NTB
Interstaatliche Hochschule
für Technik Buchs
FHO Fachhochschule Ostschweiz

MME Simulationsprojekt: ROS und Gazebo

Bericht zum Simulationsprojekt über ROS und Gazebo

Andreas Kalberer
andreas.kalberer@ntb.ch

Interstaatliche Hochschule für Technik Buchs NTB
Werdenbergstrasse 4, CH-9471 Buchs

Betreuer:
Prof. Dr. Urs Graf

31. August 2016

Zusammenfassung

Über ROS und Gazebo gibt es eine grosse Menge an Dokumentation und Wiki-Einträgen. Wer aber ROS und Gazebo neu kennenlernen will, ist teilweise von der puren Menge an Information überwältigt. Das Ziel des Simulationsprojekts ist ROS und Gazebo kennenzulernen und ein Dokument zu erstellen, das einen ein einfacheren Einstieg in die Programme ermöglichen soll. Es wird eine Übersicht über einige Komponenten von ROS, wie Kommunikationsinfrastruktur, Tools, Libraries usw., gegeben. Weiter werden die wichtigsten Tools und Befehle von ROS betrachtet und ausprobiert. Während des Simulationsprojektes wurde auch ein Modell eines zweirädrigen Fahrroboters für Gazebo erstellt. Dieses Modell kann anschliessend über ROS angesteuert und somit bewegt werden. Anhand dieses Modells kann der Aufbau der Modellbeschreibung in Gazebo und auch die Modellierung und Ansteuerung von Achsen gezeigt werden.

Abstract

There is a lot of information about ROS and Gazebo on the Internet and on the Wiki pages of these tools. But if you are new to these Tools it can be difficult to find the appropriate document or tutorial, due to the huge amount of information you can find. So this simulation project has the goal to learn ROS and Gazebo and to create a document which helps to find an easy start with these tools. In this document you will find an overview over some main components of ROS, for example its communication infrastructure, tools, libraries and so on. The main tools and commands needed to work with ROS are shown and can be tried out by the reader. During this project, a two-wheeled robot was built for Gazebo. With this model the connection and the control of the model from ROS can be shown. Also you can get a closer look, how a model is built with Gazebo and how a robt axis can be modelled and controlled.

Inhaltsverzeichnis

1. Einleitung

An der Interstaatlichen Hochschule für Technik Buchs NTB wird ein Open Source Robotikframework entwickelt mit dem Namen EEROS[1]. ROS und Gazebo sind zur Zeit die Standards in der Open Source Roboterentwicklung und eine grosse Community arbeitet daran. Genaueres zu den beiden Tools folgt in den nächsten Kapiteln. Durch die grosse Community wäre es für die Verbreitung und auch Bekanntheit von EEROS ein Gewinn, wenn eine ROS bzw. Gazebo Anbindung bereitgestellt werden könnte. Zusätzlich könnte damit auch eine Akzeptanz von EEROS in der ROS-Community erreicht werden. Für ROS existiert eine grosse Menge an Treibern und Bibliotheken für Peripherie, wie zum Beispiel Sensoren (Laserscanner, Kameras etc.). Wenn man nun solche Bibliotheken auch mit EEROS verwenden könnte, würde dies die Entwicklungszeit von Applikationen mit EEROS verkürzen. Ein grosser Unterschied zwischen ROS und EEROS ist darin zu finden, dass mit EEROS echtzeitfähige Aufgaben erledigt werden können, was mit dem normalen ROS nicht möglich ist. Es bestehen zwar Bestrebungen ROS2 echtzeitfähig zu machen, dies ist jedoch noch im Entwicklungsstatus.

An der NTB ist jedoch ROS und Gazebo zur Zeit noch recht unbekannt. Praktisch niemand hat sich bis jetzt damit tiefergehend befasst. Aus diesem Grund soll dieses Simulationsprojekt im Rahmen des Masterstudiengangs Mechatronik dazu verwendet werden ROS und Gazebo selber anzuwenden und genauer anzusehen. Damit soll es möglich werden die Anforderungen definieren zu können, damit EEROS an ROS angebunden werden kann. Zum Beispiel soll geklärt werden, wo und wie EEROS am besten eingebunden werden kann. Wie sehen die nötigen Schnittstellen aus, die ROS anbietet, wie kommuniziert ROS intern und mit externen Bibliotheken usw.

Falls dieses Dokument mit dem Ziel gelesen wird, ROS und Gazebo danach selbständig verwenden zu können, wird empfohlen ROS und Gazebo bereits vorher zu installieren. Dadurch können bestimmte Befehle und Teile aus den Tutorials gerade auf dem eigenen Rechner nachvollzogen werden. Eine kurze Anleitung mit ein paar Hinweisen zur Installation ist im Anhang in Kapitel A zu finden.

[1] siehe www.eeros.org

2. Gazebo

2.1. Was ist Gazebo?

Mit einem Simulationstool können Systeme effizient, und ohne dass das effektive System hardwaremässig aufgebaut sein muss, ausgelegt und getestet werden. Für Robotersimulation gibt es ein Open Source Tool mit dem Namen Gazebo. Gazebo hat eine graphische Oberfläche mit welcher Bewegungen usw. dargestellt werden können und bietet Interfaces zu zum Beispiel ROS[1] an. Mit Gazebo können somit schnell eigene Algorithmen getestet werden und zum Beispiel auch die Auswirkungen von Änderungen daran.

In Gazebo können auch Sensoren simuliert werden inklusive Störungen, um zum Beispiel einen Ausweichalgorithmus zu entwickeln. Mit Gazebo kann über TCP/IP kommuniziert werden, dazu wird einem Modell ein so genanntes Plugin hinzugefügt. In diesem Simulationsprojekt wird Gazebo mit ROS verbunden, welches dann über diesen Socket den Roboter ansteuern kann und auch Sensorwerte zurückgelesen werden können. Auf die Details wird zu einem späteren Zeitpunkt genauer eingegangen, wenn das eigene Roboter-Beispiel erstellt und ausgetestet wird.

Das Ziel dieses Simulationsprojektes ist einen einfachen kleinen Roboter zu erstellen und diesen in Gazebo zu simulieren und mit ROS anzusteuern. Dazu werden zuerst die Grundlagen von ROS und Gazebo Schritt für Schritt erarbeitet und vertieft. Als Vorlage dient der Unterrichtsroboter *Robi*, der am Institut für Ingenieurinformatik entwickelt wurde.

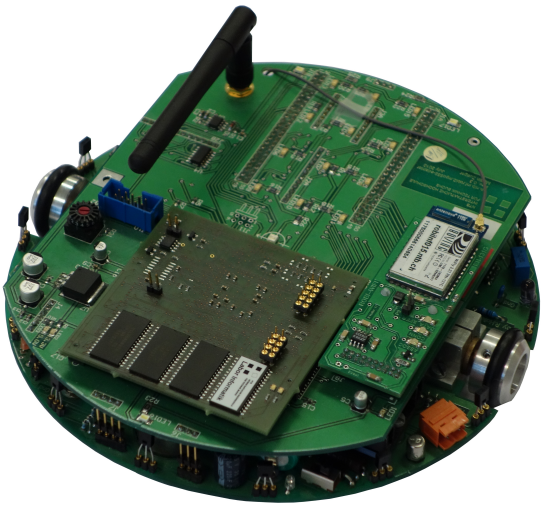

Abbildung 2.1.: Unterrichtsroboter Robi

[1] **R**obot **O**perating **S**ystem, siehe Kapitel 3 ROS

2.2. Robotermodell

In Gazebo gibt es die Möglichkeit Modelle im SDF- oder URDF-Format zu beschreiben. Die Formate sind im XML-Stil gehalten, wobei URDF (**U**nified **R**obot **D**escription **F**ormat) das traditionelle Format für Modelle in ROS ist. In URDF werden aber nicht alle Features unterstützt, die man in Gazebo beschreiben kann und möchte, und so ist empfohlen die Modelle im SDF Format zu beschreiben, falls das Modell in Verbindung mit Gazebo verwendet wird. SDF steht für **S**imulation **D**escription **F**ormat.

Dabei wird die Geometrie des Modells in zwei Arten beschrieben, einmal für das Kollisions-Modell und einmal für die Visualisierung. Dabei ist darauf zu achten, dass das Kollisions-Modell möglichst einfach gehalten wird, um die Simulation effizienter ausführen zu können. Als visuelles Modell kann auch ein CAD-Modell als Mesh verwendet werden, damit das Aussehen möglichst dem realen Vorbild entspricht.

Für Solid-Works gibt es ein Plugin mit welchem URDF-Modelle exportiert werden können. Dazu gibt es auf dem Internet ein Tutorial wie ein solches .urdf File erzeugt werden kann. Auf der Gazebo Homepage gibt es ein Tutorial dazu (http://gazebosim.org/tutorials/?tut=ros_urdf) und auf der Solid-Works Homepage[2] gibt es eine Anleitung wie das Plugin verwendet wird (siehe Fussnote).

Für das im Projekt realisierte Robi-Modell wird das Modell im SDF-Format beschrieben, da dies einfacher ohne Solid-Works erstellt werden kann und einfach verständlich und erweiterbar ist. Ein gutes Tutorial für die Erstellung eines eigenen Modells ist auf der Gazebo Homepage unter http://gazebosim.org/tutorials?tut=build_robot&cat=build_robot zu finden. Mit diesem Tutorial wurde auch das eigene Robi-Modell erstellt. Nachfolgend werden einige Hauptkomponenten der SDF-Beschreibung anhand des Robi-Modells gezeigt.

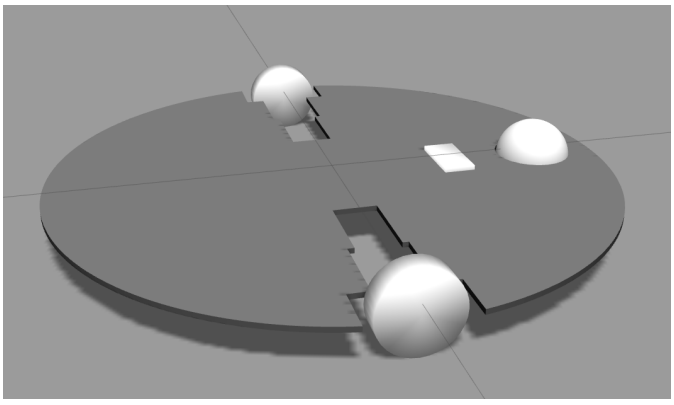

Abbildung 2.2.: Modell Robi in Gazebo

Als Erstes muss die nötige Ordnerstruktur erstellt werden. Bei der Installation von Gazebo wird im Home-Folder ein *.gazebo* Ordner erstellt. Darin befinden sich im Unterordner *models* die

[2] http://blogs.solidworks.com/teacher/wp-content/uploads/sites/3/WPI-Robotics-SolidWorks-to-Gazebo.pdf

Robotermodelle. Für dieses Projekt wird darin ein Ordner *robi* erstellt, worin sich schlussendlich alle nötigen Dateien für das Modell befinden. Für ein Modell werden grundsätzlich zwei Dateien benötigt, ein *model.config*, welches die allgemeine Beschreibung des Modells enthält wie Autor, Version, Kurzbeschreibung des Modells. Auf dieses File wird nicht genauer eingegangen, die Beschreibung dazu inklusiv allen nötigen Zeilen ist im oben genannten Tutorial ausführlich beschrieben. Das zweite File ist das *model.sdf*. Dieses .sdf-File enthält die komplette Beschreibung des Modells mit Geometrie, Trägheiten usw. Bei der Erstellung des Modells wird am besten Schritt für Schritt gearbeitet. So kann viel einfacher identifiziert werden, welcher Teil des Modells in Gazebo nicht funktioniert. Wenn alles auf einmal erstellt wird, ist es praktisch unmöglich herauszufinden, was an der Beschreibung im .sdf-File ein Problem verursacht. Das komplette Modell für den Robi ist in einem Github-Repository[3] zu finden.

Roboterglieder

Listing 1: .sdf Beschreibung Roboterglied

```
<link name="chassis">
    <pose>0 0 .012 0 0 0</pose>
    <inertial>
        <pose>0 0.001 0 0 0 0</pose>
        <inertia>
            <ixx>0.000143414</ixx>
            <iyy>0.000115030</iyy>
            <izz>0.000258336</izz>
            <ixy>-0.0000657706</ixy>
            <ixz>0.00000019534</ixz>
            <iyz>0.00000072647</iyz>
        </inertia>
        <mass>64.32</mass>
    </inertial>

    <collision name='collision'>
        <geometry>
            <cylinder>
                <radius>0.09</radius>
                <length>0.002</length>
            </cylinder>
        </geometry>
    </collision>

    <visual name='visual'>
        <pose>0 0 0.001 0 0 0</pose>
        <geometry>
            <mesh>
                <uri>model://robi/meshes/robi_base2.dae</uri>
            </mesh>
        </geometry>
    </visual>
</link>
```

[3] https://github.com/akalberer/gazebo-robi

In obigem Listing 1 sind alle wesentlichen Elemente zu sehen, die eine Beschreibung eines Links (Roboterglieds) ausmachen. In der ersten Zeile wird dem Link ein Name zugewiesen, dieser wird auch für die Gelenke usw. als Referenz verwendet. Die Position des Links wird mit dem Tag <*pose*> beschrieben. Als nächstes wird die Masse und Trägheitsmatrix des Links beschrieben, damit die Simulation korrekt die Beschleunigungen usw. berechnen kann. Dies wird mit den Tags <*inertial*> und <*inertia*> bzw. <*mass*> beschrieben.

Im zweiten Abschnitt wird das Kollisions-Modell beschrieben. Dies wird in diesem Fall einfach durch einen Zylinder beschrieben und somit so einfach wie möglich gehalten, um den Rechenaufwand klein zu halten. Dieser Abschnitt wird mit dem Tag <*collision*> eingeleitet. Der letzte Abschnitt beschreibt noch die Visualisierung in Gazebo. Dies beschreibt wie das Modell im Gazebo-GUI dargestellt wird und wird mit dem <*visual*> Tag eingeleitet. In diesem Fall wird ein Pfad zu einem Mesh angegeben, welches aus dem Altium File der Leiterplatte des realen Robis generiert wurde. Da im Altium File keine 3D-Modelle für die Bauteile hinterlegt wurden, ist der Detaillierungsgrad des Meshes im Vergleich zum realen Robi (siehe Abbildung 2.1) gering. Es wurde jedoch darauf verzichtet, das Modell detaillierter zu modellieren, da dies nicht dem Hauptinteresse der Arbeit dient.

Der Pfad zum Mesh ist dabei ab dem Unterordner *models* im Gazebo-Ordner anzugeben. Wie ein solches Mesh erzeugt wird, ist im Gazebo-Wiki[4] gut beschrieben. Dazu wird ein .step-File benötigt, der Rest kann mit Gratis-Tools erledigt werden. Dabei ist vor allem die Anpassung der Masseinheiten auf SI-Einheiten zu berücksichtigen, da in CAD-Programmen häufig mit Millimetern gearbeitet wird.

Die oben beschriebenen Elemente von Kollisions-, visuellem Modell etc. müssen für jedes Roboterglied (link) beschrieben werden. Für dieses Robi-Beispiel müssen dann also die Räder usw. für den Robi im gleichen Stil modelliert werden.

Gelenke

Weiter ist die Beschreibung der Gelenke des Roboters ein wichtiger Punkt im .sdf-File. Im Falle unseres Robis handelt es sich dabei um die beiden Gelenke bei den Radachsen. Unten ist als Beispiel die Beschreibung der linken Radachse aufgeführt.

Listing 2: .sdf Beschreibung linke Radachse

```
<joint type="revolute" name="left_wheel_hinge">
    <pose>0 0 0.012 0 0 0</pose>
    <child>left_wheel</child>
    <parent>chassis</parent>
    <axis>
        <xyz>1 0 0</xyz>
        <limit>
            <lower>-10000000000000000</lower>
            <upper>10000000000000000</upper>
        </limit>
```

[4] http://gazebosim.org/tutorials?tut=guided_i2

```
11      </axis>
12  </joint>
```

Im Tag <*joint*> wird das Gelenk definiert. Es gibt verschiedene Arten von Gelenken, auf dem ROS-Wiki[5] gibt es eine gute Zusammenfassung zu den Parametern und möglichen Typen von Gelenken (siehe Fussnote).

Dabei muss die Position der Achse angegeben werden, welches Glied (Link) mit wem verbunden wird usw. Dies geschieht über die vorher für die Glieder vergebenen Namen in der Definition (siehe Listing 1 Zeile 1). Im <*axis*> Tag wird dann angegeben um welche Achse(n) das Gelenk beweglich ist und welche Limiten für diese Achse(n) bestehen, falls dies zum Beispiel sicherheitskritisch wäre. Im Falle einer einfachen Rotationsachse, wie beim Robi,ist das Limit einfach ein sehr grosser Wert.

Ein Roboter bzw. seine Achsen können in Gazebo kann zum Beispiel aus ROS über Sockets angesteuert und bedient werden. Dies erfolgt über ein so genanntes Plugin im Gazebo Modell, welches das Interface zu ROS zur Verfügung stellt. Darauf wird später in Kapitel 4 genauer eingegangen. Zuerst werden jedoch die Grundlagen von ROS betrachtet, um einen besseren Überblick zu erhalten.

[5] http://wiki.ros.org/urdf/XML/joint

3. ROS

3.1. Was ist ROS?

ROS (**R**obot **O**perating **S**ystem) ist ein Software Framework für Robotersoftware. ROS ist kein Betriebssystem im eigentlichen Sinne wie Linux oder Windows, sondern eine Sammlung von Bibliotheken, Tools usw. mit dem Ziel Robotersoftware universeller gestalten zu können und möglichst grosse Teile der Software in einem anderen Projekt wiederverwerten zu können. Ein Hauptziel von ROS ist die Entwicklung einer gemeinschaftlichen Robotersoftware mit dem Open Source Gedanken. Jeder aus der Community kann Verbesserungen an der Software vornehmen und einen bestimmten Teil zum Beispiel (weiter-)entwickeln und wieder der Community zur Verfügung stellen. Ein weiterer Vorteil besteht darin, dass ROS durch seinen Aufbau mit verschiedenen Nodes die Verteilung der Rechenlast, zum Beispiel auf verschiedene Rechner oder Prozessoren, ermöglicht. Auch können von einem solchen Node auch mehrere Instanzen gleichzeitig gestartet werden, ohne einen Zusatzaufwand betreiben zu müssen.

Die Homepage von ROS ist unter der URL www.ros.org zu finden und ist eine gute Informationsquelle und enthält viele Tutorials, ein User Forum für Fragen an die Community usw. Die grosse Community von ROS ermöglicht zur Zeit zum Beispiel ein Release einer neuen Version von ROS jedes Jahr. Für diese Projektarbeit und damit in diesem Bericht wurde mit ROS Jade Turtle (Release 2015)[1] gearbeitet. Der Grund besteht darin, dass zur Zeit der Versionsauswahl ROS Kinetic (Release Mai 2016) noch sehr neu war und allenfalls nötige Zusatzinfos zu Bugs etc. dann noch kaum vorhanden sind.

3.1.1. ROS2

Die Entwicklung von ROS begann im November 2007. Seit damals haben sich die Anforderungen an Robotersoftware gewandelt. Mit ROS2 wird versucht einige wichtige Punkte zu verbessern. Mit ROS2 soll es möglich sein, mehrere Roboter anzusteuern. Dies wird heute bereits mit ROS gemacht, aber es gibt noch kein Standard-Vorgehen. Dies soll mit ROS2 vereinheitlicht werden. Auch soll es mit ROS2 möglich werden real-time fähige Komponenten zu schreiben. Dies soll erlauben, dass zum Beispiel ein Motorregler nun direkt in ROS implementiert werden kann. Auch real-time fähige Interprozesskommunikation (IPC) soll mit ROS2 möglich werden. Weiter soll durch die Überarbeitung die Zuverlässigkeit erhöht werden, damit ROS Software auch vermehrt im produktiven Umfeld und ausserhalb von Laboren zum Einsatz kommt. Auch

[1] https://de.wikipedia.org/wiki/Robot_Operating_System#Entwicklungsgeschichte

bietet ROS2 die Möglichkeit die API an neue Bedürfnisse anzupassen und zu optimieren. Gemäss der Design Page zu ROS2[2] sollen die aktuellen Hauptkomponenten von ROS wie Kommunikation über Topics mit Publish/Subscribe, verteilte Berechnungen etc. im Prinzip gleich bleiben, eventuell wird jedoch die API ändern und neue Elemente hinzukommen.

Das restliche Dokument bezieht sich wieder auf die Standard Version von ROS, die als ROS1 bezeichnet wird. ROS beinhaltet einige Hauptkomponenten, die nun anschliessend näher betrachtet werden.

3.2. Aufbau von ROS

Eine Zusammenstellung über die Hauptkomponenten von ROS ist auch auf der ROS-Homepage[3] zu finden mit Links zu den weiterführenden Informationen zu den einzelnen Komponenten.

3.2.1. Kommunikationsinfrastruktur

Eine Hauptkomponente von ROS ist die Kommunikationsinfrastruktur. ROS stellt dabei ein Interface für Interprozesskommunikation (IPC) zur Verfügung. Dabei kann mittels Messages zum Beispiel zwischen zwei Nodes kommuniziert werden, zum Beispiel nach dem Publish-Subscribe Verfahren[4]. Der Hintergrund in ROS dieses Verfahren zu verwenden ist auch darin zu finden, dass der Entwickler der Software gezwungen wird klare Interfaces zwischen den verschiedenen Nodes zu definieren. Dadurch soll die Kapselung und Wiederverwendbarkeit der Software verbessert werden. Im Kapitel 3.3.3 wird genauer auf die ROS Messages eingegangen.

Durch das Publish-Subscribe Verfahren können Daten wie zum Beispiel Sensordaten nicht nur einem einzigen anderen Node gesendet werden, sondern können von verschiedenen Nodes verarbeitet werden. In ROS werden die Nachrichtenkanäle als *Topics* bezeichnet. So kann zum Beispiel auch in ein File geschrieben werden und die Daten so einfach geloggt werden. Diese können zu einem späteren Zeitpunkt einfach wieder abgespielt werden, um so zum Beispiel Optimierungen an der Sensorauswertung mit immer den selben Daten ausführen zu können. So kann der Entwicklungsaufwand reduziert werden, da sichergestellt werden kann, dass eine Änderung im Code mit bestimmten Daten die gewünschte Änderung bewirkt.

In ROS gibt es noch eine zweite Möglichkeit Nachrichten auszutauschen. Dies erfolgt synchron nach dem Request-Response Prinzip. Diese Art der Kommunikation wird in ROS als *Service* bezeichnet. Genauer wird darauf ebenfalls in Kapitel 3.3.3 eingegangen.

Die dritte Nachrichtenart in ROS sind so genannte *Actions*. Diese werden für länger dauernde Aufgaben verwendet. Mit *Actions* kann zum Beispiel ein Task ausgelöst werden, während

[2] http://design.ros2.org/articles/why_ros2.html
[3] http://www.ros.org/core-components/
[4] siehe auch https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

dieser abläuft, kann der Status überwacht werden und am Schluss wird eine Benachrichtigung abgesetzt. Nun können *Actions* während der Ausführung zum Beispiel aufgrund von bestimmten Sensorwerten auch abgebrochen werden.

3.2.2. Tools und Libraries

Mit ROS wurden auch einige Standard-Nachrichten (Messages) definiert, die für die Robotik hilfreich sind. Zum Beispiel wurden Formate definiert, die in Robotern häufig vorkommende Elemente abbilden. Zum Beispiel existiert ein Format für die Position in allen Achsen (x, y, z) sowie auch enthalten die Winkel von allen Achsen (Pose). Durch Verwendung dieser Standard-Messages können anschliessend auch wieder andere Bibliotheken und Tools von ROS ohne Zusatzaufwand verwendet werden.

Auch direkt in ROS vorhanden ist eine umfangreiche Bibliothek für Koordinatentransformationen, die bei Robotern sehr häufig verwendet werden. Genaueres zur Transform (tf) Library ist auf dem ROS Wiki unter http://wiki.ros.org/tf zu finden.

Als weiteres Element wurde eine maschinenlesbare Sprache, um Roboter zu beschreiben, definiert. Dies erfolgt in ROS standardmässig in URDF, siehe dazu auch Kapitel 2.2 Robotermodell. In diesem Tutorial wird jedoch mit der SDF Beschreibung gearbeitet, da Gazebo für die Simulation des Robis verwendet wird und SDF dort Vorteile besitzt, da mehr Features verwendet werden können.

Für ROS existieren Tools mit denen auch graphisch Positionen und Geschwindigkeiten von Gelenken oder Gliedern angezeigt werden können. Auch können in diesen Tools wiederum Positionen für die Achsregler usw. vorgegeben werden. Dabei sind vor allem die Tools *rviz* und *rqt* zu nennen. Der Begriff rviz steht dabei für ROS 3D Robot Visualizer[5] und rqt für ROS Qt[6] Diese Tools können den grössten Teil der Standard-Message Typen verarbeiten und anzeigen. Mit rviz können zum Beispiel auch Sensordaten von einer Kamera, die mittels ROS eingelesen wird, angezeigt werden und somit direkt in einem Tool ausgewertet werden, was der Roboter sieht und welche Werte zum Beispiel bestimmte Regel-Sollwerte dadurch annehmen usw. Dies ermöglicht komfortables Debugging der Roboter-Software.

Mit *rqt* kann zum Beispiel auch angezeigt werden, wie die verschiedenen Nodes und Topics eines Roboters miteinander verbunden sind und was wie zusammenhängt. Zum Beispiel was für Topics existieren für diesen bestimmten Roboter. Dieses Netz kann mit dem Befehl *rqt_graph* erzeugt und angezeigt werden. In den Fussnoten zu den beiden Tools sind ausführliche Dokumentationen zu den beiden Tools zu finden. In diesem Tutorial wird auch bei Verwendung eines dieser Tools nochmals kurz auf die nötigen Befehle eingegangen.

Weiteres und genaueres zu allen Komponenten von ROS ist im ausführlichen Wiki von ROS unter http://wiki.ros.org/ zu finden.

[5] http://wiki.ros.org/rviz
[6] http://wiki.ros.org/rqt

3.2.3. Dateiorganisation

Für ROS gibt es ein Build-System, welches Catkin genannt wird. Jeder User hat auf seiner Festplatte einen so genannten Catkin-Workspace, wo alle Files für eigene oder neue ROS Packages abgelegt sind. Catkin benutzt CMake für den Build-Prozess und Catkin beinhaltet einfach gesagt Scripts, um CMake korrekt zu konfigurieren. Dies soll es dem User einfach gestalten eigene ROS Packages zu builden und auch allenfalls zu releasen. In Catkin gibt es einen Install-Ordner, worin sich dann die gebuildeten Binaries befinden. Daraus könnte dann zum Beispiel ein Debian-Paket erstellt werden usw. Falls CMake noch gänzlich unbekannt sein sollte, findet sich auf dem ROS-Wiki[7] eine Zusammenstellung mit den wichtigsten Elementen, die bei ROS bzw. Catkin benötigt werden.

Im Catkin-Workspace existiert ein *src*, *devel*(opment) und *build* Ordner. Im Source Ordner werden die Sourcen von zusätzlichen ROS Packages abgelegt oder auch von eigenen ROS Packages. In diesem Ordner wird für ein Paket ein neuer Unterordner angelegt, der am besten dem Namen des Packages entspricht, um die Übersichtlichkeit zu verbessern. Jedes ROS Package braucht ein *CMakeLists.txt* File für den Build-Prozess mit CMake sowie ein *package.xml* File, welches das Manifest File des Packages darstellt. Im Unterordner für das Package können dann je nach Bedarf wieder zusätzliche Unterordner angelegt werden für Scripts und Definitionen von Messages (msg) usw., die benötigt werden. Eine ausführliche Erklärung zum Catkin Workspace ist auf dem ROS-Wiki unter http://wiki.ros.org/catkin/workspaces zu finden. Unten in Listing 3 ist ein Auszug von oben genannter Ordnerstruktur zu sehen:

Listing 3: catkin file organization

```
catkin_workspace_folder/    // catkin workspace
  src/                      // source files
    CMakeLists.txt          // toplevel CMake file for whole catkin workspace
    package_1/              // folder for package with name package_1
      CMakeLists.txt        // CMake file for package_1
      package.xml           // manifest file for package_1
      src/                  // source folder of package_1
            *.cpp
            ...
      scripts/              // scripts folder
            *.py
            ...
      msg/                  // messages folder
            ...
      ...                   // all other needed folders and files for package_1
    package_n/
      CMakeLists.txt
      package.xml
      ...
  build/                    // build folder for catkin
    CATKIN_IGNORE           // catkin will ignore this directory
  devel/                    // development folder (set by CATKIN_DEVEL_PREFIX)
    bin/
    include/
```

[7] http://wiki.ros.org/catkin/CMakeLists.txt

```
    lib/
    .catkin
    env.bash
    setup.bash
    setup.sh
    ...
  install/                  // install folder (set by CMAKE_INSTALL_PREFIX)
    bin/
    etc/
    ...
```

Catkin bietet auch die Möglichkeit mehrere Packages zusammen zu builden und zu installieren. Dies wird dann als so genanntes Meta-Package bezeichnet. So können einfach mehrere eigene Packages an die Community verteilt werden.

Nachfolgend werden einige Befehle und nützliche Hinweise zur Verwendung von Catkin aufgelistet.

Installation

Für Ubuntu Distributionen, die ROS Version Groovy oder höher verwenden, kann Catkin mit apt-get installiert werden:

```
$ sudo apt-get install ros-jade-catkin
```

Alle anderen Distributionen können das Paket entweder von PyPi[8] via pip installieren oder direkt aus den Sourcen builden. Dies ist auf dem ROS-Wiki[9] beschrieben (siehe Fussnote) und deshalb wird hier nicht genauer darauf eingegangen, da sowieso empfohlen ist Ubuntu für ROS zu verwenden.

Verwendung von catkin

Standardmässig wird der Catkin-Workspace im Home Ordner abgelegt (z.B. ~/catkin_ws/), nachfolgend wird mit diesem Standardpfad gearbeitet.

Mit folgendem Befehl kann der Build des Catkin Workspace angestossen werden. Es empfiehlt sich, dies immer vom Hauptordner des Catkin Workspaces zu tun.

```
$ cd ~/catkin_ws
$ catkin_make
```

Dabei werden alle Packages, die sich im Unterordner src befinden miteinbezogen.

Es können alle Standard-Befehle, die man üblicherweise für CMake oder make verwendet, auch an *catkin_make* übergeben werden. Zum Beispiel, wenn die Pakete installiert werden sollen, kann einfach folgendes eingegeben werden:

[8] http://pypi.python.org/pypi
[9] http://wiki.ros.org/catkin?distro=jade#Installing_catkin

```
$ cd ~/catkin_ws
$ catkin_make install
```

Im Anhang in Kapitel C ist noch eine Anleitung zu finden, wie ein eigenes ROS Package mit Catkin erstellt wird. Darauf wird an dieser Stelle jedoch verzichtet, da zuerst einige Grundlagen von ROS bekannt sein müssen und sonst der nötige Hintergrund fehlt. Dies ist auch erst für den erweiterten Gebrauch von ROS nötig, für das einfache Kennenlernen von ROS und Gazebo existieren die nötigen Packages bereits.

3.3. ROS Komponenten

3.3.1. ROS Master

Der ROS Master ist die eigentliche Kernkomponente von ROS. Ohne den ROS Master wissen die Nodes[10] nichts voneinander. Denn der ROS Master ist wie eine Registrierungsstelle, wo sich die Nodes entsprechend registrieren und auch die Topics, auf denen Messages gesendet oder empfangen werden, sind dort vermerkt. Der ROS Master sorgt dafür, dass sich die verschiedenen Nodes finden und miteinander kommunizieren können. Die Kommunikation zwischen den Nodes erfolgt dann direkt untereinander (peer-to-peer) und nicht mehr über den ROS Master. Im Anschluss zu den folgenden einleitenden Kapiteln wird dazu auch noch ein Beispiel gezeigt, dazu sollten jedoch Grundkenntnisse zu Nodes und ROS Messages vorhanden sein. Der ROS Master stellt auch noch einen Parameter-Server[11] zur Verfügung, wo zum Beispiel Konfigurationsparameter für Nodes zur Laufzeit in einem Verzeichnis abgespeichert werden können. Diese können dann auch von anderen Nodes wieder abgerufen werden und entsprechend reagieren.

Der ROS Master wird normalerweise in einer Konsole gestartet und zwar mit folgendem Befehl:

```
$ roscore
```

Anschliessend können die benötigten Nodes gestartet werden.

3.3.2. ROS Nodes

Ein ROS Node ist eigentlich ein Executable. Ein Unterschied zu einem normalen Executable besteht jedoch darin, dass von Nodes mehrere Instanzen gestartet werden können. In einem Node werden zum Beispiel Berechnungen durchgeführt, oder es ist ein Treiber für einen Sensor usw. Es ist also ein abgekapseltes Stück Software mit einer bestimmten Funktion. Durch die Kapselung soll auch die Testbarkeit der Software verbessert werden, es kann so eine einzelne Komponente sauber getestet werden, was die Zuverlässigkeit des Gesamtsystems erhöht.

[10] siehe Kapitel 3.3.2 ROS Nodes
[11] http://wiki.ros.org/Parameter%20Server

Nodes können untereinander kommunizieren, die Kommunikation wird dabei vom ROS Master vermittelt. Der Master kann nach der Aufnahme der Kommunikation zweier Nodes grundsätzlich auch beendet werden und die Kommunikation läuft normal weiter, da die Kommunikation zwischen den Nodes peer-to-peer erfolgt. Neue Nodes können dann aber nicht mehr gestartet und zu einem bereits bestehenden Topic hinzugefügt werden, da das Verzeichnis dazu fehlt.

Ein ROS Node kann dabei wie folgt gestartet werden:

```
$ rosrun packageName nodeName
```

zum Beispiel für das Tutorial zum Turtlesim[12] sieht der Befehl wie folgt aus:

```
$ rosrun turtlesim turtlesim_node
```

Turtlesim entspricht dabei dem Namen des Packages und *turtlesim_node* bezeichnet den Node, der gestartet werden soll.

Anschliessend können alle anderen nötigen Nodes gestartet werden, wieder nach dem gleichen Prinzip wie oben.

Um zu sehen welche Nodes aktuell gerade aktiv sind gibt es ebenfalls einen Befehl:

```
$ rosnode list
```

Wenn nur der ROS Master gestartet wird (mit dem Befehl *roscore*), dann ist genau ein Node aktiv und zwar */rosout*. Dieser Node ist immer aktiv, wenn der ROS Master aktiv ist und entspricht dem Standard-Output stdout/stderr.

Wenn nun mit obigem Befehl der Turtlesim Node gestartet wird (*rosrun turtlesim ...*), dann zeigt *rosnode list* einen zweiten aktiven Node an und zwar zusätzlich zu */rosout* auch */turtlesim*.

Um genauere Infos über einen bestimmten Node zu erhalten, hilft der Befehl *rosnode info nodeName*, was für den Turtlesim-Node folgenden Output generiert:

```
$ rosnode info /turtlesim
--------------------------------------------------------------------------------
Node [/turtlesim]
Publications:
 * /turtle1/color_sensor [turtlesim/Color]
 * /rosout [rosgraph_msgs/Log]
 * /turtle1/pose [turtlesim/Pose]

Subscriptions:
 * /turtle1/cmd_vel [unknown type]

Services:
 * /turtle1/teleport_absolute
 * /turtlesim/get_loggers
 * /turtlesim/set_logger_level
 * /reset
 * /spawn
 * /clear
```

[12] http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes

```
 * /turtle1/set_pen
 * /turtle1/teleport_relative
 * /kill


contacting node http://nbinf014:37103/ ...
Pid: 3374
Connections:
 * topic: /rosout
    * to: /rosout
    * direction: outbound
    * transport: TCPROS
 * topic: /turtle1/cmd_vel
    * to: /teleop_turtle (http://nbinf014:43362/)
    * direction: inbound
    * transport: TCPROS
```

Dies listet alle relevanten Infos zu einem Node auf, zum Beispiel was der Node für Topics besitzt (Publish und Subscribe). Zusätzlich sind auch vorhandene Services aufgelistet und was für Connections zur Zeit zum Node bestehen. In diesem Fall besteht eine zum Standard-Output (*/rosout*) und eine zum */teleop_turtle* Node, mit welchem der Turtle angesteuert werden kann. Genauere Hintergründe zum Turtlesim sind auf dem ROS-Wiki[13] zu finden, welches ausführlich beschrieben ist und einen guten Einstieg in ROS darstellt.

3.3.3. ROS Messages

Verschiedene ROS Nodes kommunizieren miteinander über Messages[14], die sie zum Beispiel über Topics austauschen. Dabei werden von Messages diverse Datentypen unterstützt, zum Beispiel primitive Datentypen wie Integer, Float usw. und Arrays davon. Weiter können verschachtelte Datentypen übertragen werden, die zum Beispiel auf roboterspezifische Anwendungen zugeschnitten sind. Eine komplette Liste dazu, welche Typen unterstützt werden ist auf dem ROS-Wiki für Standard Messages[15] und so genannte Common Messages[16] zu finden (siehe zugehörige Fussnoten).

Als Beispiel wird nachfolgend ein Typ aus den Common Messages kurz genauer angesehen. Es wird der Typ *Twist* betrachtet. Dies ist ein Typ, der häufig in Roboteranwendungen Verwendung findet. Dieser definiert einen Vektor im freien Raum und beinhaltet jeweils einen 3-fach Vektor für den linearen Anteil (x-,y-,z-Achse) und Winkel aller Achsen. Zu jedem Datentypen gibt es auf der API-Homepage eine Kurzbeschreibung, was dieser Typ enthält bzw. wie er aufgebaut ist. Für *Twist* ist dies unter http://docs.ros.org/api/geometry_msgs/html/msg/Twist.html zu finden und sieht wie folgt aus:

[13] http://wiki.ros.org/turtlesim
[14] http://wiki.ros.org/Messages
[15] http://wiki.ros.org/std_msgs
[16] http://wiki.ros.org/common_msgs

Raw Message Definition

```
# This expresses velocity in free space broken into its linear and angular parts.
Vector3  linear
Vector3  angular
```

Compact Message Definition

```
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

Die Definition kann auch direkt in einem Terminal betrachtet werden mit *rosmsg*:

```
$ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Bei Bedarf kann auch ein Header einer Message hinzugefügt werden, bei dem dann Dinge wie eine fortlaufende Nummerierung (Sequence ID) oder ein Zeitstempel (Timestamp) zusätzlich abgespeichert werden können. Dieser Header ist unter *std_msgs/msg/Header.msg* definiert. Genaueres dazu findet sich ebenfalls auf dem ROS-Wiki[17].

ROS Messages (und auch Services) werden mit TCPROS übertragen. Dies ist der Transport Layer der Übertragung und verwendet Standard TCP/IP Sockets. Dabei ist in TCPROS noch spezifiziert, welche Bedingungen ein Publisher oder ein Subscriber erfüllen muss bzw. was er senden muss. Dies beinhaltet zum Beispiel für den Publisher, dass er die MD5-Summe des Message-Typs und den Message-Typ senden muss. Eine komplette Liste ist auf dem ROS-Wiki[18] zu finden.

3.3.4. Topics

Über Topics werden ROS Messages zwischen Nodes ausgetauscht. Ein Topic ist wie ein Bus, welcher über den Namen identifiziert wird. Die Kommunikation über Topics erfolgt nach dem Prinzip Publish/Subscribe. Dabei weiss ein Publisher nicht, wem er genau diese Daten sendet, die Übertragung ist anonym. Er übermittelt die Daten einfach auf diesem Topic und jemand hört zu. Dabei kann es mehrere Publisher und auch Subscriber auf dem gleichen Topic geben. Dies muss vor allem beachtet werden, wenn von einem Node mehrere Instanzen gestartet werden. Dann publishen nämlich beide Nodes auf den selben Topic. Falls dies nicht gewünscht ist, muss mittels Name-Remap der Topic umbenannt werden. Siehe dazu im Kapitel 3.3.7 über

[17] http://wiki.ros.org/msg#Header
[18] http://wiki.ros.org/ROS/TCPROS

Launch-Files. Dort ist beschrieben, wie ein Node/Topic mit verschiedenen Namen gestartet werden kann.

Damit eine Message über einen Topic empfangen werden kann, muss der Typ der Message gleich sein von Publisher und Subscriber. Auch besitzt ein Topic eine MD5-Summe, die aus dem .msg-File generiert wird. Diese muss ebenfalls übereinstimmen, sonst können keine Nachrichten ausgetauscht werden. Somit soll verhindert werden, dass per Zufall zwar etwas empfangen wird, aber der falsche Typ zum Beispiel zu falschen Werten führt. Oder auch, dass eine ältere Version des Message-Typs ein Feature noch nicht unterstützt, beim Publisher aber bereits verwendet wird. Zusätzliche Infos zu Topics finden sich auf dem ROS-Wiki[19].

Mit Hilfe des Befehls `rqt_graph` können die aktiven Nodes sowie die Topics graphisch dargestellt werden. Untenstehende Abbildung 3.1 zeigt den Graphen für das Turtlesim Beispiel. Dabei ist rot der Topic */turtle1/cmd_vel* markiert, mit welchem die Fahrbefehle vom *teleop_turtle* Node an den *turtlesim* Node übermittelt werden.

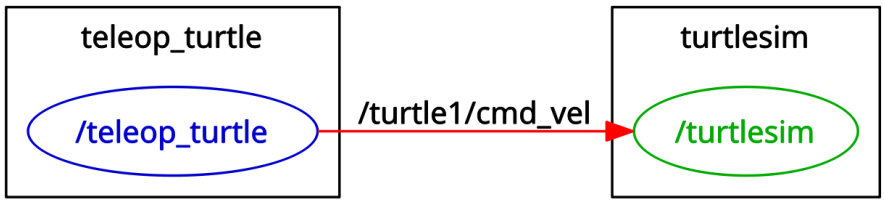

Abbildung 3.1.: rqt_graph Turtlesim Topic

Ein einfaches Beispiel zur Verwendung von Topics gibt es auf dem ROS-Wiki[20]. Dabei ist ein einfaches Publisher- und Subscriber-Beispiel in C++ gezeigt. Nachfolgend wird kurz auf die wichtigsten Code-Ausschnitte daraus eingegangen.

Als erster Punkt muss natürlich ROS initialisiert werden usw., darauf wird aber nicht genauer eingegangen, da sich dieser Teil auf Topics bezieht.

Zuerst muss das Header-File für den Message-Typ hinzugefügt werden, damit die Definition der Message bekannt ist:

```
1 #include "std_msgs/String.h"
```

In einem nächsten Schritt muss der Publisher initialisiert werden:

```
1 ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
```

Das Objekt *n* ist vom Typ *ros::NodeHandle*, dabei wird ROS bekannt gemacht, dass dieser Node eine Message vom Typ *std_msgs::Strings* publishen wird und zwar auf dem Topic *chatter*. Die Zahl 1000 beschreibt die Buffergrösse für den Publisher. Falls der Buffer überläuft, wenn zum Beispiel zu viele Zeichen zu schnell versendet werden, gehen natürlich Zeichen verloren.

Füllen des Feldes *data* der Message mit dem gewünschten Inhalt:

[19] http://wiki.ros.org/Topics
[20] http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(c%2B%2B)

```
std_msgs::String msg;

std::stringstream ss;
ss << "hello world " << count;
msg.data = ss.str();
```

Publishen der Message auf dem Topic *chatter*:

```
chatter_pub.publish(msg);
```

Für den Subscriber ist das Prinzip gleich, die wichtigsten Punkte:

```
ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
```

Oben ist ebenfalls die Initialisierung des Subscribers zu sehen, der letzte Parameter gibt noch die Callback-Function an, die aufgerufen wird, sobald eine Message empfangen wurde. Diese Callback-Function muss dann natürlich implementiert werden:

```
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```

Mit *ROS_INFO* können Ausgaben in ROS gemacht werden, es gibt auch noch diverse andere Funktionen zum Beispiel für verschiedene Debug Ausgaben, siehe dazu in der Dokumentation der ROS-Console auf dem Wiki[21].

Weiter muss noch folgende Zeile hinzugefügt werden:

```
ros::spin();
```

Diese sorgt dafür, dass die User Callback-Function überhaupt aufgerufen wird. Obige Zeile wird erst verlassen, wenn `ros::shutdown()` auftritt oder Ctrl-C gedrückt wird. Es gibt noch die Funktion `ros::spinOnce()`, welche in Loops aufgerufen werden kann. Eine komplette Aufstellung über die verschiedenen Möglichkeiten, auch für Multithread-Applikationen ist auf dem ROS-Wiki[22] zu finden.

3.3.5. Services

Ein Service ist wie bereits zu einem früheren Zeitpunkt erwähnt darauf ausgerichtet direkt auf eine bestimmte Anfrage zu reagieren. Das Publish/Subscribe Verfahren der Topics ist dafür nicht geeignet und deshalb gibt es diese zweite Kommunikationsvariante in ROS nach dem Request/Response Prinzip[23] und wird Service[24] genannt. Dabei stellt ein Node einen Service unter einem bestimmten Namen zur Verfügung, der Client sendet die Anfrage und wartet auf die

[21] http://wiki.ros.org/rosconsole
[22] http://wiki.ros.org/roscpp/Overview/Callbacks%20and%20Spinning
[23] https://en.wikipedia.org/wiki/Request%E2%80%93response
[24] http://wiki.ros.org/Services

Antwort. In vielen ROS-Libraries werden die Services auch als RPC (**r**emote **p**rocedure **c**all) bezeichnet.

Auch bei den Services gibt es wie bei den Topics einige Bedingungen, damit die Kommunikation stattfinden kann. Ein Service wird wie ein Topic über den Package Namen + Namen des Service-Files (.srv) definiert. Im Service-File wird der Aufbau des Services beschrieben. Diese Beschreibung ergibt dann den Typ des Services. Dieser Typ muss zwischen Request- und Response-Node übereinstimmen, damit eine Kommunikation zustande kommt, sowie auch wieder eine MD5-Summe, die aus dem .srv-File generiert wird.

Zu den Services gibt es ein gutes Beispiel auf dem ROS-Wiki[25], das die Implementierung eines Service und Client Nodes zeigt (siehe Fussnote). Auf die wichtigsten Elemente aus dem Tutorial wird in diesem Dokument kurz eingegangen, ansonsten soll das Tutorial als Referenz dienen. Bevor obiges Tutorial durchgearbeitet wird, muss noch das darin benötigte Service-File erstellt werden. Dieses wird in einem weiteren Tutorial[26] erstellt und ist dort auch noch verlinkt.

Die Definition eines Service-Types sieht zum Beispiel wie folgt aus:

```
1 int64 A
2 int64 B
3 ---
4 int64 Sum
```

Dabei sind die zwei ersten Zeilen (`A`, `B`) die Request-Typen, dann wird in diesem Beispiel mit der dritten Zeile `---` getrennt. Darunter folgen dann die Response-Typen, in diesem Fall `int64 Sum`.

Aus diesem Service File können mit Catkin dann alle nötigen Files generiert werden für die weitere Verwendung. Dabei wird auch ein Header-File für C++ generiert. Dieses muss dann im Source-File hinzugefügt werden, damit dieser Service-Typ verwendet werden kann. Dies bezieht sich nicht nur auf Services sondern auch auf selber definierte Message-Typen, die dann zum Beispiel für Topics verwendet werden.

```
1 #include "beginner_tutorials/AddTwoInts.h"
```

Der Teil *beginner_tutorials* entspricht dabei dem Package Namen und *AddTwoInts* dem Namen des .srv-Files. Wie ein eigenes Package mit eigenem Service-File erzeugt werden kann, ist auch im Kapitel C beschrieben.

Die folgenden Zeilen beziehen sich auf den Service-Teil des Tutorials. Die Implementierung der gewünschten Callback-Funktion muss noch erfolgen, in diesem Falle die Addition der beiden Werte, die im Request übergeben werden und die Zuweisung des Resultats an das Response-Element:

[25] http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(c%2B%2B)
[26] http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv#Creating_a_srv

```
bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res){
    ...
}
```

Wie bei der Verwendung von Topics muss auch ein Service beim Node (*n*) angemeldet werden. Dies erfolgt mit folgenden zwei Zeilen:

```
ros::ServiceServer service = n.advertiseService("add_two_ints", add);
```

Dabei wird eine Beschreibung angegeben, sowie die Callback-Funktion übergeben. In diesem Falle wird auf die Funktion *add* verwiesen.

Und auch wieder nicht fehlen darf *ros::spin()*, damit die Callback-Funktion überhaupt aufgerufen wird:

```
ros::spin();
```

Beim Client Code sind die Includes für den Request/Response Teil gleich wie beim Service Code. Die Zeile für die Registrierung des Clients beim Node sieht wie folgt aus:

```
ros::ServiceClient client = n.serviceClient<beginner_tutorials::AddTwoInts>("
    add_two_ints");
```

Der Template-Parameter gibt dabei den Service-Typen an.

Der anschliessende Aufruf des Services sieht wie folgt aus:

```
if (client.call(srv)){
...
```

Dabei ist zu erwähnen, dass dieser Call auf den Service blockierend ist. Deshalb muss nicht in einem Loop oder Ähnlichem gewartet werden. Sobald der Call abgeschlossen ist wird an dieser Stelle weiter ausgeführt und der Rückgabewert des Aufrufs verarbeitet.

3.3.6. Actionlib

Für eine weitere Variante der Kommunikation zwischen Nodes sorgt das Paket *actionlib*[27]. Üblicherweise wird Request/Response Kommunikation in ROS mit Services erledigt. Wenn jetzt aber ein Request lange dauert, möchte man eventuell den aktuellen Status des Requests wissen oder allenfalls auch abbrechen können. Dies kann über Actions erfolgen. Actions sind auf ROS-Messages aufgebaut und werden durch drei Elemente beschrieben: *Goal*, *Feedback* und *Result*.

Das *Goal* beschreibt, wie der Name schon sagt, was der Task erreichen soll. Also zum Beispiel eine Endposition, ein Objekt, das über einen Laserscanner erkannt wird oder Ähnliches. Das *Feedback* wird während des aktiven Tasks laufend an den Client zurück gesendet. Dies ist eigentlich der momentane Status, während der Task aktiv ist. Und wenn der Task erfolgreich

[27] http://wiki.ros.org/actionlib

ausgeführt wurde, wird mittels *Result* signalisiert, dass der Task abgeschlossen wurde und zum Beispiel die gewünschte Endposition erreicht wurde.

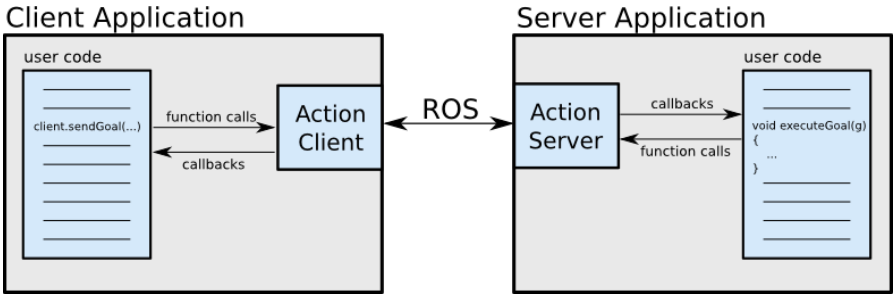

Abbildung 3.2.: Prinzip Client Server *actionlib*; Quelle: ROS Wiki 2016: *Client-Server Interaction* URL:http://wiki.ros.org/actionlib

Die Grafik zeigt den prinzipiellen Aufbau der *actionlib*. Auf der Client Seite können mittels Funktionsaufrufen Requests initiiert werden (z.B. *sendGoal*). Die Server Seite kann anschliessend wieder mittels einer Callback-Function diesen Request bearbeiten (z.B. *executeGoal*). Die genaueren Hintergründe und wie ein Request auch wieder abgebrochen werden kann ist auf dem ROS-Wiki[28] detailliert beschrieben.

Die Beschreibung einer Action erfolgt gleich wie bei Messages und Services über ein File. Bei Actions wird ein .action File erstellt, welches die Definitionen enthält. Diese Files werden im Package-Ordner in einem Unterordner *action* abgelegt. Im ROS-Wiki zur *actionlib* (siehe Fussnote [27]) gibt es ein einfaches Beispiel von einem Action-File und jeweils ein Beispielprogramm für Python und C++.

Aus dem .action File werden mit Catkin dann wieder die nötigen Headerfiles und auch die verschiedenen Message-Files generiert. Im Wiki-Artikel unter Results[29] ist aufgelistet, was alles generiert wird. Dies ist jeweils je eine Message für Goal, Result und Feedback. Diese Messages werden dann für die Kommunikation zwischen Server und Client benötigt.

3.3.7. Launch Files

Mit einem Launch File kann das Aufstarten von verschiedenen Nodes und Tools vereinfacht und automatisiert werden. Die Idee ist gleich wie bei einem Bash-Script, wo nacheinander Befehle aufgelistet werden, die ausgeführt werden sollen. Die Syntax sieht dabei wie folgt aus:

```
<launch>
    <node pkg="pkg1-name" type="executable-name" name="node_name"/>
</launch>
```

Wobei bei *type* der Name des Executables/Nodes angegeben wird und *name* ein selber gewählter Name bzw. Beschreibung sein kann, wie dieser Node bezeichnet werden soll. Dabei können dann

[28] http://wiki.ros.org/actionlib/DetailedDescription
[29] http://wiki.ros.org/actionlib#Results

in neuen Zeilen mit der gleichen Syntax alle Nodes angegeben werden, die gestartet werden sollen. Nachfolgend ein konkretes Beispiel zur Verdeutlichung für das Turtlesim Tutorial:

Listing 4: Launch File mehrere Nodes

```
<launch>
    <node pkg="turtlesim" type="turtlesim_node" name="turtle_test" ns="sim1"/>
    <node pkg="turtlesim" type="turtlesim_node" name="turtle_2" ns="sim2"/>
    <node pkg="turtlesim" type="turtle_teleop_key" name="turtle_test_teleop" ns="sim1"
        launch-prefix="gnome-terminal -e"/>
    <node pkg="turtlesim" type="turtle_teleop_key" name="turtle_test_teleop" ns="sim2"
        launch-prefix="gnome-terminal -e"/>
</launch>
```

Obiges Listing 4 zeigt das Aufstarten von mehreren Nodes und zwar je zweimal den Turtlesim Node und den Teleop-Key Node. Zwingend erforderlich sind die ersten drei Tags (`node pkg, type, name`).

Das Launch File wird dann wie folgt ausgeführt:

```
roslaunch filename.launch
```

Dabei muss der ROS Master (roscore) nicht auch noch separat gestartet werden. Dieser wird automatisch gestartet, bevor der erste Node im Launch-File ausgeführt wird.

Mit dem *ns*-Keyword kann einem Node ein gewünschter Namespace zugeordnet werden. Der Hintergrund ist wie folgt: Wenn der Teleop-Key Node zweimal gestartet wird, ohne Namespace, dann wird der Topic für beide Nodes identisch sein. Dies wird dazu führen, dass beide Nodes auf dem gleichen Topic publishen. Falls dann auch beide Turtlesim Nodes ohne Namespace gestartet werden ist der Topic *cmd_vel*, auf dem sie die Fahrbefehle empfangen, ebenfalls der Gleiche. Dies führt dazu, dass mit einem Teleop-Key Node beide Turtles gesteuert werden, aber es kann nicht individuell ein einzelner Turtle angesteuert werden. Deshalb wird oben jeweils beim einen Trutlesim Node mit Namespace `sim1` und beim zweiten mit `sim2` gestartet. Die Teleop Nodes werden dann natürlich auch für den entsprechenden Namespace gestartet. Dass dies nun verschiedene Topics sind, sieht man zum Beispiel an der Ausgabe von `rostopic list`:

```
/rosout
/rosout_agg
/sim1/turtle1/cmd_vel
/sim1/turtle1/color_sensor
/sim1/turtle1/pose
/sim2/turtle1/cmd_vel
/sim2/turtle1/color_sensor
/sim2/turtle1/pose
```

Wenn ein Namespace angegeben wird, wird dies dem Topic vorangestellt. Ohne Namepace wäre die Bezeichnung des Topics zum Beispiel nur */turtle1/cmd_vel*.

Obiges Problem, dass zum Beispiel auf den selben Topic geschrieben wird, obwohl man dies vielleicht nicht will, kann noch anders gelöst werden. In einem Launch-File kann auch ein

Remap eines Namens oder Topics gemacht werden. Folgende Zeile zeigt zum Beispiel einen Remap eines Topic Namens:

```
<remap from="cmd_vel" to="cmd_vel_turtle_xy"/>
```

Dieser neue Name gilt dann für alle nachfolgenden Nodes, die sich im gleichen Scope befinden wie der `remap` Tag. So können für Gruppen einfach bestimmte Topics umbenannt werden.

Es gibt noch einige weitere wichtige Schlüsselwörter, zwei davon werden nachfolgend noch kurz beschrieben. Eine Übersicht über alle möglichen Tags und weitere Infos zu Launch-Files sind auf dem ROS-Wiki[30] bzw. [31] zu finden.

Wenn ein Node zum Beispiel zwingend erforderlich ist, damit das Gesamtsystem funktioniert, kann dem Node im Launch-File zusätzlich ein `required` Tag hinzugefügt werden:

```
<node pkg="turtlesim" type="turtlesim_node" name="turtle_test" ns="sim1" required="true
    " />
```

Dies führt dazu, dass alle Nodes, die mit diesem Launch-File gestartet wurden, beendet werden, sobald dieser zwingende Node beendet wurde oder auch abstürzt.

Falls ein kurzer Unterbruch eines Nodes verkraftet werden kann und das System bei einem Absturz eines Nodes, oder wenn dieser geschlossen wurde, wieder startet bzw. weiterläuft gibt es den `respawn` Tag:

```
<node pkg="turtlesim" type="turtlesim_node" name="turtle_test" ns="sim1" respawn="true"
    />
```

Dies führt dazu, dass der entsprechend markierte Node immer wieder gestartet wird, wenn er beendet wurde.

[30] http://wiki.ros.org/roslaunch/
[31] http://wiki.ros.org/roslaunch/XML

4. ROS und Gazebo

Nun sollen ROS und Gazebo zusammen verwendet und betrachtet werden. Das Ziel ist die Ansteuerung unseres in Gazebo modellierten Robis mit ROS. Dazu muss ein Plugin für das Robotermodell erstellt werden und mit dem Modell in Gazebo und ROS verknüpft werden. Im nächsten Kapitel wird nun auf dieses Plugin eingegangen.

4.1. Gazebo Plugin

Damit die Achsen unseres Roboters aus ROS angesteuert werden können muss ein so genanntes Control Plugin erstellt werden. Dabei ist wieder das Gazebo-Wiki ein guter Anlaufpunkt, dort gibt es ein gutes Tutorial[1], welches die Ansteuerung eines beweglichen Sensorkopfes zeigt. Dieses Plugin aus dem Tutorial kann gut auf unseren Robi adaptiert werden. Nachfolgend soll auf die wichtigsten Punkte eingegangen werden, die für das Plugin benötigt werden.

Das Plugin muss nicht im gleichen Ordner wie das Modell erstellt werden. Aus dem Plugin wird eine Shared-Library erstellt (.so-File unter Linux), welche dann beim Start des Modells in Gazebo mitgeladen wird. Deshalb ist es sogar sinnvoll das Plugin in einem neuen Ordner zu erstellen. Das Plugin wird in unserem Fall als *robi_plugin* bezeichnet. Bei der Entwicklung des Plugins ist sicherlich auch wieder ein schrittweises Vorgehen sinnvoll. So kann mit einem sehr einfachen Plugin überprüft werden, ob dieses korrekt geladen wird und die Verbindung zum Robotermodell funktioniert.

In einem ersten Schritt soll das Plugin ermöglichen, dass aus einem C++ Programm ohne Einbindung von ROS die Radachsen angesteuert werden können. Der gesamte Source-Code des Plugins ist in einem Repository auf Github[2] abgelegt. Nachfolgend wird auf die wichtigsten Punkte des Plugins eingegangen.

4.1.1. World-File

Damit ein Plugin für das Modell geladen wird, kann zum Beispiel ein World File erstellt werden. Wie das komplette File aussieht ist auch auf Github zu sehen. Im .world-File kann das gewünschte Plugin angegeben werden, welches mitgeladen werden soll. Dies sieht für den Robi wie folgt aus, es soll ja das Control-Plugin für die Roboterachsen mitgeladen werden.

[1] http://gazebosim.org/tutorials?cat=guided_i&tut=guided_i5
[2] https://github.com/akalberer/robi_plugin

```
<!-- Load the Robi model -->
<model name="robi_model">
    <include>
        <uri>model://robi</uri>
    </include>

    <!-- Attach the plugin to the model -->
    <plugin name="robi_control" filename="librobi_plugin.so">
        <velocity_left>0</velocity_left>
        <velocity_right>0</velocity_right>
    </plugin>
</model>
```

Das Laden des Plugins wird in den *plugin* Tags initiiert. Dabei muss ein Name, sowie die Shared-Library des Plugins (.so-File) angegeben werden. Es ist zusätzlich zu sehen, dass eine Initialgeschwindigkeit für die beiden Achsen definiert wird. Diese Initialwerte werden dann beim Laden des Plugins eingelesen und den Achsen zugewiesen. Darauf wird noch kurz im Kapitel 4.1.2 API für Gazebo eingegangen, was dafür noch auf der Seite des Plugins benötigt wird.

Folgendes ist noch zu beachten. Da sich die Shared-Library nun im build-Ordner des Plugins befindet und wahrscheinlich nicht in einem Ordner, der sich standardmässig im Library-Path befindet, muss dieser Ordner noch zum Pfad hinzugefügt werden. Dies kann durch exportieren des Pfads erreicht werden, dann kann *librobi_plubin.so* gefunden werden:

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:~/PATH/TO/LIBRARY/FOLDER
```

Dies kann natürlich in einem Script automatisiert werden, welches vor dem Start der Simulation ausgeführt wird. Dabei ist darauf zu achten, dass dieses Script mit dem Befehl *source* ausgeführt wird, sonst wird das Script in einer Sub-Shell ausgeführt und der Pfad für die aktuelle Session ist dann trotzdem nicht angepasst.

Um das Modell mit dem Plugin und .world-File zu testen, müssen folgende Befehle ausgeführt werden:

```
source library_path.sh
gazebo robi.world
```

Falls schliesslich auch mit ROS kommuniziert wird, muss wie üblich `roscore` in einem zweiten Terminal zuerst ausgeführt werden, damit sich der ROS Node beim ROS Master registrieren kann. Und anstatt `source library_path.sh` kann natürlich auch wieder oben gezeigter Befehl `export LD_LIBRARY_PATH=...` ausgeführt werden.

4.1.2. C++ Plugin

Nachfolgend wird auf ein paar Elemente aus dem C++-Plugin eingegangen. Dies ist nicht vollständig, soll aber die wichtigsten Teile zeigen. Das komplette C++ Source-File ist ebenfalls im Github Repository (siehe Fussnote 2) zu finden.

Damit das Plugin compiliert und ausgeführt werden kann wird auch hier wieder ein File für CMake (CMakeLists.txt) benötigt. Darin müssen die Packages ROS und Gazebo gesucht werden und ein Target hinzugefügt werden, welches aus den Plugin-Sourcen eine Shared-Library erzeugt. Dabei muss noch beachtet werden für welche Version von Gazebo das Plugin erstellt wird. Für Gazebo älter als Version 6 muss noch mit der Boost-Library gelinkt werden, danach ist dies nicht mehr nötig. Das benötigte CMake File ist im bereits erwähnten Github-Repository zu finden und kann sehr einfach für eigene Plugins adaptiert werden.

Nun wird auf einige wichtige Elemente des C++ Plugins eingegangen. Jedes Gazebo Plugin muss eine *load* Function haben. Diese wird, wie der Name schon sagt, beim Laden des Plugins ausgeführt. Dies ist also die Initialisierung des Plugins und wird ausgeführt, sobald das Modell in die Simulation von Gazebo geladen wird. Der Header sieht dabei wie folgt aus:

```
public: virtual void Load(physics::ModelPtr _model, sdf::ElementPtr _sdf);
```

Dabei wird ein Shared Pointer des Modells an das Plugin übergeben sowie ein Pointer auf das SDF-Modell. Damit können schliesslich im Plugin Eigenschaften des Modells oder auch Parameter aus der SDF-Beschreibung des Modells ausgelesen werden.

Über das Modell kann dann auch auf die im .sdf-File definierten Achsen zugegriffen werden. Dies kann zum Beispiel über den in der SDF-Beschreibung gewählten Namen und Namespace erfolgen:

```
this->joint_left = _model->GetJoint("robi_model::robi::left_wheel_hinge");
```

Über diesen Pointer auf ein Gelenk (Joint), auch wieder ein Shared Pointer, kann dann zum Beispiel ein Regler für die Achse hinzugefügt werden. Dazu können einfach die vordefinierten PID-Regler aus Gazebo verwendet und zugewiesen werden.

```
this->pid_left = common::PID(1, 0.1, 0);
this->model->GetJointController()->SetVelocityPID(this->joint_left->GetScopedName(),
    this->pid_left);
```

Dies ist nun der Punkt wo, wie bereits zu einem früheren Zeitpunkt erwähnt, eine nicht zu alte Version von Gazebo benötigt wird. Dieser Code funktioniert ab Gazebo Version 5, bei den älteren Versionen fehlen dem JointController noch einige Features bzw. die API ist minimaler gehalten.

Nun kann bereits aus dem Plugin eine gewünschte Geschwindigkeit für die Achsen hart codiert werden, um zu versuchen, ob die Regler funktionieren und das Laden des Plugins korrekt funktioniert. Nachfolgende Codezeilen weisen nun den beiden Achsen je eine unterschiedliche Geschwindigkeit zu. Falls alles korrekt funktioniert, wird der Robi eine Rechtskurve fahren.

```
this->model->GetJointController()->SetVelocityTarget(this->joint_left->GetScopedName(),
     10.0);
this->model->GetJointController()->SetVelocityTarget(this->joint_right->GetScopedName()
    , 5.0);
```

API für Gazebo

In einem nächsten Schritt wird die Möglichkeit geschaffen den Robi über Gazebo-Messages zu steuern. Dazu wird zusätzlich in der Load-Function ein Gazebo Node erstellt und jeweils ein Subscriber für jede Achse initialisiert. Zur Verdeutlichung, dies sind noch Gazebo-Messages bzw. Subscriber und somit noch nicht für ROS-Messages geeignet. Das Prinzip der Messages und Topics, über die kommuniziert wird, ist jedoch gleich wie in ROS.

```
1  this->node = transport::NodePtr(new transport::Node());
2  this->node->Init(this->model->GetWorld()->GetName());
3
4  // Create topic names
5  std::string topicNameLeft = "~/" + this->model->GetName() + "/robi_cmd_left";
6  std::string topicNameRight = "~/" + this->model->GetName() + "/robi_cmd_right";
7
8  // Subscribe to the topic, and register a callback
9  this->sub_left = this->node->Subscribe(topicNameLeft, &RobiPlugin::OnMsgLeft, this);
10 this->sub_right = this->node->Subscribe(topicNameRight, &RobiPlugin::OnMsgRight, this);
```

In Zeile 5 und 6 ist dabei auch zu sehen, dass die Topics initialisiert werden für die linke und rechte Achse. Zusätzlich muss noch für jeden Subscriber eine Callback-Function angegeben werden (*OnMsgLeft* und *OnMsgRight*). Diese wird aufgerufen, wenn neue Daten auf dem entsprechenden Topic empfangen wurden und zur Verarbeitung bereit sind.

Diese Callback-Functions sind in unserem Falle sehr einfach gehalten, die empfangene Geschwindigkeit wird einfach direkt für die Robi-Achse vorgegeben.

```
1  private: void OnMsgLeft(ConstVector3dPtr &msg_left)
2  {
3      this->SetVelocityLeft(msg_left->x());
4  }
```

Dabei ist SetVelocityLeft eine Helper-Function, welche die Geschwindigkeitsvorgabe vereinfacht:

```
1  public: void SetVelocityLeft(const double &vel_left)
2  {
3      // Set the joint's target velocity.
4      this->model->GetJointController()->SetVelocityTarget(this->joint_left->
           GetScopedName(),vel_left);
5  }
```

Darin wird dann wieder über den Pointer auf das Modell und den Joint Controller die Geschwindigkeit für den PID-Regler vorgegeben.

Im Kapitel 4.1.1 über das World-File wird noch auf das Setzen einer Initialgeschwindigkeit aus dem SDF-File eingegangen. Damit diese Geschwindigkeiten dem Modell übergeben werden können, müssen in der Load-Function nun noch für beide Achsen die folgenden Zeilen hinzugefügt werden:

```
double velocity_left = 0;
if(_sdf->HasElement("velocity_left")){
    velocity_left = _sdf->Get<double>("velocity_left");
}
```

API für ROS

Ungefähr gleich wie für Gazebo sieht auch die API für ROS aus. Denn auch mit ROS wird mit Messages über Topics kommuniziert. Dazu werden einfach andere Typen benötigt und noch ein paar zusätzliche Zeilen, damit die neuen Subscriber auch aus ROS angesprochen werden können. Über die Verbindung eines Gazebo-Modells mit ROS gibt es auch ein gutes Tutorial[3] auf dem Gazebo-Wiki als Fortsetzung zum oben erwähnten Tutorial über Plugins in Gazebo (siehe Fussnote 1).

Zuerst muss überprüft werden, ob ROS überhaupt bereits initialisiert wurde oder nicht:

```
// Initialize ros, if it has not already bee initialized.
if (!ros::isInitialized())
{
    int argc = 0;
    char **argv = NULL;
    ros::init(argc, argv, "gazebo_client",
    ros::init_options::NoSigintHandler);
}
```

Dann muss auch wie für die API für Gazebo ein Node initialisiert werden (nun ein ROS Node) und die Subscriber. Um eine bessere Übersicht in diesem Dokument zu halten, wird hier nur die Initialisierung für die linke Achse aufgeführt. Für die rechte Achse ist dies natürlich ebenfalls einzufügen und entsprechend anzupassen.

```
1 // Create our ROS node. This is similar to the Gazebo node
2 this->rosNode.reset(new ros::NodeHandle("gazebo_client"));
3
4 // Create a topic, and subscribe to it.
5 ros::SubscribeOptions so_left = ros::SubscribeOptions::create<std_msgs::Float32>("/" +
      this->model->GetName() + "/robi_cmd_left", 1, boost::bind(&RobiPlugin::
      OnRosMsgLeft, this, _1), ros::VoidPtr(), &this->rosQueue_left);
6 this->rosSub_left = this->rosNode->subscribe(so_left);
```

In Zeile 5 ist dabei die Initialisierung des Topics zu sehen, für den der Subscriber erstellt wird. Auch die Callback-Function muss wieder angegeben werden. Und zum Schluss muss auch noch beim ROS-Node registriert werden, dass dieser Node einen Subscriber mit bestimmten Eigenschaften enthält.

Dabei ist die Callback-Function für ROS in unserem Beispiel praktisch identisch mit der Callback-Function der API für Gazebo, einfach der Parameter ist nun eine Standard-Message aus ROS.

[3] http://gazebosim.org/tutorials?cat=guided_i&tut=guided_i6

```
public: void OnRosMsgLeft(const std_msgs::Float32ConstPtr &msg_left)
{
    this->SetVelocityLeft(msg_left->data);
}
```

Was es bei Verwendung von Komponenten aus ROS noch braucht ist das Starten eines Threads für die ROS Queues:

```
this->rosQueueThread = std::thread(std::bind(&RobiPlugin::QueueThread, this));
```

In diesem Queue-Thread werden dann die Queues der Subscriber auf ankommende Messages überprüft und in diesem Beispiel auch noch die aktuelle Robi-Position gepublisht:

```
private: void QueueThread()
{
    static const double timeout = 0.01;
    while (this->rosNode->ok())
    {
        this->rosQueue_left.callAvailable(ros::WallDuration(timeout));
        this->rosQueue_right.callAvailable(ros::WallDuration(timeout));
        gazebo::math::Pose pose = this->model->GetWorldPose();
        std_msgs::Float64 msg_x;
        msg_x.data = pose.pos.x;
        std_msgs::Float64 msg_y;
        msg_y.data = pose.pos.y;
        rosPub_pose_x.publish(msg_x);
        rosPub_pose_y.publish(msg_y);
    }
}
```

Mit `callAvailable` wird überprüft, ob bei einem Subscriber Messages angekommen sind, und falls ja, wird die entsprechende Callback-Function aufgerufen. Eine komplette Übersicht, was das Aufrufen der Callback-Functions auslösen kann, wie das Spinning in ROS angestossen werden kann usw. gibt es auf dem ROS-Wiki[4].

In den Zeilen 8 - 14 von obigem Listing ist zudem zu sehen wie die Position des Robis auf einem Topic gepublisht wird. Für die beiden Publisher braucht es auch noch eine Initialisierung in der Load-Function. Diese ist in den unten aufgeführten Zeilen zu sehen:

```
this->rosPub_pose_x = this->rosNode->advertise<std_msgs::Float64>("/" + this->model->
    GetName() + "/robi_pose/x", 1);
this->rosPub_pose_y = this->rosNode->advertise<std_msgs::Float64>("/" + this->model->
    GetName() + "/robi_pose/y", 1);
```

Oben gezeigte Initialisierung eines Publishers ist weniger aufwändig als die Initialisierung eines Subscribers. Dabei muss dem Node mit `advertise` bekannt gemacht werden, dass es einen Publisher auf einem bestimmten Topic gibt, dann ist der Publisher bereit. In diesem Beispiel ist der Topic für die Position `/robi_model/robi_pose/x` und `/y`, zusätzlich braucht es den Typen der Message (`std_msgs::Float64`). Danach müssen nur noch die Daten, wie im Queue-Thread gezeigt, auf dem entsprechenden Topic versendet werden.

[4] http://wiki.ros.org/roscpp/Overview/Callbacks%20and%20Spinning

4.2. Test mit Robi Modell

Damit sind die wichtigsten Komponenten des Plugins und Robotermodells soweit betrachtet. Zum Abschluss soll eine Variante zur Ansteuerung des Robis gezeigt werden. Der Robi kann mit diesem Plugin natürlich auch zum Beispiel über ein C++-Testprogramm angesteuert werden. Wie dabei auf einen Topic gepublisht wird, ist im Kapitel 3.3.4 über Topics beschrieben. Darauf wird hier jedoch verzichtet und das Modell des Robis aus *rqt* angesteuert. Das Tool *rqt* (ROS Qt[5]) ermöglicht es, aus einer graphischen Oberfläche Nachrichten zu publishen oder auch einen Plot von bestimmten Messages zu erstellen. Dieses Tool soll nachfolgend für die Ansteuerung unseres Robis verwendet werden und auch für die Darstellung der Geschwindigkeiten und Positionen verwendet werden.

Für den Test des Robis werden einige Tools und Befehle benötigt. Folgende Aufzählung soll einen Überblick verschaffen:

1. Aufstarten des ROS Masters in einem Terminal mit dem Befehl `$ roscore`
2. Start des Robis mit Gazebo in einem zweiten Terminal: `$ gazebo robi.world`
3. Im dritten Terminal ROS Qt GUI starten: `$ rqt`

Abbildung 4.1 zeigt das Tool *rqt* mit den Plugins *Plot* (links) und *Message Publisher* (rechts). Dies kann als gute Ausgangslage für erste Versuche mit *rqt* für die Ansteuerung des Robis verwendet werden. Es gibt noch diverse andere Plugins, zum Beispiel zur Betrachtung eines Kamera-/Sensorbildes, genaueres dazu ist jedoch auf dem ROS-Wiki (siehe Fussnote 5) zu finden und wird hier nicht genauer betrachtet.

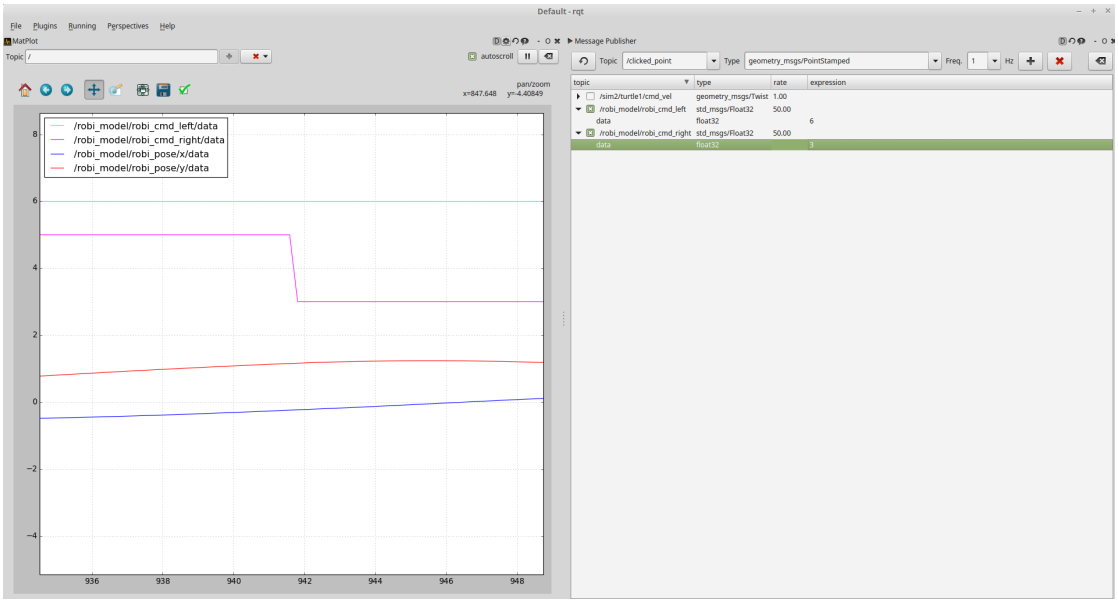

Abbildung 4.1.: Plot und Control von Robi mit *rqt*

[5] http://wiki.ros.org/rqt

Wenn man dieses Bild genauer betrachtet, sieht der Message Publisher wie folgt aus:

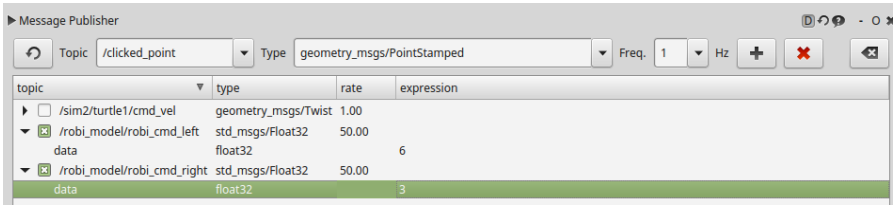

Abbildung 4.2.: Message Publisher mit *rqt*

Dabei kann angegeben werden, mit welcher Updaterate die entsprechende Message auf dem Topic versendet werden soll und unter dem Punkt *data* kann dann der gewünschte Wert für die Achse angegeben bzw. eingetragen werden.

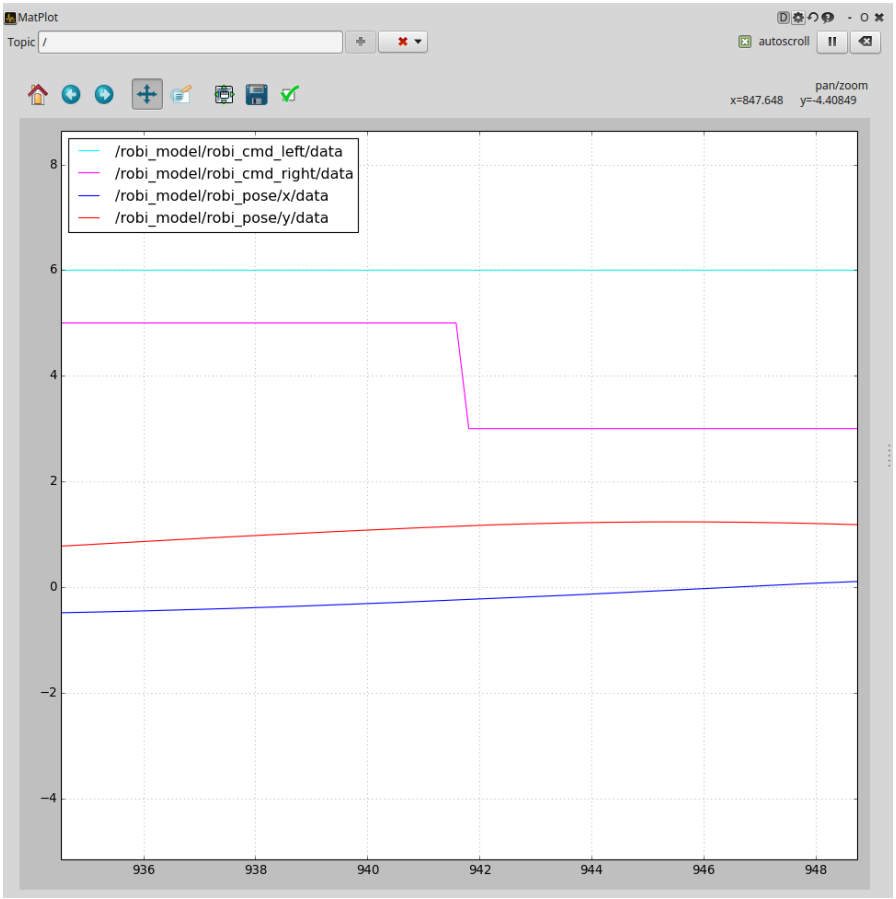

Abbildung 4.3.: Plot mit *rqt*

Auf der linken Seite ist der Plot dargestellt, dabei können mehrere Topics im gleichen Scope betrachtet werden. In nachfolgendem Bild sind die beiden Geschwindigkeiten, die vorgegeben werden, dargestellt und die Position des Roboters. In diesem Fall ist zum Beispiel ein Sprung in der Geschwindigkeit des rechten Rades zu sehen. Anschliessend reagiert die Position (rote und blaue Linie) auf die Geschwindigkeitsänderung der Räder. Am Besten wird dies in einem eigenen Versuch ausprobiert, dann kann der User intuitiv die Position im Gazebo GUI mit dem

Scope vergleichen, per Text und Bild ist dies weniger anschaulich. Die verschiedenen Topics werden mit einer Farbe markiert und mit der Update-Rate, mit der sie gepublisht werden, im Plot dargestellt.

Einen Überblick über die vorhandenen Topics und Nodes kann man sich wieder mit dem Tool *rqt_graph* verschaffen. Dieser Graph ist in Abbildung 4.4 zu sehen. Dabei sind die beiden Publisher für die Position des Robis und die beiden Subscriber in Gazebo für die Achsgeschwindigkeiten sichtbar.

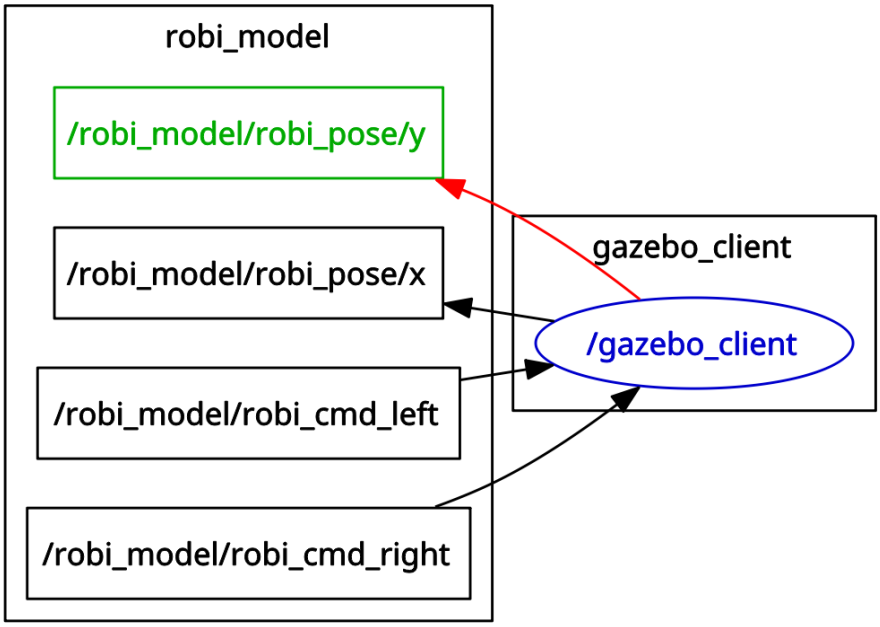

Abbildung 4.4.: *rqt_graph* von Robi

Damit ist der Robi grundsätzlich bereit und kann nun je nach Bedarf weiter ausgebaut werden. Zum Beispiel könnten dem Robi die Infrarotsensoren, die er im Realen besitzt, nachgebildet werden und hinzugefügt werden. Dann könnte auch für die Simulation ein Programm geschrieben werden, dass Hindernissen ausgewichen wird usw.

5. Ergebnisse und Ausblick

5.1. Schlussfolgerung

In diesem Simulationsprojekt konnte einiges an Know-how über ROS und Gazebo erarbeitet werden. Mit dieser Dokumentation soll der Einstieg für neue ROS und Gazebo Nutzer etwas vereinfacht werden, beziehungsweise eine Übersicht geschaffen werden, was ROS und Gazebo ausmacht. Bei spezifischen Fragen bietet das ROS- und Gazebo-Wiki genügend Dokumentation. Dieses Dokument soll ermöglichen, dass ein User nach den richtigen Begriffen suchen kann, wenn er die beiden Tools noch nicht kennt.

Während dieses Simulationsprojektes konnte ein einfacher, zweirädriger Robi erstellt werden, wie er im Bachelorstudium im Informatikunterricht real verwendet wird. So konnte die Erstellung eines Modells betrachtet werden und auch wie dieses dann in Gazebo mit ansteuerbaren Achsen versehen wird. Weiter war es möglich zu zeigen, wie diese Achsen in Gazebo über ein Plugin mit ROS-Tools angesteuert werden können.

Es konnte eine Übersicht gewonnen werden über ROS und Gazebo, um diskutieren zu können, wo und wie unser Robotikframework EEROS sinnvoll mit ROS verknüpft werden könnte. Dabei zeigte sich auch, dass ROS zwar für gewisse Fragestellungen eine Lösung bereit hält, aber zum Beispiel für die Regelung eines Antriebsmotors auf dem aktuellen Stand kaum geeignet ist. Da würde unser Framework ein Konzept und die nötigen Tools liefern.

5.2. Ausblick

Der in diesem Projekt erstellte Robi ist noch nicht komplett modelliert und angesteuert. ROS und Gazebo bieten noch viele Möglichkeiten, um den Robi zum Beispiel mit Sensoren usw. auszustatten. Dies könnte zum Beispiel in eine weitere Arbeit einfliessen. Dazu könnten für das Robi-Modell Reflexlichtschranken modelliert werden, wie sie das reale Vorbild besitzt und mit diesen dann Hindernissen, die im Gazebo-GUI aufgestellt werden, ausgewichen werden. Auch könnten zum Beispiel für den Robi neue Konzepte zur Hinderniserkennung ausprobiert werden, zum Beispiel mit einem Laserscanner. Ein solcher wird zum Beispiel in einem Gazebo-Tutorial[1] erstellt (siehe Fussnote). Auch könnte die Simulation mit dem Original verglichen werden, und vielleicht sogar mal versucht werden eine komplette Robi-Steuerung in ROS zu implementieren. Es existieren also noch viele Möglichkeiten diesen Robi zu verfeinern und weitere Features von ROS und Gazebo auszuprobieren.

[1] http://gazebosim.org/tutorials?cat=guided_i&tut=guided_i1

Literaturverzeichnis

[ROS2016] ROS Homepage: *Allgemeine Infos zu ROS, siehe Fussnoten URL:http://www.ros.org [Stand 30.08.2016]*

[ROW2016] ROS Wiki: *Diverse Artikel, Details siehe Fussnoten in Dokument URL:http://wiki.ros.org [Stand 30.08.2016]*

[GAZ2016] Gazebo Homepage: *Tutorials und Informationen, siehe Fussnoten Dokument URL:http://gazebosim.org/ [Stand 30.08.2016]*

[BIB2016] Bitbucket - OpenSourceRoboticsFoundation (OSRF): *Source-Repositories und Packages von Gazebo und Tutorials etc. URL:https://bitbucket.org/osrf/ [Stand 30.08.2016]*

[AMA2016] Amazonaws - OpenSourceRoboticsFoundation (OSRF): *Gazebo API und Dokumentation URL:http://osrf-distributions.s3.amazonaws.com/gazebo/api/dev/index.html [Stand 30.08.2016]*

[SOL2016] Solidworks Blog: *Modeling for Gazebo URL:http://blogs.solidworks.com/teacher/wp-content/uploads/sites/3/WPI-Robotics-SolidWorks-to-Gazebo.pdf [Stand 30.08.2016]*

[ARB2016] Arnaud Bertrand: *ROS Gazebo Tutorial: The Definitive Crash Course URL:http://arnaudbertrand.io/blog/2015/06/26/ros-gazebo-tutorial-the-definitive-crash-course [Stand 30.08.2016]*

[DAN2016] Danping Zou, Huizhong Zhou: *Install Mint17 + ROS Indigo + Gazebo3 + Qt5.3.0 URL:http://mediasoc.sjtu.edu.cn/wordpress/wp-content/uploads/2013/11/InstallaltionROS+Gazebo+Qt_0722.pdf [Stand 30.08.2016]*

[GIT2016] Github - Simulation Tools in ROS: *Gazebo-ROS-Packages URL:https://github.com/ros-simulation/gazebo_ros_pkgs [Stand 30.08.2016]*

[WIK2016] Wikipedia: *Artikel Publish Subscribe usw., siehe Fussnoten URL:http://wikipedia.org/ [Stand 30.08.2016]*

A. Installationsanleitung ROS und Gazebo

ROS und Gazebo sind eng verknüpfte Programme. In den neusten Versionen ist die Trennung zwar grösser und für den User einfacher geworden. Es empfiehlt sich jedoch trotzdem, sich vorher darüber zu informieren, welche ROS Version mit welcher Gazebo Version arbeiten kann. Eine Übersicht dazu ist auf dem Gazebo-Wiki[1] zu finden.

In diesem Projekt wurde mit ROS Jade und Gazebo Version 5.3 gearbeitet und dies hat sich bewährt. Zuerst wurde ROS Indigo mit Gazebo 2 eingesetzt, aber da zeigte sich, dass viele aktuelle Features noch fehlen, die in Tutorials usw. verwendet werden. Zum Beispiel die PID-Regler in Gazebo 5 haben eine bessere API bekommen, welche in Gazebo 2 noch umständlicher war.

Für die Installation wird auf das ROS-Wiki[2] verwiesen. Für Gazebo ist ebenfalls auf dem Wiki[3] eine Anleitung zu finden. Für Ubuntu gibt es dabei für beide Tools fertige Packages und so können diese einfach mit *apt-get install* installiert werden.

Wenn ROS und Gazebo zusammen verwendet werden möchten sind noch die Gazebo-ROS Packages nötig. Diese sind ebenfalls als Ubuntu Package verfügbar, zumindest für gewisse Kombinationen von ROS und Gazebo:

```
$ sudo apt-get install ros-jade-gazebo-ros-pkgs
```

Falls kein fertiges Package zur Verfügung steht, können die Tools natürlich auch aus den Sourcen gebuildet werden. Anleitungen dazu sind auf den ROS- und Gazebo-Wikis zu finden.

Linux Mint

Für dieses Projekt wurde Linux Mint als Betriebssystem verwendet. Dies erfordert etwas Zusatzaufwand im Vergleich zur Verwendung eines Standard-Ubuntus. Unter Linux Mint können zwar die Ubuntu Packages von ROS und Gazebo installiert werden. In den Befehlen zur Installation wird dabei aber der Distributions-Name direkt aus dem System gelesen und so entsprechend das richtige Package ausgewählt. Für Linux Mint gibt es jedoch kein entsprechendes Repository, aber es kann einfach das von der entsprechenden Ubuntu Version ausgewählt werden. Dabei muss folgende Zeile:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /
    etc/apt/sources.list.d/ros-latest.list'
```

[1] http://gazebosim.org/tutorials?tut=ros_wrapper_versions
[2] http://wiki.ros.org/jade/Installation/Ubuntu
[3] http://gazebosim.org/tutorials?tut=install_ubuntu

ersetzt werden durch:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu trusty main" > /etc/apt/
    sources.list.d/ros-latest.list'
```

Dies kann auf alle Befehle übertragen werden, bei denen der Codename der Distribution ausgelesen wird. Dies funktioniert auf Linux Mint 17 mit den Packages von Ubuntu 14.04 (Code name *trusty*), da Linux Mint dabei die gleiche Package Base besitzt. Auf dem Linux Mint-Wiki[4] ist eine Aufstellung zu finden (siehe Fussnote 4), welche Mint Version welche Ubuntu Package Base besitzt. So können auch für andere Versionen von Linux Mint die entsprechenden Ubuntu Packages installiert werden.

[4] https://www.linuxmint.com/download_all.php

B. Nützliche Befehle für ROS

Befehl	Beschreibung
roscore	startet den ROS Master, muss immer zuerst gestartet werden
rosrun packageName nodeName	startet einen ROS Node
roslaunch filename.launch	führt ein Launch-File aus, der ROS Master muss dabei nicht vorher gestartet werden
rqt	graphisches Tool mit Scope, Möglichkeit Messages zu publishen etc.
rostopic list	listet alle aktiven Topics auf
rostopic echo /topic/name	schreibt Messages von gewähltem Topic auf Konsole
rosnode list	listet alle aktiven Nodes auf
rosbag record /topic/name	speichert Messages von Topic, ROS Bags können wieder abgespielt werden

C. Eigenes ROS Package erstellen

Nachfolgendes Kapitel soll die wichtigsten Punkte zusammenfassen, wie ein eigenes ROS Package erstellt wird. Eine ausführliche Variante, mit allen Optionen, ist auf dem ROS-Wiki[1] zu finden. Das Ziel dieser Anleitung ist auf die wichtigsten Punkte einzugehen.

Die Source-Files werden im Catkin-Workspace im Unterordner *src* abgelegt. Genaueres zur File Organisation, was ein Package alles enthalten muss, ist in Kapitel 3.2.3 Dateiorganisation genauer erläutert. Wenn nun ein neues Package angelegt wird, wird im *src* Ordner ein Unterordner mit dem Packagenamen angelegt und allen nötigen Files erstellt (CMakeLists.txt und package.xml). Dies erfolgt als ersten Schritt mit folgendem Befehl:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg <pkg_name> [depend1][depend2][depend3][...]
```

Dabei müssen auch noch die Dependencies des Packages angegeben werden. Dies fügt dann bereits im CMake-File entsprechende Einträge hinzu, welche nicht mehr manuell hinzugefügt werden müssen. Zum Beispiel, wenn eine Abhängigkeit zu Standard-Messages (*std_msgs*), C++ (*roscpp*) und Python (*rospy*) vorhanden ist, sieht der Befehl wie folgt aus:

```
$ catkin_create_pkg first_example_pkg std_msgs roscpp rospy
```

Dies wird wie oben erwähnt den benötigten Ordner mit dem CMakeLists- und Package-File erzeugen.

In einem zweiten Schritt werden, falls dies noch nötig sein sollte, im File *package.xml* die build- und run-Abhängigkeiten angepasst. Diese werden bereits vorkonfiguriert, wenn beim Befehl `catkin_create_pkg` alle Abhängigkeiten angegeben wurden, ansonsten können sie hier noch ergänzt oder angepasst werden. Es gibt gegen Ende des Files Zeilen, die wie folgt aussehen:

```
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<run_depend>roscpp</run_depend>
<run_depend>rospy</run_depend>
<run_depend>std_msgs</run_depend>
```

Dabei wird jede Komponente aufgeführt, die benötigt wird. Die <*build_depend*> Tags geben an, welche Komponenten zur Compilierzeit benötigt werden und <*run_depend*> welche Komponenten zur Laufzeit nötig sind.

[1] http://wiki.ros.org/ROS/Tutorials/catkin/CreatingPackage

Nachdem dies im Package-File erledigt ist, müssen diese Abhängigkeiten auch noch im CMake-File angegeben bzw. geändert werden. Dabei müssen für die Abhängigkeiten die entsprechenden Packages in CMake gefunden werden. Dies erfolgt mit folgenden Zeilen:

```
find_package(catkin REQUIRED COMPONENTS
    roscpp
    rospy
    std_msgs
)
```

Anschliessend müssen im File weiter unten die Zeilen einkommentiert werden, bei denen das Executable erstellt und gelinkt wird. Die sind bereits vorbereitet und müssen nur noch mit den korrekten Files und Targets ergänzt werden.

```
## Declare a C++ executable
add_executable(talker src/talker.cpp)
add_executable(listener src/listener.cpp)
```

Oben die build-Targets und nachfolgend die link-Targets:

```
## Specify libraries to link a library or executable target against
target_link_libraries(talker
   ${catkin_LIBRARIES}
)
target_link_libraries(listener
   ${catkin_LIBRARIES}
)
```

Anschliessend ist das eigene Package soweit vorkonfiguriert, dass es erstellt werden kann. Dabei wird, wie in Kapitel 3.2.3 `catkin_make` verwendet, um das Package zu builden.

Falls für das eigene Package auch noch ein eigener Message- oder Service-Typ erstellt wird, können diese Definitions-Files jeweils in einem *msgs* oder *srv* Ordner abgelegt werden und im CMake File angegeben werden. Wie dies gemacht wird, ist in einem ROS Tutorial[2] gut beschrieben.

Im *package.xml* müssen zwei Zeilen hinzugefügt werden, damit die Messages generiert werden:

```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

Und im CMakeLists.txt müssen die Teile einkommentiert werden, welche für die Message-Generation vorgesehen sind.

```
## Generate services in the 'srv' folder
add_service_files(
    FILES
    AddTwoInts.srv
)

## Generate added messages and services with any dependencies listed here
```

[2] http://wiki.ros.org/ROS/Tutorials/CreatingMsgAndSrv#Creating_a_srv

```
8  generate_messages(
9     DEPENDENCIES
10    std_msgs  # Or other packages containing msgs
11 )
```

Im `find_package` Teil muss zusätzlich noch `message_generation` hinzugefügt werden. Danach kann das Package wieder mit `catkin_make install` erzeugt und diesmal auch installiert werden. Dann finden sich nämlich die nötigen Header-Files, die für die neu erstellten Messages benötigt werden, im *devel* Ordner unter */include/package_name*.