Institut für Robotik und Prozessinformatik

Technische Universität Braunschweig

Prof. Dr. J. Steil

ROBOTIKPRAKTIKUM

Versuch 4

ROS - Robot Operating System

Inhaltsverzeichnis

Kapitel 1

Einleitung

Aktuelle Robotersysteme bestehen typischerweise neben dem eigentlichen Manipulator aus zahlreichen zusätzlichen Hardwarekomponenten. Hinzu kommen verschiedene Softwaremodule für unterschiedliche Aufgaben. Eine beispielhafte Menge an Komponenten, die über mehrere Steuerungscomputer verteilt sind, ist in der Abbildung 1.1 dargestellt.

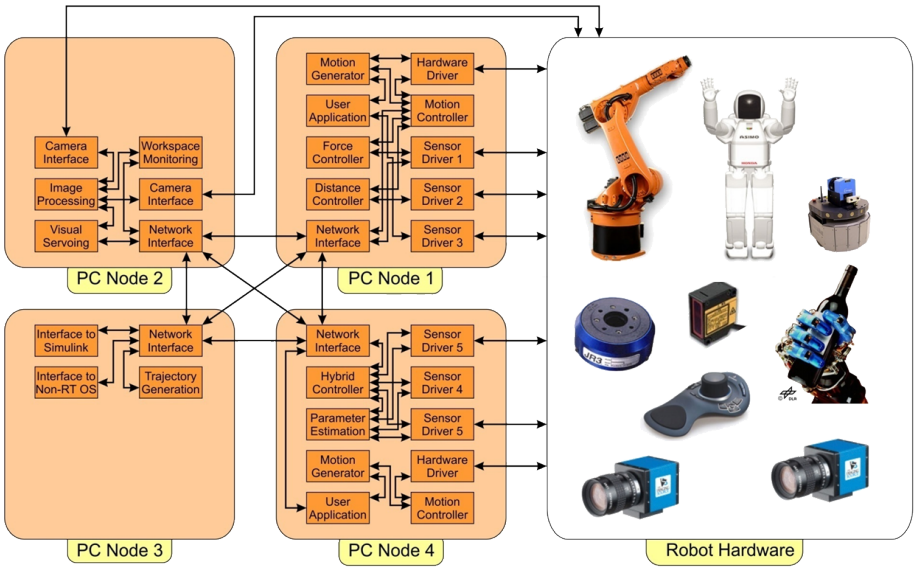


Abbildung 1.1: Übersicht über typische Komponenten eines Robotersystems

In solch einem System wird typischerweise eine Middleware eingesetzt, um zum Beispiel die Kommunikation der einzelnen Softwaremodule untereinander und über Computergrenzen hinweg zu gewährleisten. Außerdem bringt der Einsatz einer Middleware den Vorteil mit sich, dass sie eine Hardwareabstraktion darstellt und somit der Portierungsaufwand beim Austausch von Hardwarekomponenten minimiert werden kann.

Im industriellen Umfeld wird als Middleware zum Beispiel die echtzeitfähige Lösung *RTI Connext DDS*[1] eingesetzt. Im akademischen Bereich und in der industriellen Forschung wird hingegen häufig auf das *Robot Operating System* kurz ROS[2] gesetzt.

[1] https://www.rti.com/products/dds/ Stand: 23.11.2021
[2] http://www.ros.org/ Stand: 23.11.2021

Durch eine breite Hardwareunterstützung, ein großes Repository mit Implementierungen verschiedenster Algorithmen, einer aktiven Community und einer guten Dokumentation konnte *ROS* eine weite Verbreitung in der Forschung erreichen. Zwar wird die Möglichkeit eines schnellen Entwurfs von Prototypen geboten, jedoch wird nicht die Echtzeitfähigkeit anderer Lösungen erreicht.

Um einen Einblick in *ROS* zu erhalten, sind im Rahmen dieses Versuches unterschiedliche Aufgaben zu lösen, sodass einige Eigenschaften und Funktionen kennen gelernt sowie angewendet werden. Der Schwerpunkt liegt hierbei auf dem modularen Aufbau eines Systems, der Kommunikation der verschiedenen Module untereinander, der Integration von Sensoren und Eingabegeräten sowie der Auswertung und Visualisierung von Messdaten.

Kapitel 2

ROS - Robot Operating System

Dieses Kapitel stellt das *Robot Operating System*, im Folgenden nur als ROS bezeichnet, vor. Der Fokus liegt hierbei auf der Beschreibung der charakteristischen Eigenschaften sowie der verwendeten Konzepte[1]. Befehle, die für den praktischen Umgang mit ROS und der Versuchsdurchführung benötigt werden, sind nicht Teil dieses Kapitels sondern stattdessen im Anhang „ROS Noetic Cheatsheet“ zusammengefasst. Besteht über das Praktikum hinaus Interesse an ROS, so sei auf das online[2] verfügbare Buch *A Gentle Introduction to ROS* von *Jason M. O'Kane* verwiesen.

2.1 Konzepte von ROS im Detail

Die Vielzahl an kommerziell verfügbarer Hardware (Manipulatoren sowie Sensoren) erschwert es für kleine Forschungsgruppen nur selbst entwickelte und für die eigene/verfügbare Hardware angepasste Software zu verwenden. Eine Hardwareänderung bedeutet in diesem Fall einen hohen Integrationsaufwand und somit kann es vorkommen, dass die verwendete Hardware nicht auf die aktuellen Forschungsprojekte zugeschnitten ist, da der Integrationsaufwand für die optimale Hardware gescheut wird.

In der weltweiten Robotikcommunity existieren Implementierungen verschiedener Algorithmen unterschiedlicher Forschungsgruppen zu typischen Problemstellungen in der Robotik. Um diese Algorithmen nutzen zu können, ohne sie erneut implementieren zu müssen, werden gleiche Interfaces, Konventionen und Softwarearchitekturen benötigt.

Die angesprochenen Punkte verdeutlichen, welche Probleme mit der Erstellung komplexer, robuster und vielseitig einsetzbarer Software für den Einsatz in der Robotik einher gehen und zu lösen sind. Erst dann können auch für den Menschen sehr einfache Aufgaben erfolgreich unter Verwendung von Robotern gelöst werden, obwohl es bei jeder Anwendung zu Variationen von Objekten, Parametern oder der Umgebung kommt. Um dennoch für einfache Aufgaben roboterbasierte Lösungen anbieten zu können, sind einzelnen Forschungsgruppen auf die von der Community entwickelten Einzellösungen, beispielsweise für die Handhabung der genannten Variation angewie-

[1] http://wiki.ros.org/ROS/Concepts Stand: 23.11.2021
[2] https://cse.sc.edu/~jokane/agitr/agitr-letter.pdf Stand: 23.11.2021

sen. Denn ohne diese existierenden Einzellösungen zu verwenden, kann der Aufwand die vorhandenen Mittel überschreiten.

Die sogenannte ROS-Gleichung, dargestellt in Abbildung 2.1, adressiert diese genannten Bedürfnisse und beschreibt damit das Zusammenspiel der Community beim kollaborativen Erstellen von effektiven und robotikspezifischen Softwarelösungen. Die Gleichung stellt die Verbindung einer Anwendergemeinschaft mit verfügbaren Gerätetreibern und Algorithmen sowie einer Toolchain zur Erfassung, Auswertung und Visualisierung von Messdaten in einer einheitlichen Softwarearchitektur dar. Zusätzlich ermöglicht der modulare Aufbau von ROS, dass einzelne Teile aus der verfügbaren Menge an Software in Kombination mit der minimalen Infrastruktur beliebig zu einem Gesamtsystem zusammengestellt werden können. Dies ist einer der Gründe, aus denen ROS einen hohen Verbreitungsgrad in der Robotik erlangt hat.

Abbildung 2.1: Die ROS-Gleichung [https://www.ros.org/blog/ecosystem/, Stand: 23.11.2021]

Die Entwicklung von ROS begann im Jahr 2007 an der Stanford University. Anschließend erfolgte die Weiterentwicklung unter der Schirmherrschaft von *Willow Garage*[3]. Im April 2012 wurde die gemeinnützige Organisation *Open Source Robotics Foundation (OSRF)*[4] gegründet, welche die Entwicklung seit dem fortführt.

Aktuell werden die Kernkomponenten von ROS in die Kategorien *Kommunikationsinfrastruktur*, *robotikspezifische Features*, *Diagnose*, *Lokalisieren, Navigation und Posenbestimmung* und *Werkzeuge* unterteilt. Somit bietet ROS mehr Funktionalität als eine klassische Middleware wie zum Beispiel *RTI Connext DDS* und wird deshalb auch als Meta-Betriebssystem bezeichnet. Im Folgenden werden Elemente der einzelnen Kernkomponenten in Form einer Aufzählung übersichtlich dargestellt, bevor im Abschnitt 2.2 die ROS-interne Kommunikation näher betrachtet wird.

Kommunikationsinfrastruktur:

- Message Passing
- Recording and Playback of Messages
- Remote Procedure Call
- Distributed Parameter System

robotikspezifische Feature:

- Standard Robot Messages
- Robot Geometry Library
- Robot Description Language

[3] https://en.wikipedia.org/wiki/Willow_Garage Stand: 23.11.2021
[4] http://www.osrfoundation.org/ Stand: 23.11.2021

- Preemptable Remote Procedure Calls

Diagnose

Lokalisieren, Navigation und Posenbestimmung

Werkzeuge:

- Command-Line Tools
- rviz
- rqt

Detaillierte Informationen zu den einzelnen Punkten sind Stand 23.11.2021 unter http://wiki.ros.org/Tools zu finden.

Abschließend ist anzumerken, dass ROS auf einer sprach-neutralen Architektur basiert, sodass die Anwender ihre Software in der von ihnen favorisierten Programmiersprache erstellen können.

2.2 Der Kommunikationsgraph

Der Kommunikationsgraph von ROS repräsentiert eine Kommunikationsarchitektur, welche die Kommunikationen zwischen Prozessen in synchroner als auch in asynchroner Form gewährleistet. Dabei ist es nicht von Bedeutung, ob die Prozesse auf dem selben Computer laufen oder über mehrere Computer im Netzwerk verteilt sind. Da der Kommunikationsgraph ein zentrales Element dieses Versuches ist, werden zunächst seine Elemente vorgestellt und danach die synchrone und asynchrone Kommunikation näher beschrieben.

- **ROS-Master:** Der ROS-Master ist für die Registrierung und die Verwaltung aller verfügbaren Services und Nodes zuständig. Er sorgt dafür, dass eine Kommunikation aufgebaut werden kann und ist somit von seiner Funktion her mit einem DNS-Server für die Namensauflösung vergleichbar.
- **Node:** Ein Node ist ein eigenständiger Prozess, welcher Berechnungen ausführt und innerhalb des Kommunikationsgraphs als Quelle oder Senke für Nachrichten fungiert.
- **Parameterserver:** Ein Parameterserver ist ein geteiltes Verzeichnis, welches globale Parameter (zum Beispiel Konfigurations– und Initialisierungsparameter) enthält und für alle anderen Elemente des Kommunikationsgraphen zugänglich ist.
- **Message:** Nodes kommunizieren über Nachrichten, welche über ein spezifisches Topic ausgetauscht werden.
- **Topic:** Ein Topic ist ein Kanal mit einem spezifischen Namen über den Nodes Nachrichten austauschen. Sie sind für einen kontinuierlichen Datenfluss, wie zum Beispiel beim Streamen von Sensorwerten oder des Roboterzustands, gedacht. Quellen können Nachrichten senden, auch wenn keine Senken mit dem Topic verbunden sind und außerdem können mehrere Senken mit einer Quelle über das gleiche Topic verbunden sein. Eine Quelle verschickt eine Nachricht zu

einem selbstgewählten Zeitpunkt und die Senke erhält die Nachricht sobald sie verfügbar ist, da Topics für asynchrone[5] Kommunikation verwendet werden.

- **Service:** Ein Service eignet sich zur synchronen[6] Kommunikation, da er einen blockierenden Aufruf eines Dienstes darstellt. Typischer Weise wird ein Service für *remote procedure calls*, wie der Abfrage eines aktuellen Zustands, spezifischer Werte oder der schnellen Berechnung der inversen Kinematik verwendet. Im Allgemeinen sollten Services nicht für länger laufende Prozesse verwendet werden, da ansonsten unerwartete Seiteneffekte im restlichen Kommunikationsgraphen, ausgelöst durch das Blockieren, auftreten können. Die Kommunikation verläuft immer zwischen exakt einer Quelle und einer oder mehreren Senken.
- **Bags:** Bei einem Bag handelt es sich um ein Dateiformat, mit welchem sich der Kommunikationsgraph abbilden, speichern und zu einem späteren Zeitpunkt mit dem richtigen Zeitverhalten wiedergeben lässt.

2.2.1 Asynchrone Kommunikation

Die asynchrone Kommunikation zwischen Nodes basiert auf Nachrichten, die zu einem bestimmten Topic verschickt oder empfangen werden. Um die Kommunikation aufzubauen, muss die Quelle ihre Topics bei dem ROS-Master ankündigen. Eine Senke fragt bei dem ROS-Master ab, welche Topics verfügbar sind. Anschließend findet die Kommunikation direkt zwischen den Nodes statt. Dieses Verhalten ist beispielhaft in Abbildung 2.2 skizziert. Auf einem Topic können Nachrichten von mehreren Quellen verschickt und ebenso von mehreren Senken empfangen werden. Dabei ist die Verwendung von Standardnachrichtentypen als auch von benutzerdefinierten Nachrichten möglich.

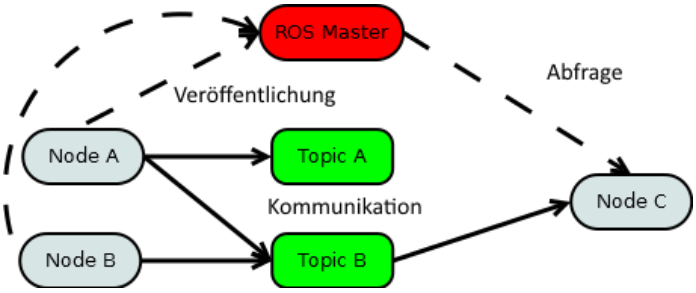


Abbildung 2.2: Asynchrone Kommunikation in einem ROS-System bestehend aus einem ROS Master und drei Nodes unter Verwendung von zwei Topics

2.2.2 Synchrone Kommunikation

Die Kommunikation über einen Service besteht aus einem Nachrichtenpaar. Eine Nachricht dient der Anfrage/Anforderung und die andere Nachricht enthält die Antwort. Der Service selber wird von einem Node unter einem Namen, als String formatiert, angeboten. Die Clients senden die Anforderungen an diesen Service und sind solange blockiert, bis sie eine Antwort erhalten haben. Für den Programmierer wirkt dieses Verhalten als würde ein *remote procedure call* verwendet werden. Um eine

[5] siehe Abschnitt 2.2.1
[6] siehe Abschnitt 2.2.2

höhere Performance zu erreichen, kann ein Client eine persistente Verbindung zu einem Service aufbauen. Dieser Fall ist jedoch nicht so robust gegenüber Änderungen im Aufbau des Kommunikationsgraphen und besagte Änderungen sind gerade der Vorteil der Node-basierten und dynamisch erstellbaren Architektur des Gesamtsystems.

Kapitel 3

Versuchsaufbau

Um die ROS-Konzepte näher zu bringen, wird in diesem Versuch der mobile Roboter Pioneer P3-DX[1] verwendet. Auf diesem ist ein SICK Laserscanner[2] montiert (Abbildung 3.1). Zur Steuerung sind beide Geräte über USB-RS232-Adapter an einen Raspberry Pi 2 angeschlossen.

Abbildung 3.1: Pioneer P3-DX mit SICK Laserscanner und Raspberry Pi

Zur Bearbeitung der Aufgabenstellungen wird ein PC, auf dem Ubuntu 20.04 mit Long-Term-Support installiert ist, verwendet. Dieser PC und der Raspberry Pi sind über ein eigenes WLAN miteinander verbunden und dank der Netzwerkfähigkeit von ROS können einige Komponenten des ROS-System auf dem PC und andere auf dem

[1] https://www.generationrobots.com/media/Pioneer3DX-P3DX-RevA.pdf Stand 23.11.2021
[2] http://sicktoolbox.sourceforge.net/docs/sick-lms-technical-description.pdf Stand 23.11.2021

Raspberry Pi starten ohne manuell per SSH darauf zugreifen zu müssen (siehe Abschnitt 4.5). Zur Steuerung des Pioneers ist ein kabelloses Gamepad über einen USB-Funk-Stick mit dem PC verbunden. Der gesamte Versuchsaufbau ist in dem Deploymentdiagramm in Abbildung 3.2 dargestellt.

Die derzeit aktuellste Version von ROS ist Noetic Ninjemys [3]. ROS Noetic ist sowohl auf dem PC als auch auf dem Pi bereits installiert.

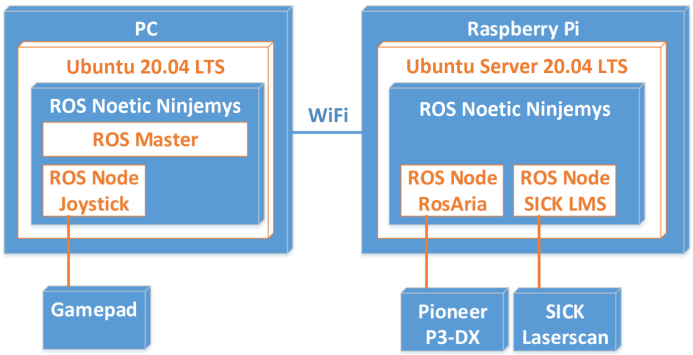


Abbildung 3.2: Deploymentdiagramm des Versuchaufbaus

[3] http://wiki.ros.org/noetic Stand 23.11.2021

Kapitel 4

Versuchsdurchführung und Aufgaben

Das Ziel diese Versuchs besteht in der Implementierung eines Systems zum Verfahren des Pinoeers unter Verwendung eines Gamepads. Als Infrastruktur ist hierfür eine bereits existierendes ROS-Setup zu verwenden. Darüber hinaus werden die Sonar-Sensoren[1] zur Hinderniserkennung genutzt und abschließend das Labor mit Hilfe des auf dem Roboter montierten Laserscanners kartografiert.

In diesem Kapitel werden schrittweise die wichtigsten ROS-Komponenten sowie die dazugehörigen Kommandozeilen-Befehle erläutert. Wie die jeweiligen Befehle und Strukturen konkret anzuwenden sind, entnehmen Sie dem beigefügten ROS Cheat Sheet. Detaillierte Informationen finden Sie unter den dazugehörigen Links auf http://wiki.ros.org.

4.1 Vorbereitung

Auf dem Praktikumsrechner wurde bereits eine Noetic-Distribution von ROS installiert, diese befindet sich in dem Verzeichnis `/opt/ros/noetic/` und bietet eine Vielzahl an Kommandozeilen-Befehlen. Damit Ihnen diese zur Verfügung stehen, müssen Sie zunächst das Setup-Skript als Quelle für Ihre Shell hinzufügen. Führen Sie folgende Befehle aus:

```
echo "source /opt/ros/noetic/setup.bash" >> .bashrc
source ~/.bashrc
```

Damit wird das Setup in jedem Terminal, das Sie öffnen geladen. Wenn Sie nun in einem Terminal `ros` eingeben und dann zwei mal TAB drücken, werden Ihnen alle verfügbaren ROS-Befehle angezeigt.

[1] Ortung von Gegenständen mittels ausgesandter Schallimpulse

4.2 ROS Master und Packages

Für jedes ROS System muss zunächst ein Master gestartet werden. Die Aufgabe des Masters besteht darin, ROS Nodes und Topics (siehe Abschnitt 2.2) zu registrieren und zu benennen. Er initialisiert Peer-to-Peer-Verbindungen zwischen Nodes, damit diese miteinander kommunizieren können.

ROS Software ist in Packages organisiert. Ein Package kann ROS Nodes, unabhängige Bibliotheken, Konfigurations-Dateien oder vieles mehr enthalten, das zusammen ein nützliches Modul ergibt. Packages dienen dazu Funktionalität einfach und modular bereit zu stellen, so dass sie für verschiedene Aufgaben wiederverwendet werden können.

ROS Befehle:

- `roscore` - Startet den ROS Master und den Parameter-Server
- `rospack` - Zeigt Informationen zu installierten Packages an

Aufgabe:

- Starten Sie den ROS Master!

Im Terminal sollten Sie nun die Ausgaben sehen, dass der Master erfolgreich gestartet wurde. Für die weiteren Arbeiten mit ROS muss der Master geöffnet bleiben. In dem Terminal können aber nur Ausgaben betrachtet werden, das heißt für weitere ROS-Software benötigen Sie jeweils ein neues Terminal, welches sich in der Ubuntu-Konsole durch Drücken von `Strg + Shift + T` öffnen lässt. Sollten Sie den Master (oder einen anderen Node) beenden wollen, erreichen Sie das durch Drücken von `Ctrl + C`.

4.3 Nodes, Topics und Messages

ROS Nodes sind nicht viel mehr als ausführbare Dateien in ROS, welche in den Programmiersprachen C++ oder Python implementiert werden können. Nodes kommunizieren mit anderen Nodes indem sie Messages über ROS Topics senden und empfangen. Topics sind Busse, die über ihren Namen eindeutig identifizierbar sind und denen ein eindeutiger Message-Typ zugeordnet ist. über Topics können Nodes miteinander Messages austauschen. Nodes, die Daten über ein Topic senden nennt man Publisher, Daten empfangende Nodes sind Subscriber. Beim Starten eines Nodes meldet sich dieser mit seinem Namen am ROS Master an und informiert diesen außerdem darüber, auf welchen Topics er Daten senden und empfangen möchte. Der Master stellt daraufhin die entsprechenden Peer-to-Peer-Verbindungen her.

ROS Befehle:

- `rosnode` - Zeigt Informationen über laufende Nodes an
- `rostopic` - Zeigt Informationen über registrierte Topics an
- `rosmsg` - Zeigt Informationen über verfügbare Message-Typen an
- `rosrun` - Startet einen Node
- Die korrekte Verwendung der Befehle finden Sie im ROS Cheat Sheet

Zunächst benötigen wir einen Node, der als Schnittstelle zum Gamepad fungiert und die Eingaben weiterleiten kann. Dazu nutzen wir das bereits vorhandene Package joy[2] mit dem zugehörigen Node `joy_node`. Die Eingaben vom Gamepad müssen daraufhin von einem weiteren Node in Bewegungsbefehle umgewandelt werden. Ein Standard in ROS für solche Befehle ist der Message-Typ geometry_msgs/Twist[3]. Für die Umwandlung ist bereits das Package teleop_twist_joy[4] installiert.

Aufgaben:

- Starten Sie den Node `joy_node` aus dem Package `joy` in einem neuen Terminal!
- Untersuchen Sie den laufenden Node auf die dazugehörigen Topics. über welches Topic werden die Gamepad-Eingaben transportiert? überprüfen Sie die Funktion des Gamepads[a] indem Sie sich das den Eingaben entsprechende Topic am Terminal ausgeben lassen!
- Starten Sie in einem neuen Terminal den Node `teleop_node` zum Umwandeln der Gamepad-Eingaben in Twist-Messages.
- Aus welchen Komponenten besteht eine Twist-Message? Welche Einträge sind dabei für den Pioneer relevant?
- Beenden Sie anschließend beide Nodes wieder durch Drücken von `Ctrl + C` im jeweiligen Terminal!

[a] Sollte sich das Gamepad im Standby-Modus befinden, lässt es sich durch Drücken der Home-Taste reaktivieren.

4.4 Launch-Files und Catkin

ROS-Systeme können aus sehr vielen Nodes bestehen, so dass es schnell sehr aufwendig würde, alle Nodes einzeln zu starten. Launch-Files erleichtern den Start ganzer Systeme. Innerhalb einer XML-Struktur werden alle einzelnen Nodes und ihre Eigenschaften aufgelistet und können so mit einem einzigen Befehl gestartet werden. Außerdem kann man noch weitere Konfigurationen in Launch-Files festlegen, mit denen wir uns später noch beschäftigen.

[2] http://wiki.ros.org/joy
[3] http://docs.ros.org/api/geometry_msgs/html/msg/Twist.html
[4] http://wiki.ros.org/teleop_twist_joy

ROS Befehle:

- `roslaunch` - Startet ein Launch-File
- `catkin_create_pkg` - Legt ein neues Package im Catkin-Workspace an
- `catkin_make` - Kompiliert und erstellt den Catkin-Workspace
- Die korrekte Verwendung der Befehle finden Sie im ROS Cheat Sheet

Launch-File Elemente:

- `<launch>` - XML-Wurzelelement eines Launch-Files
- `<node>` - XML-Element für ROS Nodes

Da Launch-Files Teile von ROS Packages sind, benötigen wir nun einen eigenen Workspace, der es uns erlaubt, neue Packages anzulegen. Zum Erstellen der Packages nutzen wir das in der ROS-Installation enthaltene Tool *Catkin*.

Aufgaben:

- Wechseln Sie in den Ordner `catkin_ws/` in Ihrem Home-Verzeichnis und legen Sie Ihren Catkin-Workspace mit Hilfe des Kommandozeilen-Befehls an!
- Legen Sie das Package `robprak` mit Hilfe der Kommandozeilen-Befehle in Ihrem Catkin-Workspace an! Erstellen Sie darin den Ordner `launch` und fügen Sie ein .launch-File hinzu, mit welchem Sie die beiden für das Gamepad genutzen Nodes auf einmal starten können!
- Ergänzen Sie die Datei `.bashrc` in Ihrem Home-Verzeichnis um die Zeile: `source ~/catkin_ws/devel/setup.bash`
- Erstellen Sie Ihren Workspace und testen Sie Ihr Launch-File!

4.5 ROS im Netzwerk

Ein großer Vorteil von ROS ist die Netzwerkfähigkeit. Wenn ein Node sich an einem Master im Netzwerk anmelden möchte, findet er diesen über die Umgebungsvariable `ROS_MASTER_URI` und teilt dem Master die eigene Adresse mit, welche als Umgebungsvariable `ROS_IP` gespeichert sein muss. Mit diesen Informationen kann der Master die Kommunikations-Verbindungen auch über das Netzwerk herstellen. Mit Hilfe von Launch-Files können Nodes auch auf anderen Systemen gestartet werden. Dazu muss das System mit einem Machine-Tag beschrieben werden. Innerhalb des Node-Tags kann dann das System, auf dem der Node laufen soll, als Attribut mitgegeben werden.

Launch-File Elemente:

- `<machine>` - XML-Element für Systeme im Netzwerk

Für die Ansteuerung des Pioneers und viele weitere mobile Roboter existiert die `OpenSource-Bibliothek ARIA`, welche auch bereits als ROS-Package RosAria[5] implementiert wurde. Da diese direkt mit der RS232-Schnittstelle des Pioneers kommuniziert, muss der Node auf dem Raspberry Pi gestartet werden. Folgende Informationen benötigen wir für die Kommunikation des ROS-Masters mit dem Raspbery Pi:

- **IP-Adresse:** 192.168.0.77
- **User:** pi
- **Passwort:** pioneerpi
- **Pfad zum Env-Loader (Pi):** /opt/ros/indigo/env2.sh

Aufgaben:

- Setzen Sie dauerhaft die Umgebungsvariablen, indem Sie folgende Zeilen der Datei `.bashrc` in Ihrem Home-Verzeichnis hinzufügen:
  ```
  export ROS_IP=192.168.0.68
  export ROS_MASTER_URI=http://192.168.0.68:11311
  export ROSLAUNCH_SSH_UNKNOWN=1
  ```
- Fügen Sie Ihrem Launch-File ein Machine-Tag für den Raspberry Pi hinzu! Anschließend erweitern Sie das File um den Node RosAria, so dass dieser auf dem Pi ausgeführt wird! Ergänzen Sie den Node-Tag um ein weiteres XML-Element: `<param name="port" value="/dev/pioneer"/>`. Parameter werden im folgenden Kapitel näher erläutert.
- Schalten Sie den Pioneer ein und testen Sie Ihr Launch-File!

4.6 ROS Graph, Remapping und Parameter

Wenn Sie das bis hierhin erstellte Launch-File starten, werden Sie feststellen, dass zwar eine Verbindung zum Pioneer hergestellt wird, die Steuerung mit dem Gamepad allerdings noch nicht möglich ist. Um den Grund dafür zu finden, wollen wir einen Blick auf den Graphen unseres ROS Systems werfen. Dafür benutzen wir den Node `rqt_graph` aus dem gleichnamigen Package. Starten Sie diesen in einem neuen Terminal um eine Visualisierung Ihres ROS Graphen zu erhalten.

Nodes werden durch Ellipsen dargestellt, Topics durch Rechtecke. Zunächst sind unsere drei gestarteten Nodes erkennbar, sowie das Topic `joy` worüber bereits kommuniziert wird. Der Node `RosAria` ist nicht mit dem Rest des Graphen verbunden. Um nun sämtliche Topics im System anzuzeigen, wechseln Sie im DropDown-Menü links

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

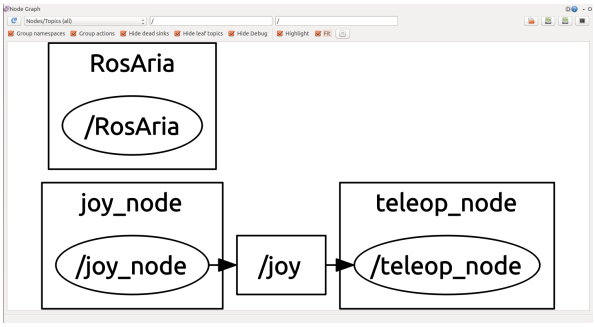


Abbildung 4.1: Der ROS Graph mit den aktiven Nodes und Topics

oben auf den Eintrag *Nodes/Topics (all)* und entfernen die Häkchen bei *Hide dead sinks* sowie *Hide leaf topics*.

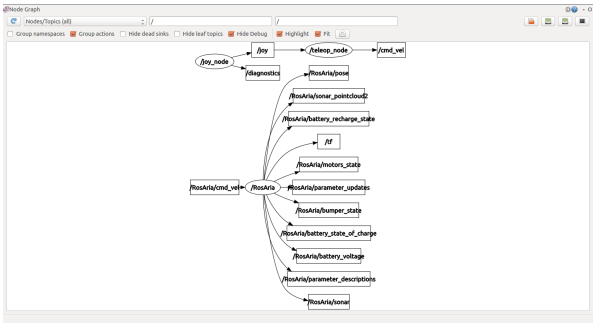


Abbildung 4.2: Der ROS Graph mit allen Nodes und Topics

Nun wird auch schnell klar, warum keine Kommunikation zwischen den zu untersuchenden Nodes besteht: Der Node `teleop_node` ist Publisher für das Topic `cmd_vel`. Der Node RosAria ist Subscriber für das Topic `RosAria/cmd_vel` (der Namespace kann hier variieren, entsprechend dem Namen, den Sie im Launch-File für den Node festgelegt haben).

Um das System so zu konfigurieren, dass statt zwei Topics, das selbe verwendet wird nutzen wir Remapping. Damit wird einem Topic statt des im Quellcode festgelegten Namens ein vom Benutzer gewählter Name zugeordnet. Remapping kann sowohl in der Kommandozeile als auch in Launch-Files beschrieben werden.

Viele ROS Nodes verfügen über verschiedene Parameter, mit deren Hilfe sie flexibel zu konfigurieren sind. Der Befehl `roscore` startet neben dem Master auch einen Parameter Server, der dafür zuständig ist alle vom Benutzer gesetzten Parameter an die betroffenen Nodes weiterzugeben. Wenn keine gesetzt werden, werden die Nodes mit Standard-Werten gestartet. Parameter können sowohl in der Kommandozeile als auch in Launch-Files gesetzt werden. Für die meisten Nodes sind die verfügbaren Parameter in der Dokumentation im ROS-Wiki ausführlich beschrieben.

ROS Befehle:

- `rosparam` - Setzt und liest Parameter

Launch-File Elemente:

- `<remap>` - XML-Element zum Remappen von Topics
- `<param>` - XML-Element zum Setzen von Parametern

Aufgaben:

- Modifizieren Sie ihr Launch-File, so dass die Twist-Messages korrekt übergeben werden!
- Aktivieren Sie den Turbomodus, indem Sie in Ihrem Launch-File den Parameter `enable_turbo_button` innerhalb des Teleop-Nodes setzen. Der Parameter erwartet einen Integer-Wert, der dem entsprechenden Knopf entspricht. Nutzen Sie `rostopic`) um die Zahl für eine von Ihnen gewählte Taste des Gamepads herauszufinden.
- Fahren Sie den Pioneer mit dem Gamepad kollisionsfrei durch das Labor!

4.7 Visualisierung

RViz ist ein mächtiges Visualisierungs-Tool für ROS zum Anzeigen von Modellen, Koordinatensystemen, Sensordaten und vielem mehr. Das Tool lässt sich als ROS Node `rviz` aus dem gleichnamigen Package starten.

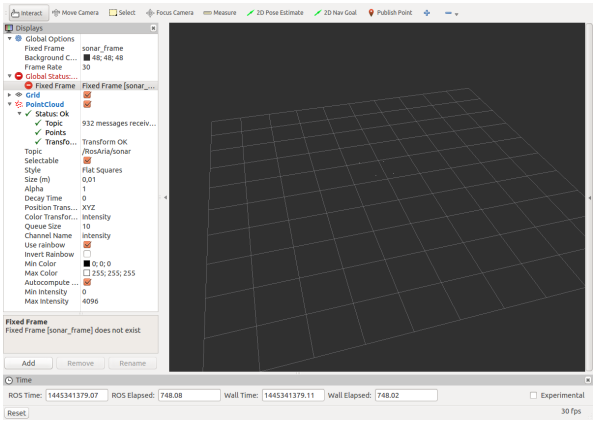


Abbildung 4.3: Der ROS Graph mit allen Nodes und Topics

Links im Display-Bereich kann man die zu visualisierenden Daten hinzufügen und Einstellungen vornehmen. Über den Button *Add* kann man zwischen einer Vielzahl von Message-Typen auswählen. Anschließend muss unter dem Punkt Topic die Quelle

für die Daten angegeben werden. Meist erkennt die Software bereits das passende Topic selbst. In den *Global Options* muss unter *Fixed Frame* noch der Name des zugehörigen Frames eingetragen werden. Dieser lässt sich zum Beispiel herausfinden, indem man sich im Terminal ein paar Messages des gewünschten Topics ausgeben lässt.

Aufgabe:

- Visualisieren Sie die Sensordaten der im Pioneer verbauten Sonar-Sensoren und machen Sie sich mit dem Message-Typ `sensor_msgs/PointCloud` vertraut!

4.8 Schreiben eines ROS Nodes

Im folgenden Abschnitt beschäftigen wir uns mit der Implementierung eines eigenen Nodes, dessen Aufgabe es ist, Hindernisse vor dem Pioneer mit Hilfe des Sonars zu erkennen, die aktuelle Bewegung daraufhin zu stoppen und den Roboter zu drehen, bis der Weg wieder frei ist.

Um weiterhin ROS in den Vordergrund zu stellen, wird Ihnen ein Template für diesen Node zur Verfügung gestellt, worin der algorithmische Teil der Aufgabe bereits enthalten ist und die ROS-spezifischen Strukturen noch zu ergänzen sind. In diesem Praktikum wird ausschließlich C++ zur Implementierung von Nodes verwendet. Sie können die Aufgabe auch mit Python lösen, dafür liegen aktuell aber weder Template noch Musterlösung vor.

ROS C++ Klassen:

- `ros::NodeHandle` - Schnittstelle zum Erstellen von Subscribern und Publishern
- `ros::Publisher` - Managt das Senden von Messages an ein Topic
- `ros::Subscriber` - Managt das Empfangen von Messages über ein Topic
- `ros::Rate` - Stellt Schleifen in einer gewünschten Frequenz bereit

Aufgaben:

- Erstellen Sie den Ordner `src` in Ihrem Package und kopieren Sie die Datei `sonar_motion_stop.cpp` aus dem Ordner Templates auf Ihrem Desktop dorthin!
- Vervollständigen Sie den Node `sonar_motion_stop` indem Sie alle mit TODO markierten Stellen im C++ - Template abarbeiten!
- Ersetzen Sie die Datei `CMakeLists.txt` in Ihrem Package durch die gleichnamige Datei aus dem Template-Ordner!
- Erstellen Sie Ihren Catkin-Workspace, um Ihren Node zu kompilieren. Nutzen Sie dazu den Befehl `catkin_make` (falls der Node nicht erstellt wird, ergänzen Sie die Option `--force-cmake`).

- Erweitern Sie Ihr Launch-File um den erstellten Node! Entfernen Sie das Remapping des Topics `cmd_vel` aus der vorherigen Aufgabe! Testen Sie Ihren Node, indem Sie den Pioneer mit dem Gamepad auf einen Karton zusteuern!

4.9 Tranformationen und Frames

Transformationen und Frames kennen Sie bereits aus der Robotik Vorlesung. In ROS werden Frames über ihren Namen, die `frame_id` eindeutig identifiziert. Die Beziehung zwischen zwei Frames wird durch durch eine relative Pose mit 6 Freiheitsgraden repräsentiert, die aus einer Translation gefolgt von einer Rotation besteht. Seien W und A zwei Frames, dann ist die Pose von A in W gegeben durch die Translation vom W-Ursprung zum A-Ursprung und der Rotation der A-Koordinatenachsen in W. Die Translation ist dabei ein Vektor mit drei Elementen in W-Koordinaten. Die Rotation von A is gegeben durch eine Rotationsmatrix.

Das Package tf[6] bietet umfangreiche Möglichkeiten mit Frames und Transformationen sowie ihren Änderungen über die Zeit zu arbeiten. Dafür beinhaltet es verschiedene Datentypen und Nodes: `view_frames` ist sehr praktisch zur Visualisierung der aktuell vorhandenen Transformationen und Frames und erzeugt ein PDF-File mit dem entsprechenden Graphen. `tf_echo <source_frame> <target_frame>` gibt die Transformation zwischen zwei gewünschten Frames auf der Konsole aus. Der Node `static_transform_publisher <x> <y> <z> <yaw> <pitch> <roll> <frame_id> <child_frame_id> <period_in_ms>`[7] kann dazu genutzt werden, eine statische Transformation (z.B. vom Roboter-Ursprung zu einem verbauten Sensor) dauerhaft über `tf` zu publishen. Dabei kann ein translatorischer und rotatorischer Offset angegeben werden, sowie die Frequenz in der gesendet werden soll.

Transformationen werden in ROS üblicherweise über das Topic `tf` bereitgestellt. Anhand der Bezeichnungen von `frame_id` und `child_frame_id` lässt sich feststellen, um welche Transformation es sich handelt. Mit Hilfe der Timestamps lässt sich der zeitliche Zusammenhang zwischen Transformationen herstellen.

Die Sonar-Sensoren sind bestenfalls dafür geeignet, Hindernisse in der unmittelbaren Umgebung des Roboters zu erkennen. Wesentlich präziser arbeitet ein Laserscanner, wie der auf dem Pioneer montierte. Dieser tastet einen Halbkreis ab und liefert 180 Abstandswerte bis zu 8m pro Zeitintervall. Auf dem Raspberry Pi ist bereits das ROS-Package sicktoolbox_wrapper[8] installiert. Der passende Node für das Laserscanner-Modell heißt `sicklms`.

Mit Hilfe des Scanners ist es möglich den Arbeitsraum des Pioneers zu kartografieren. Dazu wird das SLAM-Verfahren (Simultaneous Localization and Mapping) verwendet, eine Methode mit der ein mobiler Roboter gleichzeitig eine Karte seiner Umgebung erstellen uns seine Pose innerhalb dieser Karte schützen kann. Letzteres geschieht dabei durch Auswertung der Odometrie-Daten des Roboters. Für dieses Verfahren nutzen wir das bereits installierte ROS-Package gmapping[9] mit dem Node `slam_gmapping`.

Um die Transformation zwischen Karte und Pose des Roboters zu berechnen, werden die Transformationen vom Startpunkt des Roboters (`odom`) zu dessen aktuellerPose (`base_link`), sowie vom Roboter zum Laserscanner (`laser`) benötigt. Die erste wird vom `RosAria` Node bereitgestellt, für die zweite wird ein `static_transform_publisher`

[6] http://wiki.ros.org/tf
[7] Das Attribute args des Node-Tags kann zum Übergeben von Parametern genutzt werden.
[8] http://wiki.ros.org/sicktoolbox_wrapper
[9] http://wiki.ros.org/gmapping

Node benötigt. Damit alle Transformationen zeitlich synchronisiert sind, muss dieser Node auf dem Pi gestartet werden.

Aufgaben:

- Erweitern Sie Ihr Launch-File mit allen Nodes, die Sie zur Kartografierung mittels SLAM-Verfahren benötigen! Übergeben Sie dem Laserscanner-Node die Paramter `port=/dev/sicklms` und `baud=38400`! Verwenden Sie eine Transformation vom Roboter zum Laser, bei der alle Werte 0 sind. Übergeben Sie diese Werte mit dem Attribut `args`!
- Kartografieren Sie das Labor und lassen Sie sich das Ergebnis in RViz anzeigen!

4.10 ROS Services

Die Kommunikation über das Modell mit Publisher und Subscriber ist zwar flexibel, jedoch nicht sehr geeignet für Prozeduraufrufe in verteilten System. ROS Services ermöglichen eine solche Request-Reply-Kommunikation. Ein Service ist definiert durch zwei Messages, eine für die Anfrage und eine für die Antwort. Dabei können die Messages aus einem oder mehreren Message-Typen bestehen, oder auch aus keinem. Die wird in entsprechenden `.srv`-Dateien innerhalb eines Packages festgelegt. Die Implementierung des Services erfolgt innerhalb der Implementierung eines Nodes. Dieser kann den Service dann über einen in der Implementierung festgelegten Namen anbieten. Ein Service kann von einem weiteren Node aufgerufen werden (Client-Node), oder mit Hilfe des Konsolenbefehls `rosservice call`.

Ihre Aufgabe ist es, den Node `sonar_motion_stop` um einen Service zu erweitern. Ergänzen Sie dazu den Node um eine Instanz der Klasse `ros::ServiceServer` sowie deren Initialisierung durch die NodeHandle-Methode
`advertiseService(string serviceName, callback function)`. Wird der Service aufgerufen, sollen Twist-Messages für den Publisher generiert werden, so dass der Roboter im hindernisfreien Raum wieder in seine Ursprungspose zurückführt. Dies könnte in drei Phasen geschehen: 1) Den Roboter rotieren, so dass er sich in Richtung seines Startpunktes ausrichtet. 2) Den Roboter vorwärts fahren lassen, bis er seinen Startpunkt erreicht. 3) Den Roboter rotieren, bis er seine urpsrüngliche Null-Pose wieder eingenommen hat.

ROS Befehle:

- `rosservice` - Zeigt Informationen über verfügbare Services an und ruft diese auf

Aufgaben:

- Erstellen Sie in Ihrem Package `robprak` den Ordner `srv` und kopieren Sie die Datei `HeadingHome.srv` aus dem Ordner Templates auf Ihrem Desktop dort hin!

- Entfernen Sie die Kommentarzeichen in der Datei `CMakeLists.txt` in Ihrem Package für die Zeilen `add_service_files(FILES HeadingHome.srv)`!
- Entfernen Sie die Kommentarzeichen für die mit TODO Aufgabe 4.10 gekennzeichnete Funktion in der Datei `sonar_motion_stop.cpp`!
- Implementieren Sie einen ROS Service für Ihren Node `sonar_motion_stop` mit dem oben beschriebenen Ergebnis und testen Sie den Service!
- Hinweise: Nutzen Sie die Behandlungsmethode des Services nur, um eine Antwort zu generieren und setzen Sie Membervariablen so, dass der eigentliche Algorithmus in der Methode zum Berechnen der Twist-Messages erfolgt. Verwenden Sie einen zusätzlichen Subscriber für das Topic `tf`. Die hierüber gesendeten Messages vom Typ `tf2_msgs/TFMessage` verwenden Quaternionen für die Darstellung der Rotation. In der Ebene können Sie die aktuelle Rotation des Roboters mittels `2 * asin(z)` ermitteln, wobei `z` den entsprechende Wert der Quaternion darstellt. Sie müssen alle verwendeten Message-Typen als `include` hinzufügen: `robprak/HeadingHome.h` sowie `tf2_msgs/TFMessage.h`.

ROS Noetic Cheatsheet

Filesystem Management Tools

rospack	A tool for inspecting packages.
rospack profile	Fixes path and pluginlib problems.
roscd	Change directory to a package.
rospd/rosd	Pushd equivalent for ROS.
rosls	Lists package or stack information.
rosed	Open requested ROS file in a text editor.
roscp	Copy a file from one place to another.
rosdep	Installs package system dependencies.
roswtf	Displays a errors and warnings about a running ROS system or launch file.
catkin_create_pkg	Creates a new ROS stack.
wstool	Manage many repos in workspace.
catkin_make	Builds a ROS catkin workspace.
rqt_dep	Displays package structure and dependencies.

Usage:

```
$ rospack find [package]
$ roscd [package[/subdir]]
$ rospd [package[/subdir] | +N | -N]
$ rosd
$ rosls [package[/subdir]]
$ rosed [package] [file]
$ roscp [package] [file] [destination]
$ rosdep install [package]
$ roswtf or roswtf [file]
$ catkin_create_pkg [package_name] [depend1]..[dependN]
$ wstool [init | set | update]
$ catkin_make
$ rqt_dep [options]
```

Start-up and Process Launch Tools

roscore

The basis nodes and programs for ROS-based systems. A roscore must be running for ROS nodes to communicate.

Usage:

```
$ roscore
```

rosrun

Runs a ROS package's executable with minimal typing.

Usage:

```
$ rosrun package_name executable_name
```

Example (runs turtlesim):

```
$ rosrun turtlesim turtlesim_node
```

roslaunch

Starts a roscore (if needed), local nodes, remote nodes via SSH, and sets parameter server parameters.

Examples:

Launch a file in a package:

```
$ roslaunch package_name file_name.launch
```

Launch on a different port:

```
$ roslaunch -p 1234 package_name file_name.launch
```

Launch on the local nodes:

```
$ roslaunch --local package_name file_name.launch
```

Introspection and Command Tools

rosnode

Displays debugging information about ROS nodes, including publications, subscriptions and connections.

Commands:

rosnode ping	Test connectivity to node.
rosnode list	List active nodes.
rosnode info	Print information about a node.
rosnode machine	List nodes running on a machine.
rosnode kill	Kill a running node.

Examples:

Kill all nodes:

```
$ rosnode kill -a
```

List nodes on a machine:

```
$ rosnode machine aqy.local
```

Ping all nodes:

```
$ rosnode ping --all
```

rostopic

A tool for displaying information about ROS topics, including publishers, subscribers, publishing rate, and messages.

Commands:

rostopic bw	Display bandwidth used by topic.
rostopic echo	Print messages to screen.
rostopic find	Find topics by type.
rostopic hz	Display publishing rate of topic.
rostopic info	Print information about an active topic.
rostopic list	List all published topics.
rostopic pub	Publish data to topic.
rostopic type	Print topic type.

Examples:

Publish hello at 10 Hz:

```
$ rostopic pub -r 10 /topic_name std_msgs/String hello
```

Clear the screen after each message is published:

```
$ rostopic echo -c /topic_name
```

Display messages that match a given Python expression:

```
$ rostopic echo --filter "m.data=='foo'/topic_name
```

Pipe the output of rostopic to rosmsg to view the msg type:

```
$ rostopic type /topic_name | rosmsg show
```

rosservice

A tool for listing and querying ROS services.

Commands:

rosservice list	Print information about active services.
rosservice node	Print name of node providing a service.
rosservice call	Call the service with the given args.
rosservice args	List the arguments of a service.
rosservice type	Print the service type.
rosservice uri	Print the service ROSRPC uri.
rosservice find	Find services by service type.

Examples:

Call a service from the command-line:

```
$ rosservice call /add_two_ints 1 2
```

Pipe the output of rosservice to rossrv to view the srv type:

```
$ rosservice type add_two_ints | rossrv show
```

Display all services of a particular type:

```
$ rosservice find rospy_tutorials/AddTwoInts
```

rosparam

A tool for getting and setting ROS parameters on the parameter server using YAML-encoded files.

Commands:

rosparam set	Set a parameter.
rosparam get	Get a parameter.
rosparam load	Load parameters from a file.
rosparam dump	Dump parameters to a file.
rosparam delete	Delete a parameter.
rosparam list	List parameter names.

Examples:

List all the parameters in a namespace:

```
$ rosparam list /namespace
```

Setting a list with one as a string, integer, and float:

```
$ rosparam set /foo "['1', 1, 1.0]"
```

Dump only the parameters in a specific namespace to file:

```
$ rosparam dump dump.yaml /namespace
```

rosmsg

Displays Message/Service (msg/srv) data structure definitions.

Commands:

rosmsg show	Display the fields in the msg/srv.
rosmsg list	Display names of all msg/srv.
rosmsg md5	Display the msg/srv md5 sum.
rosmsg package	List all the msg/srv in a package.
rosmsg packages	List all packages containing the msg/srv.

Examples:

Display the Pose msg:

```
$ rosmsg show Pose
```

List the messages in the nav_msgs package:

```
$ rosmsg package nav_msgs
```

List the packages using sensor_msgs/CameraInfo:

```
$ rosmsg packages sensor_msgs/CameraInfo
```

tf_echo

A tool that prints the information about a particular transformation between a source_frame and a target_frame.

Usage:

```
$ rosrun tf tf_echo <source_frame> <target_frame>
```

Examples:

To echo the transform between /map and /odom:

```
$ rosrun tf tf_echo /map /odom
```

rqt_topic

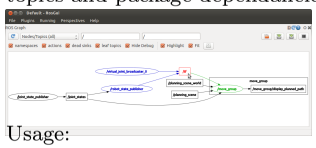
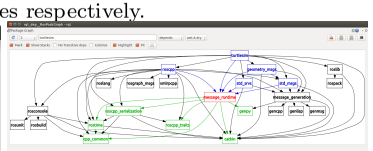
A tool for viewing published topics in real time.

Usage:

```
$ rqt
Plugin Menu->Topic->Topic Monitor
```

rqt_graph, and rqt_dep

Tools for displaying graphs of running ROS nodes with connecting topics and package dependancies respectively.

Usage:

```
$ rqt_graph
$ rqt_dep
```

Data Visualization Tools

view_frames

A tool for visualizing the full tree of coordinate transforms.

Usage:

```
$ rosrun tf2_tools view_frames.py
$ evince frames.pdf
```

ROS Noetic Catkin Workspaces

Create a catkin workspace

Setup and use a new catkin workspace from scratch.

Example:

```
$ source /opt/ros/noetic/setup.bash
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

Checkout an existing ROS package

Get a local copy of the code for an existing package and keep it up to date using wstool.

Examples:

```
$ cd ~/catkin_ws/src
$ wstool init
$ wstool set tut --git git://github.com/ros/ros_tutorials.git
$ wstool update
```

Create a new catkin ROS package

Create a new ROS catkin package in an existing workspace with catkin create package.

Usage:

```
$ catkin_create_pkg <package_name> [depend1] [depend2]
```

Example:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg tutorials std_msgs rospy roscpp
```

Build all packages in a workspace

Use catkin_make to build all the packages in the workspace and then source the setup.bash to add the workspace to the ROS_PACKAGE_PATH.

Examples:

```
$ cd ~/catkin_ws
$ ~/catkin_make
$ source devel/setup.bash
```

CMakeLists.txt

Your CMakeLists.txt file MUST follow this format otherwise your packages will not build correctly.

`cmake_minimum_required()` Specify the name of the package
`project()` Project name which can refer as ${PROJECT_NAME}
`find_package()` Find other packages needed for build
`catkin_package()` Specify package build info export

Build Executables and Libraries:

Use CMake function to build executable and library targets. These macro should call after `catkin_package()` to use `catkin_*` variables.

```
include_directories(include ${catkin_INCLUDE_DIRS})
add_executable(hoge src/hoge.cpp)
add_library(fuga src/fuga.cpp)
target_link_libraries(hoge fuga ${catkin_LIBRARIES})
```

Message generation:

There are `add_{message,service,action}_files()` macros to handle messages,services and actions respectively. They must call before `catkin_package()`.

```
find_package(catkin COMPONENTS message_generation std_msgs)
add_message_files(FILES Message1.msg)
generate_messages(DEPENDENCIES std_msgs)
catkin_package(CATKIN_DEPENDS message_runtime)
```

If your package builds messages as well as executables that use them, you need to create an explicit dependency.

```
add_dependencies(hoge ${PROJECT_NAME}_generate_messages_cpp)
```


Launch-File XML Format

This section describes the XML Format used for roslaunch .launch files. These are parts of ROS packages and usually located within the folder `packagename/launch/`.

<launch> tag

The root element of any roslaunch file acts as container for the other elements.

Elements:
<node>
<param>
<remap>
<machine>
<include>

<node> tag

Specifies a ROS node that you wish to have launched.

Attributes:
pkg="mypackage" - Package of node
type="nodetype" - Node Type
name="nodename" - Node name
args="ärg1 arg2 arg3" - Pass arguments to node (optional)
machine="machine-name" - Launch node on designated machine (optional)

Elements:
<param>
<remap>
<env>

<param> tag

Defines a parameter to be set on the Parameter Server.

Attributes:
name="namespace/name" - Parameter name
value="value" - Defines the parameter value (optional)
type=ßtr|int|double|bool" - Type of the parameter (optional)

<remap> tag

Allows you to pass in name remapping arguments to the ROS node that you are launching.

Attributes:
from="„original-name" - Name that your are remapping
to="new-name" - Target name

<machine> tag

Declares a remote machine that you can run ROS nodes on.

Attributes:
name="machine-name" - Name to assign to machine
address="127.0.0.1" - Network address or hostname of machine
env-loader=/path/to/env.sh" - Specify environment file on machine
default="true|false|never" - Sets machine as default (optional)
user="„username" - SSH user name for logging into machine (optional)
password="pw" - SSH password (optional)

Elements:
<env>

<env> tag

Allows you to set environment variables on nodes that are launched.

Attributes:
name=ënvironment-variable-name" - Environment variable you are setting
value=ënvironment-variable-value" - Environment variable value

<include> tag

Enables you to import another roslaunch XML file into the current file.

Attributes:
file="$(find pkg-name)/path/filename.xml" - Name of file to include

C++ Reference for ROS

ros::NodeHandle Class

roscpp's interface for creating subscribers, publishers, etc. You must call one of the `ros::init` functions prior to instantiating this class.

Methods:

```
Publisher advertise(string topicName, int queueSize)
```
Returns a ros::Publisher and notifies the master that the node will be publishing messages on the given topic

```
Subscriber subscribe(string topicName, int queueSize,
                     function callback)
```
Returns a ros::Subscriber and notifies the master to register interest in a given topic

```
ServiceServer advertiseService(string serviceName,
                               function callback)
```
Returns a ros::ServiceServer and notifies the master to register the provided service

ros::Publisher Class

Manages an advertisement on a specific topic.

Methods:
`void publish(const M &message)` - Publish a message on the topic associated with this Publisher

ros::Subscriber Class

Manages an subscription callback on a specific topic.

ros::ServiceServer Class

Manages an service advertisement.

ros::Rate Class

Class to help run loops at a desired frequency.

Methods:
`Rate(double frequency)` - Constructor, creates a Rate