informationen über talker node ausgeben

```
> rosnod info /talker
```

```
drohne@drohne-VirtualBox:~$ rosnod info /talker
-----
Node [/talker]
Publications:
* /chatter [std_msgs/String]
* /rosout [rosgaph_msgs/Log]

Subscriptions: None

Services:
* /talker/get_loggers
* /talker/set_logger_level

contacting node http://drohne-VirtualBox:38165/ ...
Pid: 3194
Connections:
* topic: /rosout
* to: /rosout
* direction: outbound (59399 - 127.0.0.1:42968) [11]
* transport: TCPROS
```

Informationen über ein *chatter* topic einsehen

```
> rostopic info /chatter
```

```
drohne@drohne-VirtualBox:~$ rostopic info /chatter
Type: std_msgs/String

Publishers:
* /talker (http://drohne-VirtualBox:38165/)

Subscribers: None
```

# Beispiel

## Konsole – *talker* node analysieren

Typ/Art eines *chatter* topic überprüfen

```
> rostopic type /chatter
```

```
drohne@drohne-VirtualBox:~$ rostopic type /chatter  
std_msgs/String
```

Message Inhalt eines topic ausgeben

```
> rostopic echo /chatter
```

```
drohne@drohne-VirtualBox:~$ rostopic echo /chatter  
data: "hello world 2912"  
---  
data: "hello world 2913"  
---  
data: "hello world 2914"
```

Frequenz analysieren

```
> rostopic hz /chatter
```

```
drohne@drohne-VirtualBox:~$ rostopic hz /chatter  
subscribed to [/chatter]  
average rate: 10.027  
  min: 0.098s max: 0.101s std dev: 0.00074s window: 10  
average rate: 10.007  
  min: 0.098s max: 0.101s std dev: 0.00074s window: 20  
average rate: 10.006  
  min: 0.098s max: 0.101s std dev: 0.00084s window: 30  
average rate: 10.002  
  min: 0.096s max: 0.102s std dev: 0.00109s window: 40  
average rate: 10.003  
  min: 0.096s max: 0.102s std dev: 0.00102s window: 50  
average rate: 10.003  
  min: 0.096s max: 0.102s std dev: 0.00098s window: 60
```

# Beispiel

## Konsole – *listener* node starten

Eine listener demo node starten

```
> rosrun roscpp_tutorials listener
```

```
drohne@drohne-VirtualBox:~$ rosrun roscpp_tutorials listener
[ INFO] [1651561601.096045031]: I heard: [hello world 4073]
[ INFO] [1651561601.196312793]: I heard: [hello world 4074]
[ INFO] [1651561601.297132682]: I heard: [hello world 4075]
[ INFO] [1651561601.396541589]: I heard: [hello world 4076]
[ INFO] [1651561601.498762646]: I heard: [hello world 4077]
```

# Beispiel

## Konsole – *listener* node analysieren

Neue listener node einsehen mit

```
> rosnod list
```

```
drohne@drohne-VirtualBox:~$ rosnod list
/listener
/rosout
/talker
```

Verbindung der Knoten über das chatter topic zeigen

```
> rostopic info /chatter
```

```
drohne@drohne-VirtualBox:~$ rostopic info chatter
Type: std_msgs/String

Publishers:
* /talker (http://drohne-VirtualBox:38165/)

Subscribers:
* /listener (http://drohne-VirtualBox:41181/)
```

# Beispiel

## Konsole – Message von der Konsole aus veröffentlichen

Talker node über Konsole schließen mit Strg + C

Eigene message veröffentlichen

```
> rostopic pub /chatter  
std_msgs/String "data: 'FH  
Kufstein ROS Workshop'"
```

```
drohne@drohne-VirtualBox:~$ rostopic pub /chatter std_msgs/String "data: 'FH Kufstein  
ROS Workshop'"  
publishing and latching message. Press ctrl-C to terminate
```

Ausgabe des listener überprüfen

```
[ INFO] [1651561803.502450418]: I heard: [hello world 6090]  
[ INFO] [1651561803.602195010]: I heard: [hello world 6091]  
[ INFO] [1651561803.702842607]: I heard: [hello world 6092]  
[ INFO] [1651561803.803681128]: I heard: [hello world 6093]  
[ INFO] [1651561803.902248692]: I heard: [hello world 6094]  
[ INFO] [1651561809.644642854]: I heard: [FH Kufstein ROS Workshop]
```

# ROS Arbeitsbereich Umgebung

- » Definiert den Kontext für den aktuellen Arbeitsbereich
- » Standard-Arbeitsbereich geladen mit


```
> source /opt/ros/noetic/setup.bash
```

Catkin-Arbeitsbereich überlagern

```
> cd ~/catkin_ws  
> source devel/setup.bash
```

Arbeitsbereich überprüfen

```
> echo $ROS_PACKAGE_PATH
```



Muss fürs  
Beispiel  
eingerrichtet  
werden!

Installation einsehen

```
> cat ~/.bashrc
```

# catkin Build System

- » catkin ist das ROS-Build-System zur Erzeugung von ausführbaren Dateien, Bibliotheken und Schnittstellen
- » Die Verwendung der Catkin Command Line Tools über catkin\_Make wird empfohlen

→ **catkin build** anstelle von catkin\_make verwenden, da neuer und komfortabler

Die catkin-Kommandozeilen-Tools müssen an dieser Stelle installiert werden.

- » Anleitung:
  - <https://catkin-tools.readthedocs.io/en/latest/installing.html>



» Der catkin-Arbeitsbereich enthält die folgenden Bereiche

Hier arbeiten



Der Quellbereich (*source space*) enthält den Quellcode. Hier kann man den Quellcode für die Pakete, die man erstellen möchte, klonen, erstellen und bearbeiten.

Nicht berühren



Im *build*-Bereich wird CMake aufgerufen, um die Pakete im Quellbereich zu bauen. Cache-Informationen und andere Zwischendateien werden hier aufbewahrt.

Nicht berühren



Der Entwicklungsbereich (*development space - devel*) ist der Bereich, in dem die gebauten Pakete platziert werden (bevor sie installiert werden).

Bei Bedarf den gesamten Build- und Devel-Bereich bereinigen

```
> catkin clean
```

# catkin Build System

Test-Ordner erstellen und hinnavigieren

```
> mkdir -p ~/Documents/projects/ros_tutorial/catkin_ws/src &&  
cd ~/Documents/projects/ros_tutorial/catkin_ws
```

Workspace initialisieren

```
> catkin init
```

Zum *source space* navigieren

```
> cd src
```

Paket erstellen

```
> catkin create pkg package_name
```

Builden

```
> catkin build
```

Wann immer gebuildet oder ein neues Paket erstellt wird,

**Umgebung aktualisieren**

```
> source devel/setup.bash
```

```
Build Space:      [missing]  
Devel Space:     [missing]  
Install Space:   [unused]  
Log Space:       [missing]  
Source Space:    [exists]  
DESTDIR:         [unused]
```

» [https://catkin-tools.readthedocs.io/en/latest/quick\\_start.html](https://catkin-tools.readthedocs.io/en/latest/quick_start.html)

# catkin Build System

Einrichtung des catkin-Arbeitsbereichs überprüfen

```
> catkin config
```

Um zum Beispiel den CMake-Build-Typ auf Release (oder Debug usw.) zu setzen

```
> catkin build --cmake-args  
-DCMAKE_BUILD_TYPE=Release
```

```
Profile: default
Extending: [cached] /home/drohne/Documents/projects/crazyswarm/ros_ws/devel:/opt/ros/kinetic/devel
Workspace: /home/drohne/Documents/projects/ros_test/catkin_ws

-----
Build Space: [exists] /home/drohne/Documents/projects/ros_test/catkin_ws/build
Devel Space: [exists] /home/drohne/Documents/projects/ros_test/catkin_ws/devel
Install Space: [unused] /home/drohne/Documents/projects/ros_test/catkin_ws/install
Log Space: [exists] /home/drohne/Documents/projects/ros_test/catkin_ws/logs
Source Space: [exists] /home/drohne/Documents/projects/ros_test/catkin_ws/src
DESTDIR: [unused] None

-----
Devel Space Layout: linked
Install Space Layout: None

-----
Additional CMake Args: None
Additional Make Args: None
Additional catkin Make Args: None
Internal Make Job Server: True
Cache Job Environments: False

-----
Buildlisted Packages: None
Skiplist Packages: None

-----
Workspace configuration appears valid.
```

# ROS Launch

- » roslaunch ist ein Werkzeug zum Starten mehrerer Nodes (sowie zum Setzen von Parametern)
- » werden in XML als \*.launch-Dateien geschrieben
- » wenn Master noch nicht läuft, startet roslaunch automatisch einen roscore

Zum Ordner navigieren und Launch-Datei starten

```
> roslaunch file_name.launch
```

Launch-Datei aus einem Paket starten

```
> roslaunch package_name  
file_name.launch
```

Beispiel einer Konsolenausgabe für

```
roslaunch roscpp_tutorials talker_listener.launch
```

```
drohne@drohne-VirtualBox:~$ roslaunch roscpp_tutorials talker_listener.launch
... logging to /home/drohne/.ros/log/d344484a-cab5-11ec-abd8-cb1ba44872f6/roslaun
.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://drohne-VirtualBox:38993/

SUMMARY
=====

PARAMETERS
* /roscpp_tutorials: noetic
* /roscpp_tutorials: 1.15.14

NODES
/
  listener (roscpp_tutorials/listener)
  talker (roscpp_tutorials/talker)

auto-starting new master
process[master]: started with pid [22023]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to d344484a-cab5-11ec-abd8-cb1ba44872f6
process[rosout-1]: started with pid [22034]
started core service [/rosout]
process[listener-2]: started with pid [22037]
process[talker-3]: started with pid [22038]
[ INFO] [1651564283.071712240]: hello world 0
[ INFO] [1651564283.172109663]: hello world 1
[ INFO] [1651564283.272497478]: hello world 2
```

! Achtung beim Kopieren und Einfügen von Code aus dem Internet

[talker\\_listener.launch](#)

```
<launch>  
  <node name="listener" pkg="roscpp_tutorials" type="listener" output="screen"/>  
  <node name="talker" pkg="roscpp_tutorials" type="talker" output="screen"/>  
</launch>
```

! Beachten Sie den Unterschied in der Syntax für selbstschließende Tags:  
<tag></tag> and <tag/>

- **launch:** Root-Element der Launch-Datei
- **node:** Jeder *<node>* Tag gibt eine zu startende Node an
- **name:** Name der Node (frei wählbar)
- **pkg:** Paket, das den Knoten enthält
- **type:** Typ der Node, es muss eine entsprechende ausführbare Datei mit demselben Namen geben
- **output:** Gibt an, wo die Protokollmeldungen ausgegeben werden sollen (screen: Konsole, log: Protokolldatei)

# ROS Launch Argumente

Erstellen von wiederverwendbaren Launch-Dateien mit `<arg>` Tag, der wie ein Parameter funktioniert (standardmäßig optional)

```
<arg name="arg_name"
      default="default_value"/>
```

Argumente im Launch-File verwenden

```
$(arg arg_name)
```

Beim Starten können Argumente gesetzt werden

```
> roslaunch launch_file.launch
   arg_name:=value
```

*range\_world.launch*

```
<?xml version="1.0"?>
<launch>
  <arg name="use_sim_time" default="true"/>
  <arg name="world" default="gazebo_ros_range"/>
  <arg name="debug" default="false"/>
  <arg name="physics" default="ode"/>

  <group if="$(arg use_sim_time)">
    <param name="/use_sim_time" value="true" />
  </group>

  <include file="$(find gazebo_ros)
              /launch/empty_world.launch">
    <arg name="world_name" value="$(find gazebo_plugins)/
                                test/test_worlds/$(arg world).world"/>
    <arg name="debug" value="$(arg debug)"/>
    <arg name="physics" value="$(arg physics)"/>
  </include>
</launch>
```

# ROS Launch

## Andere Launch-Dateien einbinden

Einbindung anderer Startdateien mit dem  
<include>-Tag zur Organisation großer  
Projekte

```
<include file="package_name" />
```

Den Systempfad zu anderen Paketen finden

```
$(find package_name)
```

Übergabe von Argumenten an die  
eingebundene Datei

```
<arg name="arg_name"  
value="value" />
```

*range\_world.launch*

```
<?xml version="1.0"?>  
<launch>  
  <arg name="use_sim_time" default="true"/>  
  <arg name="world" default="gazebo_ros_range"/>  
  <arg name="debug" default="false"/>  
  <arg name="physics" default="ode"/>  
  
  <group if="$(arg use_sim_time)">  
    <param name="/use_sim_time" value="true" />  
  </group>  
  
  <include file="$(find gazebo_ros)  
                                /launch/empty_world.launch">  
    <arg name="world_name" value="$(find gazebo_plugins)/  
                                test/test_worlds/$(arg world).world"/>  
    <arg name="debug" value="$(arg debug)"/>  
    <arg name="physics" value="$(arg physics)"/>  
  </include>  
</launch>
```

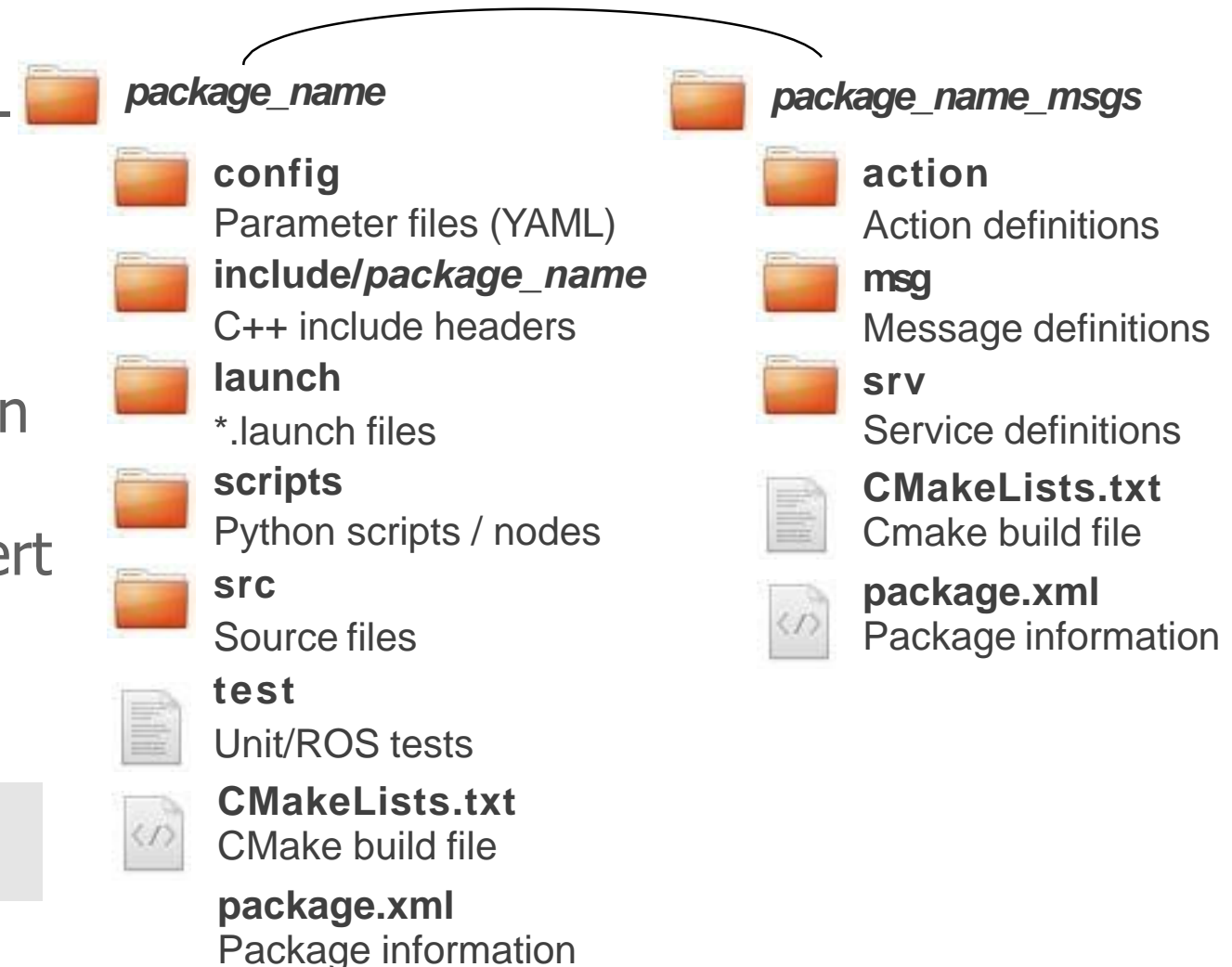
# ROS Packages

Definitionen-Pakete von Messages  
von anderen Paketen trennen!

- » ROS-Software ist in Paketen organisiert, die Quellcode, Launch-Dateien, Konfigurationsdateien, Message-Definitionen, Daten und Dokumentation enthalten können
- » Ein Paket, das auf anderen Paketen aufbaut bzw. diese benötigt (z. B. Nachrichten-Definitionen), deklariert diese als Abhängigkeiten

Ein neues Paket erstellen

```
catkin create pkg package_name  
{dependencies}
```





# ROS Packages

## package.xml

» Die Datei package.xml definiert die Eigenschaften des Pakets

- Name des Pakets
- Versionsnummer
- Verfasser
- **Abhängigkeiten von anderen Paketen**
- ...

### package.xml

```
<?xml version="1.0"?>
<package>
  <name>crazyswarm</name>
  <version>0.0.3</version>
  <description>Crazyswarm: A swarm of Crazyflie-based quadrotors</description>

  <maintainer email="whoenig@usc.edu">Wolfgang Hoenig</maintainer>

  <license>MIT</license>

  <url type="website">http://crazyswarm.readthedocs.io</url>
  <url type="bugtracker">https://github.com/USC-ACTLab/crazyswarm/issues</url>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>message_generation</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>tf</build_depend>
  <build_depend>crazyflie_cpp</build_depend>
  <build_depend>libobjecttracker</build_depend>
  ...

  <run_depend>message_runtime</run_depend>
  <run_depend>std_msgs</run_depend>
  <run_depend>tf</run_depend>
  ...
  <run_depend>rospy</run_depend>
</package>
```

# ROS Packages

## CMake Tool

- » open-source, cross-platform und freies Werkzeug
- » Meta build system
- » kontrolliert Buildprozess
- » verwendet Skripts, nämlich CMakeLists
- » generiert damit *build files*, *Makefile* für spezifische Umgebungen
- » unterstützt versch. Build- und Betriebssysteme
- » unterstützt mehrere Programmiersprachen wie C++, C#

» Die CMakeLists.txt ist eine Datei mit Anweisungen für das CMake-Build-System

1. Erforderliche CMake-Version (cmake\_minimum\_required)
2. Paketname (project())
3. C++ Standard und Kompilierfunktionen konfigurieren
4. Andere CMake/Catkin-Pakete finden, die für die Erstellung benötigt werden (find\_package())
5. Generierung von Messages/Services/Actions (add\_message\_files(), add\_service\_files(), add\_action\_files())
6. Aufrufen der Nachrichten-/Dienst-/Aktionsgenerierung (generate\_messages())
7. Spezifizieren des Exports von Paket-Bauinformationen (catkin\_package())
8. Zu erstellende Bibliotheken/Ausführbare Dateien (add\_library()/add\_executable()/target\_link\_libraries())
9. Zu erstellende Tests (catkin\_add\_gtest())
10. Regeln installieren (install())

### *CMakeLists.txt*

```
cmake_minimum_required (VERSION 2.8.3)
project (crazyswarm)

## Use C++14, or 11...
set (CMAKE_CXX_STANDARD 14 )
set (CMAKE_CXX_STANDARD_REQUIRED TRUE )

## Find catkin macros and libraries
find_package (catkin REQUIRED
              COMPONENTS
              roscpp
              sensor_msgs
              )

...
```

- » Python-Scripts u. Abhängigkeiten benötigen auch package.xml, CMakeLists, Makefile

```
> catkin create pkg tutorial_pkg --catkin-deps rospy
```

### *CMakeLists.txt*

```
cmake_minimum_required (VERSION 3.0.2)
project (my_pkg)

## Find catkin macros and libraries
find_package(catkin REQUIRED COMPONENTS
             rospy)

catkin_package()

## catkin_python_setup()
```

# ROS Packages

## Python Scripts, Module und Abhängigkeiten

- » Innerhalb von *tutorial\_pkg* die Ordner
  - *bin*
  - *src/py\_package*
- » und in *py\_package* die Datei *\_\_init\_\_.py* erstellen

```
> cd ~/Documents/projects/ros_tutorial/catkin_ws/tutorial_pkg  
> mkdir bin  
> mkdir src  
> mkdir src/py_package  
> touch src/py_package/__init__.py
```

- » Anmerkung: normalerweise heißt das python package exakt gleich wie das Parent-Package

# ROS Packages

## Python Scripts, Module und Abhängigkeiten

» Innerhalb von *src/py\_package* die Datei

○ *hello.py* erstellen

» *hello.py* Inhalt:

```
def say(name):  
    print('Hello ' + name)
```

» Innerhalb von *bin* die Datei

○ *hello* erstellen

» *hello* Inhalt:

```
import py_package.hello  
  
if __name__ == '__main__':  
    py_package.hello.say('my friend!')
```

# ROS Packages

## setup.py

- » Scripts müssen in PYTHONPATH verfügbar sein
- » über setup.py
- » CMake macro
- » Nicht für manuelle Ausführung, sondern durch Catkin!

```
## ! DO NOT MANUALLY INVOKE THIS setup.py, USE CATKIN INSTEAD

from distutils.core import setup
from catkin_pkg.python_setup import generate_distutils_setup

# fetch values from package.xml
setup_args = generate_distutils_setup(
    packages=['py_package'],
    scripts=['bin/hello'],
    package_dir={'': 'src'},
)

setup(**setup_args)
```

# ROS Packages

## Python Scripts, Module und Abhängigkeiten

- » catkin\_python\_setup() hinzufügen
- » Installationsanweisung hinzufügen

### CMakeLists.txt

```
cmake_minimum_required (VERSION 3.0.2)
project (tutorial_pkg)

## Find catkin macros and libraries
find_package(catkin REQUIRED COMPONENTS
    rospy)

## Uncomment this if the package has a setup.py. This macro ensures
## modules and global scripts declared therein get installed
## See http://ros.org/doc/api/catkin/html/user\_guide/setup\_dot\_py.html
catkin_python_setup()

catkin_install_python(PROGRAMS bin/hello
    DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```



# ROS Packages

## Python Scripts, Module und Abhängigkeiten

### » Package builden

```
> cd ~/Documents/projects/ros_tutorial/catkin_ws  
> catkin build  
> source devel/setup.bash
```

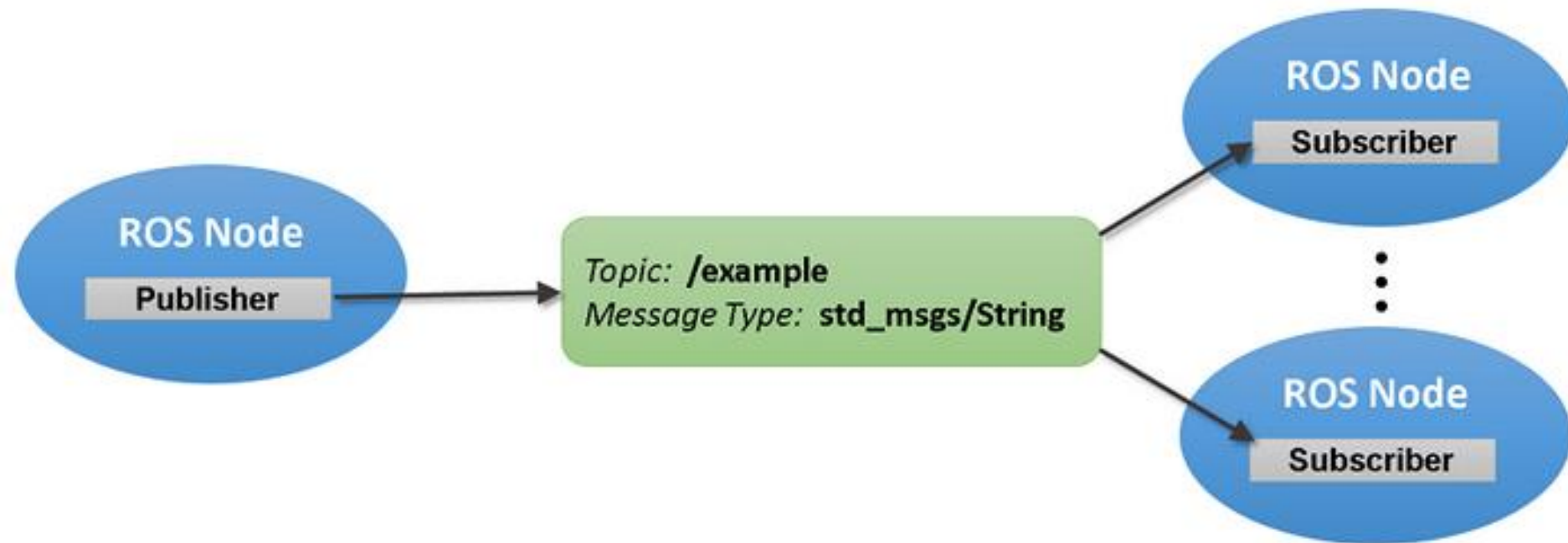
```
> rosrun tutorial_pkg hello  
Hello my friend!
```

# ROS Python Client Library (rospy)

- » Wesentliche Komponenten der Client-Bibliothek
  - Publisher / Subscriber
  - Parameter
  - Logging
  - (Services und Actions)

- » Message passing, Nachrichtenweitergabe
- » ROS Publisher/Subscriber Interface
- » Publisher veröffentlicht vordefinierte Messages auf ein Topic
- » Subscriber abonniert das Topic und erhält Nachrichten, wenn diese veröffentlicht werden
- » Publisher kann mehrere Topics veröffentlichen, Subscriber kann mehrere abonnieren

# Publisher / Subscriber



# rospy Publisher

```
>> import rospy
>> from std_msgs.msg import String

>> def talker():
>>     pub = rospy.Publisher('chatter', String, queue_size=10)
>>     rospy.init_node('talker', anonymous=True)
>>     rate = rospy.Rate(10) # 10hz
>>     while not rospy.is_shutdown():
>>         hello_str = 'hello world %s' % rospy.get_time()
>>         rospy.loginfo(hello_str)
>>         pub.publish(hello_str)
>>         rate.sleep()

>> if __name__ == '__main__':
>>     try:
>>         talker()
>>     except rospy.ROSInterruptException:
>>         pass
```

```
> chmod +x talker.py
```

# rospy Subscriber

```
>> import rospy
>> from std_msgs.msg import String

>> def callback(data):
    ○ rospy.loginfo(rospy.get_caller_id() + 'I heard %s', data.data)

>> def listener():
    ○ rospy.init_node('listener', anonymous=True)

>>     rospy.Subscriber('chatter', String, callback)

    rospy.spin()

>> if __name__ == '__main__':
    listener()
```

```
> chmod +x listener.py
```

# Python Nodes bauen

- » auch Python Nodes müssen mit CMake gebaut werden
- » wegen Autogenerierung des Codes für Messages und Services

```
> cd ~/Documents/projects/ros_tutorial/catkin_ws  
> catkin build  
> source devel/setup.bash
```

# Python Nodes ausführen

» für Ausführung wird ein ROS Master / roscore benötigt

```
> roscore
```

```
> rosrun ros_tutorial talker.py
```

```
> rosrun ros_tutorial listener.py
```



# Parameter / Parameter Server

- » Konfigurationsinformationen festlegen, globale Variablen
- » Integers, Floats, Strings, boolsche Werte, Datum, Listen
- » Setzen in Launchfiles
- » args, params, rosparams
- » Namensgebung wie bei Topics/Messages
- » Parameter Server
  - Ansammlung von Werten/Parametern
  - geteiltes Dictionary
  - kann über Terminal, Nodes ausgelesen werden

# Parameter abrufen

## » Terminal:

```
> rosparam list
```

```
> rosparam get logNode/enabled
```

## » rospy:

```
> import rospy  
  
> rospy.param_get(parameter_name)
```

# Parameter setzen

» Launchfile: *hover\_swarm.launch*

```
<?xml version="1.0"?>
<launch>
  <node pkg="crazyswarm" type="logNode.py" name="logNode"
output="screen">
    <param name="enabled" value="1" />
  </node>
  <rosparam>
    # Logging configuration (Use enable_logging to actually enable
logging)
    genericLogTopics: ["log1"]
    genericLogTopicFrequencies: [10]
    genericLogTopic_log1_Variables: []
    # firmware parameters for all drones (
firmwareParams:
    commander:
      enHighLevel: 1
    stabilizer:
      estimator: 2 # 1: complementary, 2: kalman
      controller: 2 # 1: PID, 2: mellinger
    ....
  </rosparam>
</launch>
```

# Parameter setzen

» rospy:

```
» import rospy  
  
» rospy.param_set(parameter_name, input_value)
```

# ROS Python Client Library (rospy) Logging

- » Mechanismus zur Protokollierung von menschenlesbarem Text von Nodes in der Konsole und in Protokolldateien
- » Anstelle von print z.B. *loginfo* verwenden
- » Automatische Protokollierung auf der **Konsole**, in der **Logdatei** und im/**rosout-topic**
- » Verschiedene Schweregrade (INFO, WARN, etc.)

```
rospy.logdebug(msg)
rospy.logwarn(msg)
rospy.loginfo(msg)
rospy.logerr(msg)
rospy.logfatal(msg)
```

	Debug	Info	Warn	Error	Fatal
stdout	✓	✓			
stderr			✓	✓	✓
Log file	✓	✓	✓	✓	✓
/rosout	✓	✓	✓	✓	✓

- ! Um die Ausgabe auf der Konsole zu sehen, die Ausgabekonfiguration in der Launch-Datei auf Bildschirm setzen

```
<launch>
  <node name="listener" ... output="screen"/>
</launch>
```

# ROS Services

- » Die Anfrage/Antwort-Kommunikation zwischen Nodes wird mit Diensten realisiert
  - Der Diensteserver kündigt den Dienst an
  - Der Dienstclient greift auf diesen Dienst zu
- » Ähnlich der Struktur von Nachrichten werden Dienste in \*.srv-Dateien definiert

Verfügbare Dienste auflisten

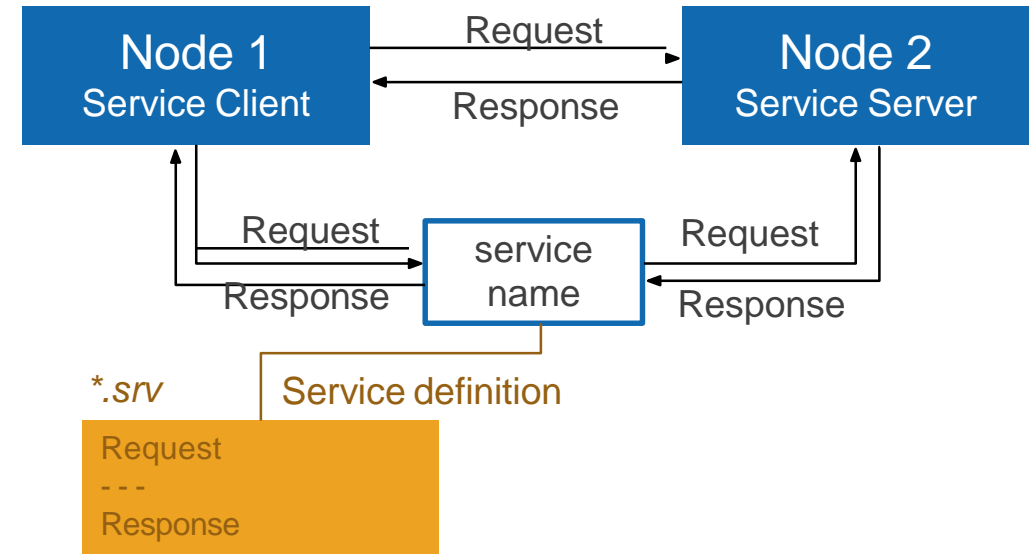
```
> rosservice list
```

Den Typ eines Dienstes anzeigen

```
> rosservice type /service_name
```

Aufruf eines Dienstes mit dem Inhalt der Anfrage, Autovervollständigung mit Tab

```
> rosservice call /service_name args
```



# ROS Services

## Beispiele

[std\\_srvs/Trigger.srv](#)

```
---  
bool success  
string message
```

Request

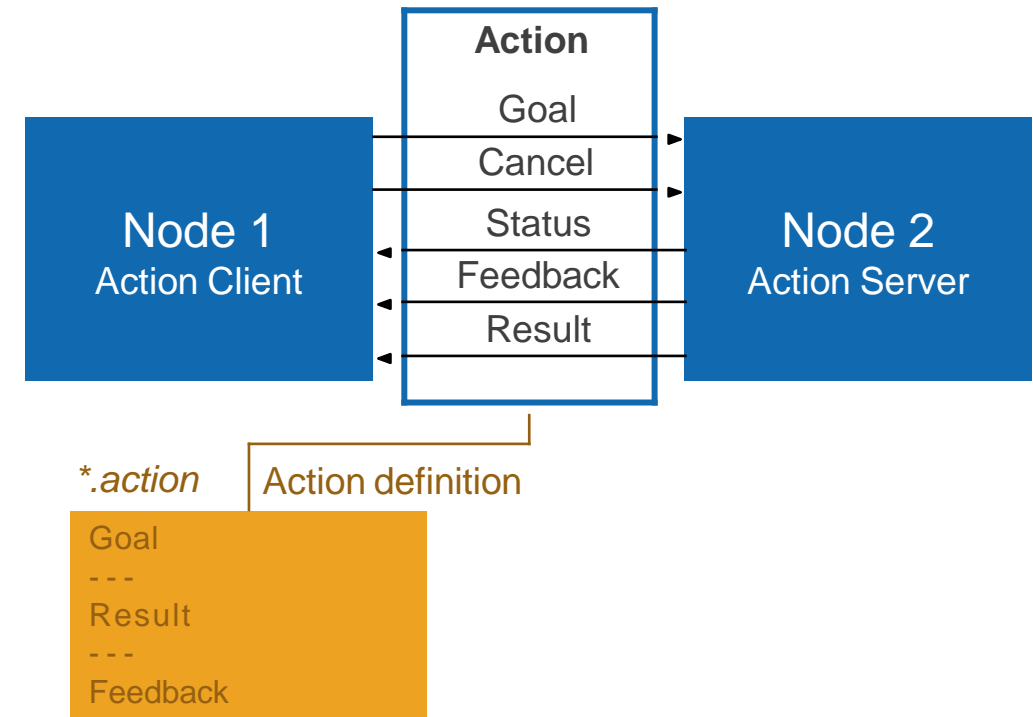
Response

[nav\\_msgs/GetPlan.srv](#)

```
geometry_msgs/PoseStamped start  
geometry_msgs/PoseStamped goal  
float32 tolerance  
---  
nav_msgs/Path plan
```

# ROS Actions (actionlib)

- » Ähnlich wie Dienstauftrufe (service calls), aber mit der Möglichkeit
  - die Aufgabe abubrechen (Preempt)
  - eine Rückmeldung über den Fortschritt zu erhalten
- » Bester Weg zur Implementierung von Schnittstellen zu zeitlich verlängerten, zielorientierten Verhaltensweisen
- » Ähnlich strukturiert wie Dienste, werden Actions in \*.action-Dateien definiert
- » Intern werden Actions mit einer Reihe von Themen implementiert

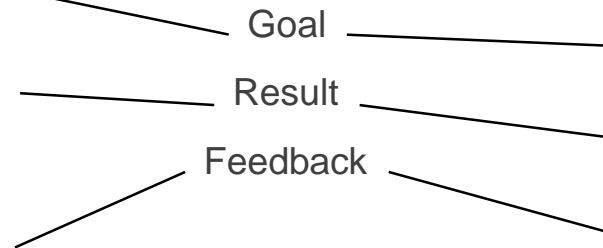




# ROS Actions (actionlib)

## Averaging.action

```
int32 samples
---
float32 mean
float32 std_dev
---
int32 sample
float32 data
float32 mean
float32 std_dev
```



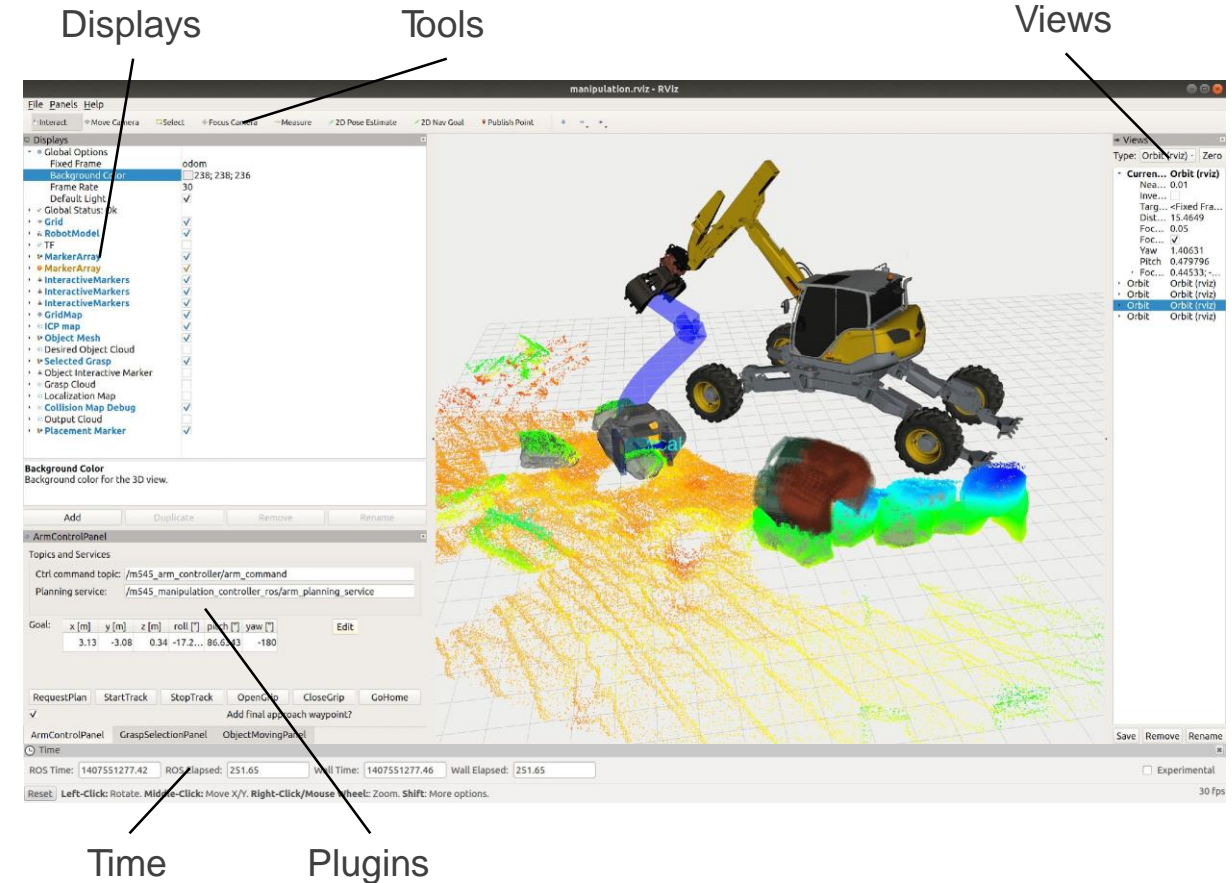
## *FollowPath.action*

```
navigation_msgs/Path path
---
bool success
---
float32 remaining_distance
float32 initial_distance
```

- » 3D-Visualisierungstool für ROS
- » Abonnieren von Topics und visualisieren von Inhalten der Messages
- » Verschiedene Kameraansichten (orthografisch, von oben nach unten, usw.)
- » Interaktive Werkzeuge zur Veröffentlichung von Benutzerinformationen
- » Speichern und Laden von Einstellungen als RViz-Konfiguration
- » Erweiterbar mit Plugins

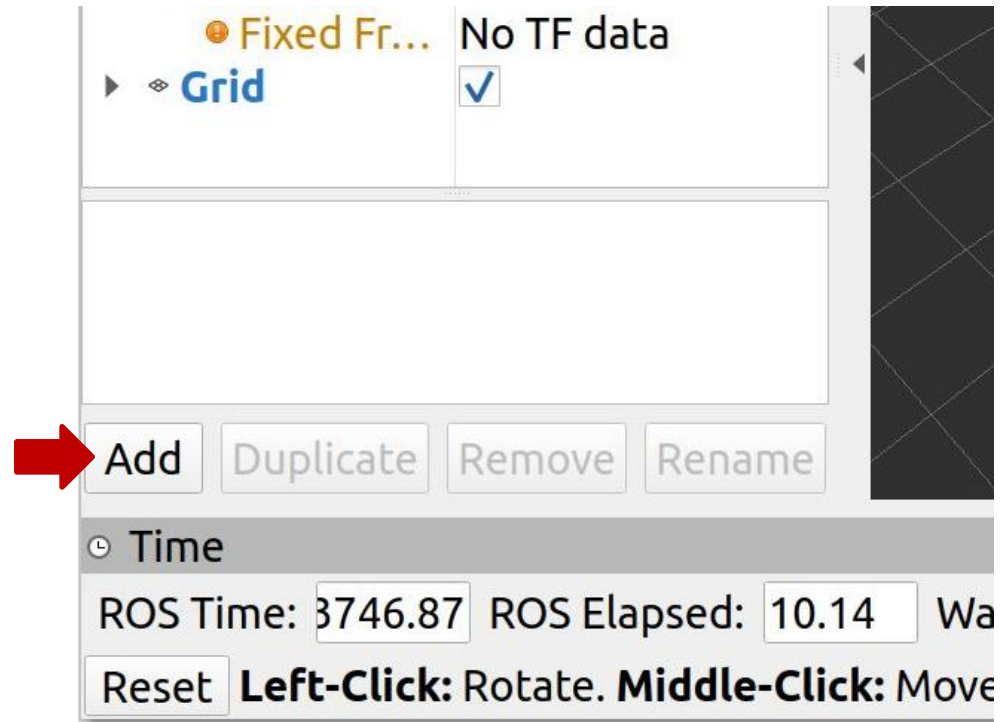
Rviz starten

```
> rviz
```



# RViz

## Display Plugins



Konfiguration speichern mit Strg + S

- |                    |                  |
|--------------------|------------------|
| Axes               | Odometry         |
| Camera             | Path             |
| DepthCloud         | PointCloud       |
| Effort             | PointCloud2      |
| FluidPressure      | PointStamped     |
| Grid               | Polygon          |
| GridCells          | Pose             |
| Group              | PoseArray        |
| Illuminance        | Range            |
| Image              | RelativeHumidity |
| InteractiveMarkers | RobotModel       |
| LaserScan          | TF               |
| Map                | Temperature      |
| Marker             | WrenchStamped    |
| MarkerArray        |                  |

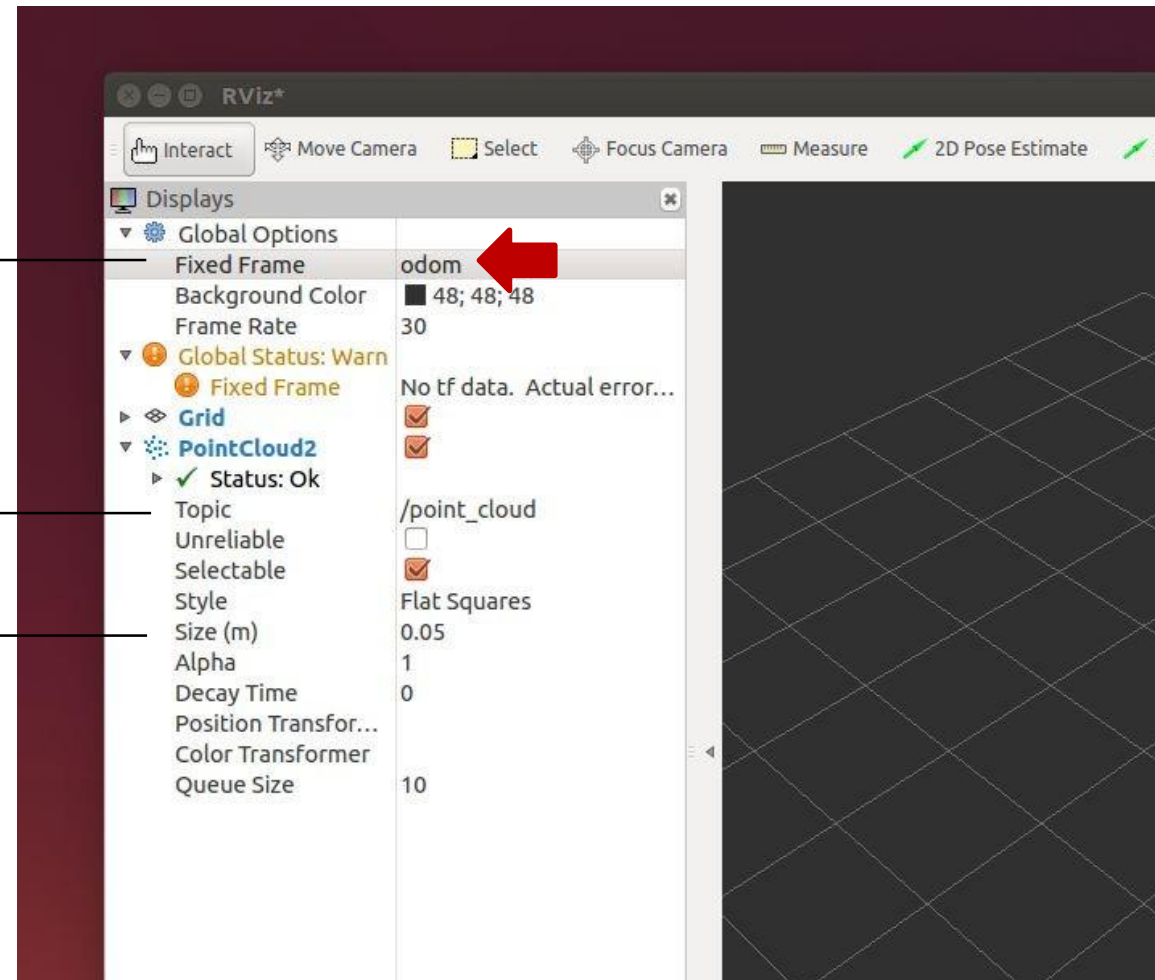
# RViz

## Visualisierung

! Frame, in dem die Daten angezeigt werden (muss vorhanden sein!)

Topic zum Anzeigen wählen

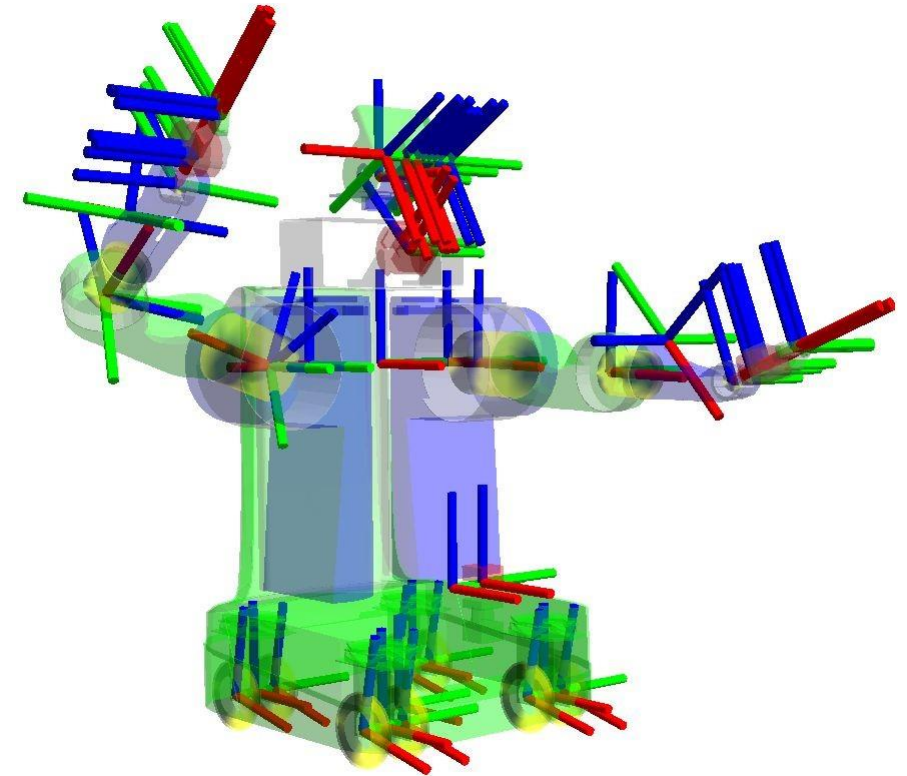
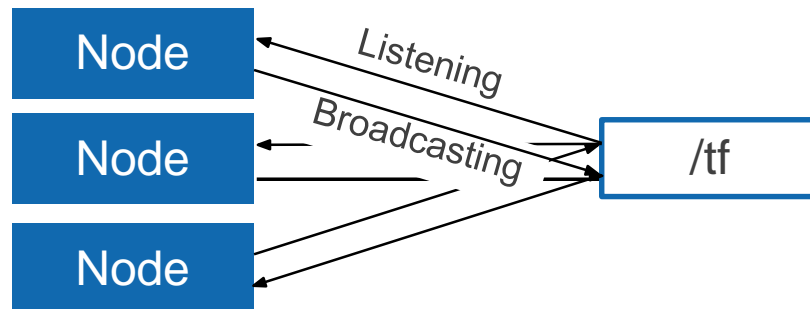
Anzeigeoptionen ändern (z. B. die Größe)



- » Verfolgung von Position, Ausrichtung und Geschwindigkeit von Objekten im Raum
- » Zustand / State von Robotern verfolgen
  - Zentrale Voraussetzung der Robotik
- » Beschreibung in 3 Dimensionen
  - x, y, z-Achsen **right-handed Koordinatensystem**
  - x = forward, y = left, z = up
  - Position und Rotation (Pose)
  - Pose muss in Relation zu Koordinatenframe sein
- » Koordinatenframes
  - „world“ Position
  - Relative Position

# TF Transformation System

- » Werkzeug zur Verfolgung von Koordinatenframes
- » behält Beziehung zwischen Koordinatenframes in einer gepufferten Baumstruktur bei
- » ermöglicht dem Benutzer die Transformation von Punkten, Vektoren usw. zwischen Koordinatenframes zu gewünschten Zeitpunkten
- » Implementiert als Publisher/Subscriber-Modell auf den Topics `/tf` und `/tf_static`





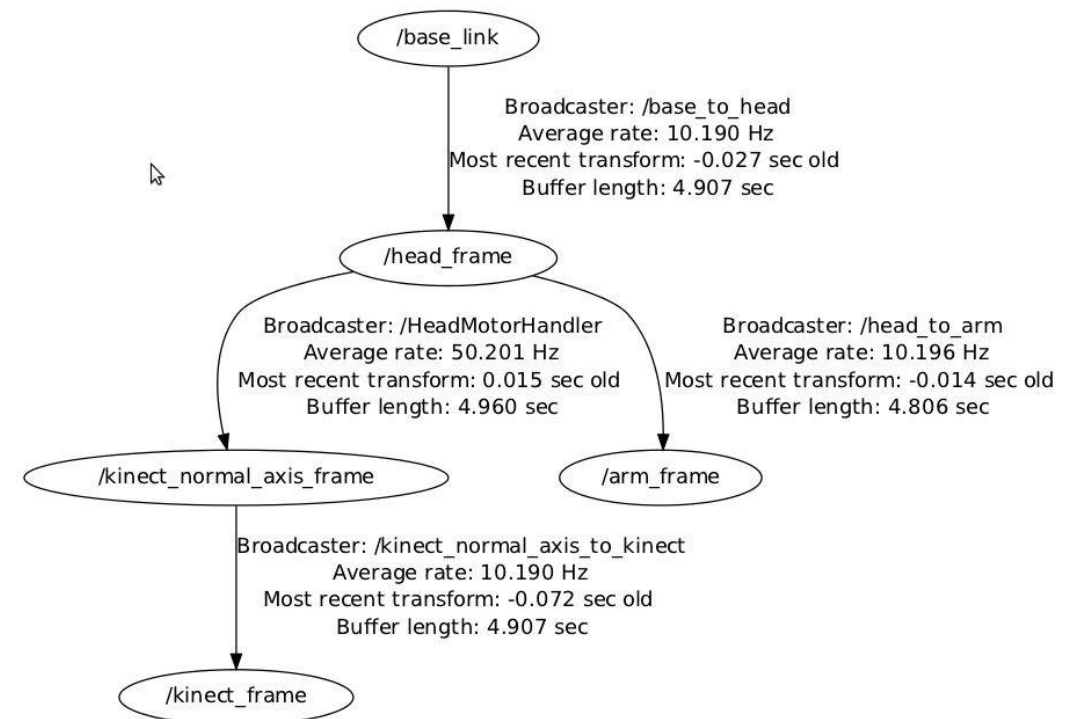
# TF Transformation System

## Transform Tree

- » TF-Listener verwenden einen Puffer, um alle gesendeten Transformationen zu hören
- » Abfrage nach bestimmten Transformationen aus dem Transformationsbaum

*tf2\_msgs/TFMessage.msg*

```
geometry_msgs/TransformStamped[] transforms
std_msgs/Header header
uint32 seqtime stamp
string frame_id
string child_frame_id
geometry_msgs/Transform transform
geometry_msgs/Vector3 translation
geometry_msgs/Quaternion rotation
```



# TF Transformation System Tools

## Command line

Informationen über den aktuellen Transformationsbaum ausgeben

```
> rosrun tf tf_monitor
```

Informationen über die Transformation zwischen zwei Bildern ausgeben

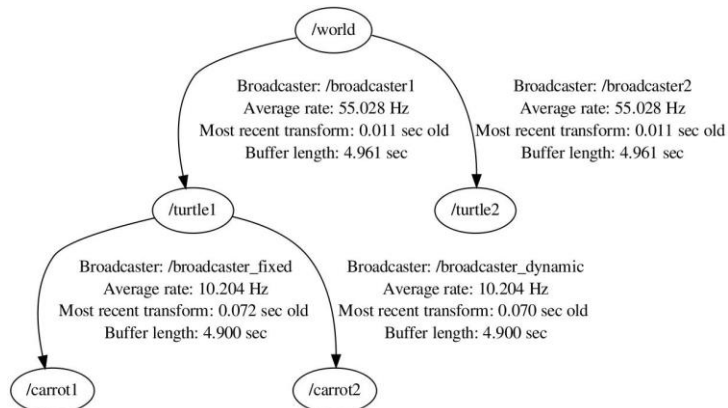
```
> rosrun tf tf_echo  
source_frame target_frame
```

## View Frames

Erzeugt ein visuelles Diagramm (PDF) des Transformationsbaums.

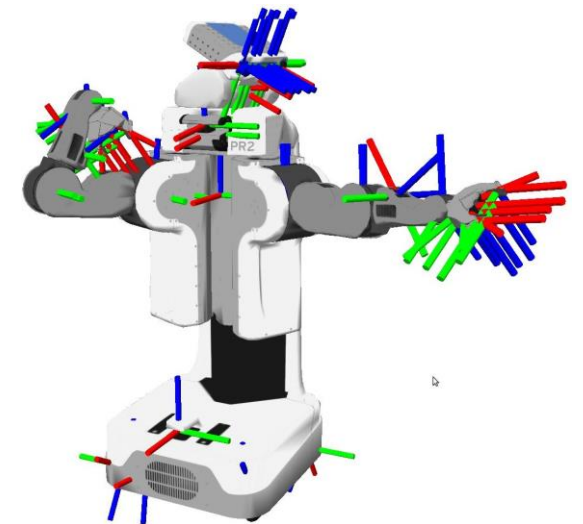
```
rosrun tf view_frames
```

```
rosrun tf2_tools  
view_frames.py
```



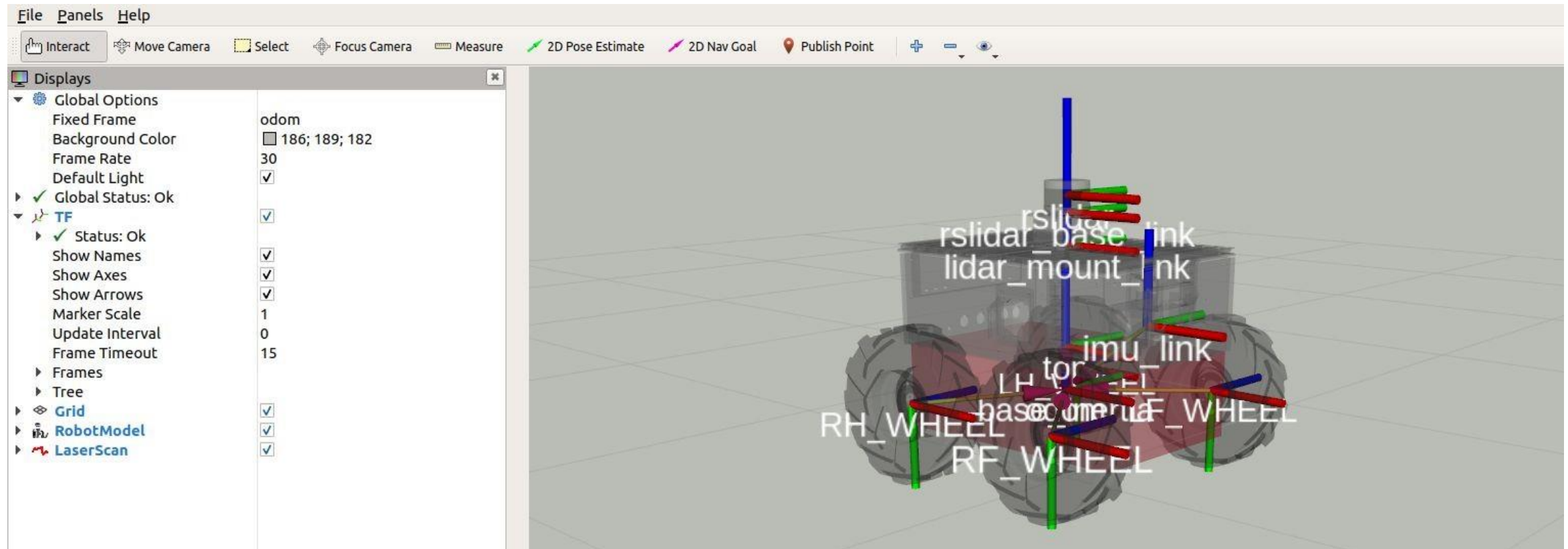
## RViz

3D-Visualisierung der Transformationen





# TF Transformation System RViz Plugin



# ROS Parameter, Dynamic Reconfigure, Topics, Services, und Actions Vergleich I

	Parameters	Dynamic Reconfigure	Topics	Services	Actions
<b>Beschreibung</b>	Globale konstante Parameter	Lokale, veränderbare Parameter	Kontinuierliche Datenströme	Blockierender Aufruf zur Bearbeitung einer Anfrage	Nicht-blockierende, zielorientierte Aufgaben
<b>Anwendung</b>	Konstante Einstellungen	Tuning-Parameter	Kontinuierlicher Datenfluss in eine Richtung	Trigger oder Berechnungen	Aufgabenausführungen und Roboteraktionen
<b>Beispiele</b>	Topic-Namen, Kameraeinstellungen, Kalibrierungsdaten, Robotereinrichtung	Controller-Parameter	Sensordaten, Roboterzustand	Änderung triggern, Zustand abfragen, Menge berechnen	Navigation, Greifen, Bewegungsausführung

# ROS Parameter, Dynamic Reconfigure, Topics, Services, und Actions Vergleich II

Parameters			Topics	
Beschreibung	Globale konstante Parameter		Kontinuierliche Datenströme	
Anwendung	Konstante Einstellungen		Kontinuierlicher Datenfluss in eine Richtung	
Beispiele	Topic-Namen, Kameraeinstellungen, Kalibrierungsdaten, Robotereinrichtung		Sensordaten, Roboterzustand	

# ROS Bags

- » Ein Bag ist ein Format zum Speichern von Nachrichtendaten
- » Binäres Format mit der Dateierweiterung \*.bag
- » Geeignet für die Protokollierung und Aufzeichnung von Datensätzen zur späteren Visualisierung und Analyse

Alle topics in einem bag festhalten

```
> rosbag record --all
```

Spezifische topics festhalten

```
> rosbag record topic_1 topic_2  
topic_3
```

- » Aufnahme mit Strg + C stoppen
- » Bags werden mit Startdatum und Uhrzeit als Dateiname im aktuellen Ordner gespeichert (z.B. 2022-04-27-13-34-13.bag)

Informationen über ein bag ausgeben

```
> rosbag info bag_name.bag
```

Ein bag lesen und Inhalt veröffentlichen

```
> rosbag play bag_name.bag
```

Abspieloptionen können definiert werden, z.B.

```
> rosbag play --rate=0.5  
bag_name.bag
```

--rate=*factor* Publish rate factor

--clock Publish the clock time

--loop Loop playback etc.

## Debuggen mit den erlernten Werkzeugen

- » Code häufig kompilieren und ausführen, um Fehler frühzeitig zu erkennen
- » Verstehen von Kompilierungs- und Laufzeitfehlermeldungen
- » Analysewerkzeuge zur Überprüfung des Datenflusses verwenden (roscat info, rostopic echo, usw.)
- » Daten visualisieren und darstellen (RViz)
- » Programm in kleinere Schritte unterteilen und Zwischenergebnisse überprüfen (ROS\_INFO, ROS\_DEBUG usw.)
- » Code robust machen, indem man Argumente und Rückgabewerte überprüft und Exceptions abfängt
- » Unit-Tests und Integrationstests schreiben

# Beispiel Turtlesim

## » Turtlesim

- Standard-ROS-Einführungsbeispiel
- <http://wiki.ros.org/turtlesim>

## » Turtlesim installieren

```
> sudo apt-get install ros-$(rosversion -d)-turtlesim
```

## » Master starten

```
> roscore
```

## » Turtlesim starten

```
> rosrn turtlesim turtlesim_node
```

» Einfacher Simulator für Erlernung ROS Konzepte

» **Messages**

- [geometry\\_msgs/Twist Message](#)

» **Subscribed Topics**

- `turtleX/cmd_vel` (`geometry_msgs/Twist`)

- Lineare und Winkelgeschwindigkeits-Kommandos für `turtleX`.

Turtle führt Geschwindigkeitsbefehl für 1 Sekunde aus, dann Timeout.

- `Twist.linear.x` = Vorwärtsgeschwindigkeit

`Twist.linear.y` Seitwärtsgeschwindigkeit

`Twist.angular.z` = Winkelgeschwindigkeit.

## » Message

- [turtleSim/Pose](#)

## » Published Topics

- Zustand / Pose der Turtle im freien Raum
- x, y Position, theta (Ausrichtung),  
Geschwindigkeit und Winkelgeschwindigkeit



## » Services

- `reset (std_srvs/Empty)`
  - » Zurücksetzen auf Startkonfiguration
- `kill (turtlesim/Kill)`
  - » bestimmte Turtle zerstören (Name muss angegeben werden).
- `spawn (turtlesim/Spawn)`
  - » Turtle bei x, y und theta erstellen, gibt Namen zurück.
- `turtleX/teleport_absolute (turtlesim/TeleportAbsolute)`
  - » Teleportiert turtleX zu x, y, theta.
- `turtleX/teleport_relative (turtlesim/TeleportRelative)`
  - » Teleportiert turtleX eine lineare Distanz und Winkelabstand von turtles jetziger Position.

# Beispiel Turtlesim

- » **Parameter** (weniger wichtig)
  - background\_b (int, default: 255)
  - background\_g (int, default: 86)
  - background\_r (int, default: 69)

## » 2D-Ebene, 11x11

## » Einfache Lineare Bewegung

- Vorwärts/Rückwärts bewegen auf bestimmte Distanz mit konstanter Geschwindigkeit

## » Bewegung zu beliebigem Punkt

- Lineare und Winkelgeschwindigkeit beachten
- sollen proportional zur Distanz und damit variabel sein