Mobile Robotik mit ROS

Robot Operating System

11.10.2021

Einführung in ROS

In diesem Kapitel

1. Grundlagen Anwendung Linux
2. Was ist ROS
3. Publisher und Subscriber
4. Service Server und Service Client
5. Turtlesim als Einführungsprojekt
6. ROS-Launch
7. ROS2

Ubuntu kann genauso wie Windows oder auch iOS mittels der **grafischen Benutzeroberfläche**, Maus und Tastatur verwendet werden. Viele Anwendungen in Linux laufen jedoch direkt im **Terminal** – so auch die **Anwendungen in ROS**. Zusammen erarbeiten wir hier ein paar grundlegende Befehle im Terminal zum Umgang mit Ubuntu.

Öffnen Sie zunächst das Terminal mittels der Tastenkombination STRG + ALT + T

Alternativ können Sie mittels der Windowstaste in Ubuntu den so genannten Dash öffnen: Hier haben Sie die Möglichkeit Programme zu Suchen. Tippen Sie daher „Terminal“ danach ein, und bestätigen Sie das gefundene Programm mittels der Enter-Taste.

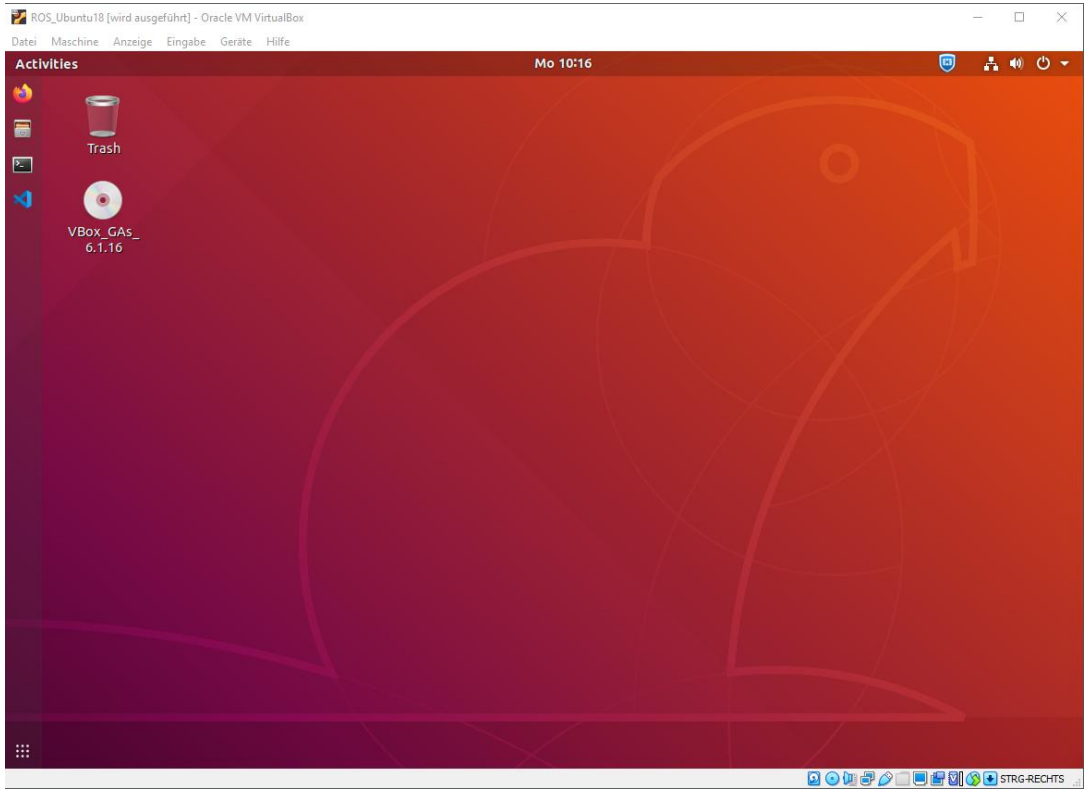
Beliebig viele **Tabs** können in einem Terminal mittels der Tastenkombination STRG + SHIFT + T

Die gewohnten Tastenkombinationen für Copy und Paste müssen im Bereich des Terminals etwas verändert werden:

Kopieren: STRG + ALT + C

Einfügen: STRG + ALT + V

Die Tastenkombination **STRG + C** führt dagegen zum Abbruch eines laufenden Programms im Terminal. Das kann beim Betrieb von ROS nützlich sein, sollten Sie ein Programm frühzeitig beenden wollen.



Virtuelle Maschine mit Ubuntu und ROS

Einführung in ROS

Zahlen und Fakten

ROS entstand **2008** als gefördertes Community Projekt. Die meisten Entwickler waren im Silicon Valley verortet. Die Firma **Willow Garage** übernahm den größten Teil der Koordinierung und Verwaltung für ROS. Ziel von ROS war es einen einheitlichen **Framework** für **mobile Robotik** zu finden, der unabhängig von verschiedenen Programmiersprachen schnell auf verteilten Systemen einsatzbereit ist.

Der bisherigen ROS-Version wird häufig vorgeworfen **nicht Echtzeitfähig** zu sein, bedingt durch die **Kommunikation über TCP/IP**. Dieses Problem wird in der Version **ROS 2** gelöst, da hier die Kommunikation über das Real-Time Transport Protocol (RTP). ROS 1 und ROS 2 können mittels einer Bridge beide gleichzeitig auf einem Robotersystem laufen. ROS 1 bietet aktuell mehr Funktionalität durch die Entwicklung in der Open Source-Community seit 2008 – aber ROS2 holt stark auf.

Weiteres: https://blog.generationrobots.com/de/ros-robot-operating-system/

ROS ist im Jahr 2021 sicherlich am weitverbreitetsten im Bereich der mobilen Robotik. Es existieren weitere Frameworks, die sich für den Betrieb auf einem Roboter eignen:

Microsoft Robotics Developer Studio:

Multiplattform-System von Microsoft. Kostenlos und besitzt ein Simulations-Tool. Nur mit Windows kompatibel. Programmierung ist unter einer .NET verhaltenen Sprache (am besten mit C#).

NAOQi:

Robotersystem für den NAO Roboter von Aldebaran Robotics. Programmierbar unter C++ oder Python. NAOQI ist wie ROS auch Open-Source.

URBI:

Von der französischen Firma Gostai, mit eigener Skriptsprache (URBIScript) und ebenso Open-Source. Der Framework kann in C++ programmiert werden URBI ist Multi-Plattformfähig.

11.10.2021

Peer-to-Peer: Ein Robotersystem kann aus mehreren Robotern bestehen, zum Beispiel verbunden über WLAN. Bei der Peer-to-Peer-Architektur kann jeder Computer innerhalb des Netzwerks mit jedem anderen Teilnehmer direkt Informationen austauschen.

Mikrokernel: ROS kommt mit einer eigenen Kommunikationsschicht und verwendet zahlreiche bestehende Werkzeuge um zum einen die Kommunikation im Robotersystem zu vereinheitlichen, aber auch um einen schnellen Einstieg zu gewähren.

Sprachvielfallt: ROS kann nicht nur in einer Sprache programmiert werden. Zahlreiche Programmiersprachen haben ich im Kontext der Robotik etabliert. Darunter am gängigsten C++, Java und Python. ROS ermöglicht so einen schnellen Einstieg unabhängig von einer einzelnen vorgegebenen Programmiersprache.

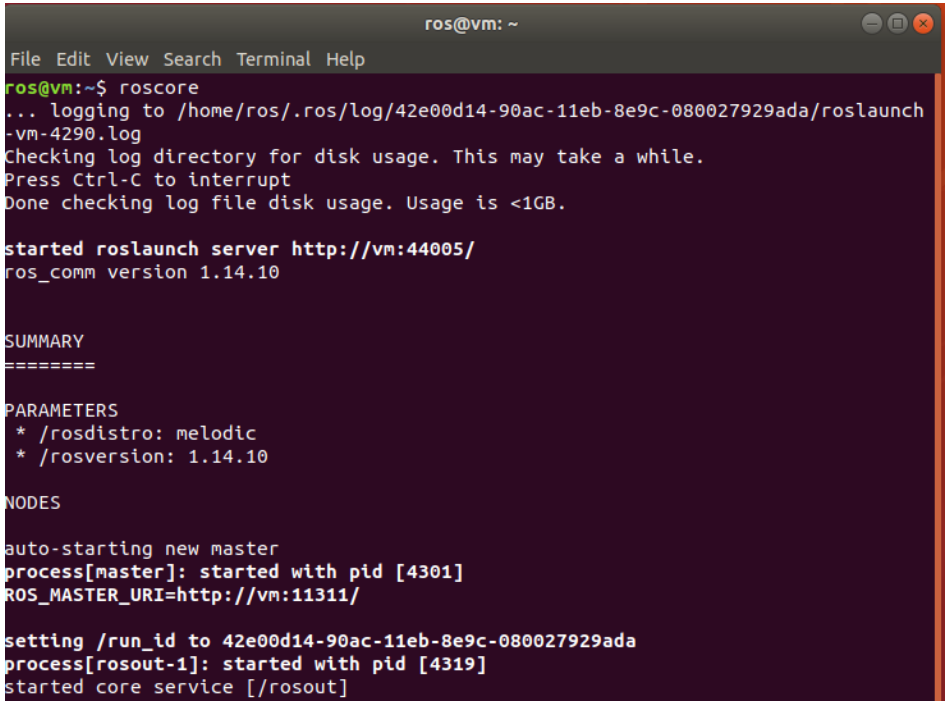
Leichtgewichtig: Die Grundinstallation von ROS ist wenige Megabyte groß und hat somit auch auf kleinen Einplatinencomputern oder gar Microcontrollern ausreichen Platz. Größere Bibliotheken können auf Wunsch selbst nachinstalliert werden.

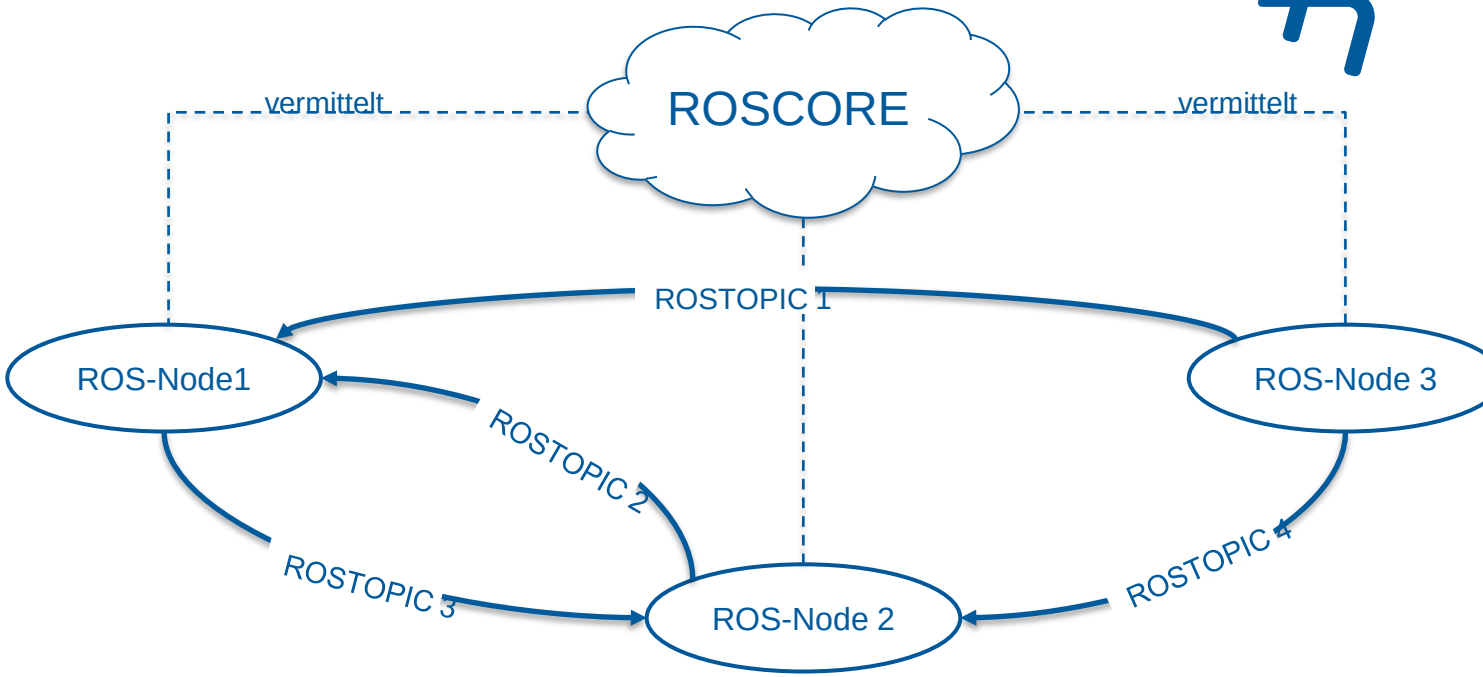
Kostenlos und Open-Source: Der Kern des Quellcodes von ROS ist frei verfügbar, einsehbar und auch änderbar. Die Open-Source-Community pflegt seit 2008 denn Quellcode von ROS und viele Pakete sind entstanden, welche frei verfügbar sind, und eine schnelle Entwicklung von eigenen Applikationen und Algorithmen erlaubt.

11.10.2021

Einführung in ROS

Aufbau von ROS





Der ROSCORE spielt beim Start eines jeden einzelnen Knoten die Rolle, dass er die Kommunikation zwischen den Knoten etabliert. Nach dieser Verknüpfung hat der ROSCORE keine Rolle mehr und könnte beendet werden.

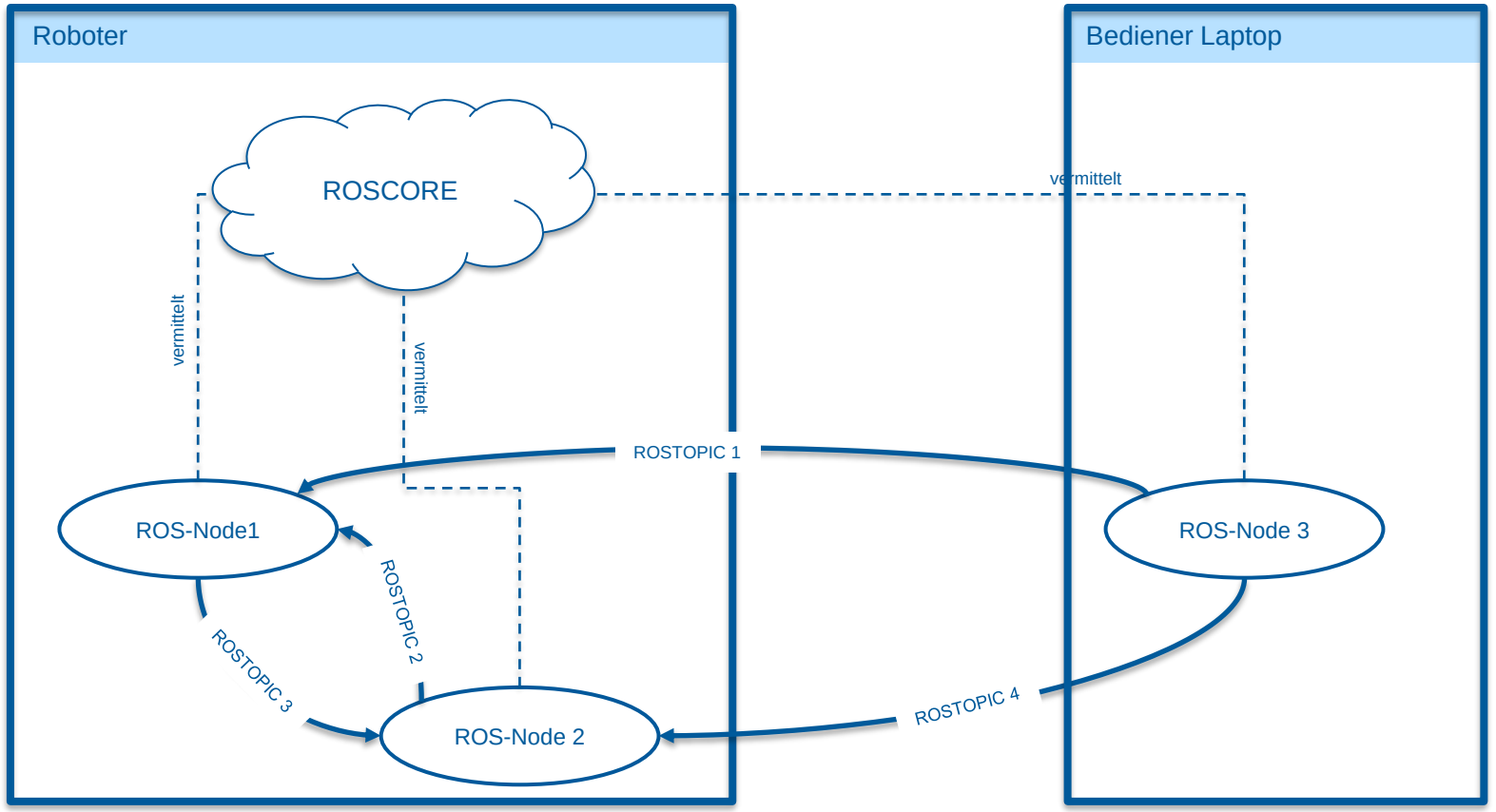
Aufgabe: Der ROS-Core wird mittels des Befehls

```
$ roscore
```

gestartet. Welche Informationen enthält das Fenster nach dem Start des ROSCOREs ?

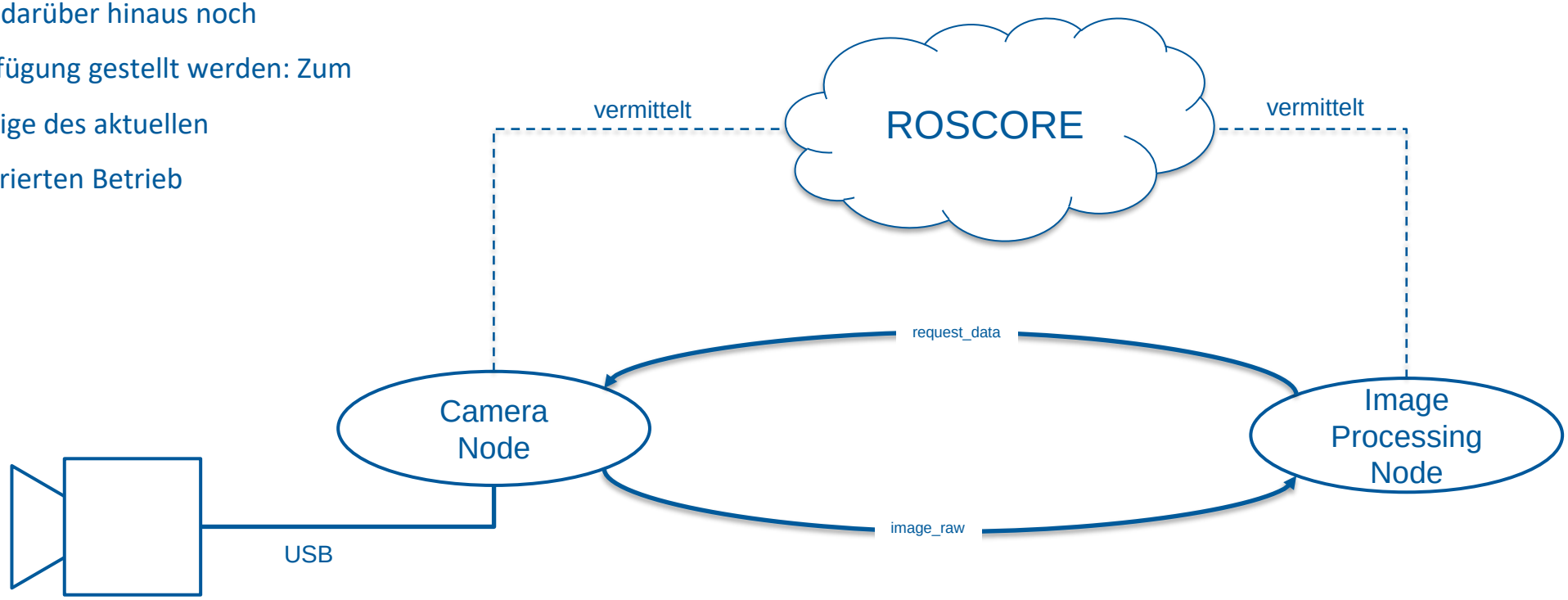
In einem ROS-System existiert immer nur ein **einzelner Master**. Die Knoten können allerdings auf beliebig vielen Teilnehmern im gleichen Netzwerk verteilt laufen.

Auch bei selbstständig agierenden Robotern ist häufig die Kommunikation zu einem externen Computer gewünscht. Entweder zum Programmieren oder Überwachen der Funktionalität oder zur Visualisierung.



Das folgende Beispiel zeigt das Abholen von Kameradaten – zum Beispiel über USB -- in einem Knoten, während die Bildverarbeitung in einem zweiten Knoten abläuft.

Die Daten der Kamera könnten darüber hinaus noch weiteren Interessenten zur Verfügung gestellt werden: Zum Beispiel einem Knoten zur Anzeige des aktuellen Kamerabildes für einen teleoperierten Betrieb



Start eines ROS-Knotens mittels ROSRUN

Der Start von ROS-Knoten erfolgt meist über das Terminal. ROS verwendet dazu eine eigene Syntax. Diese besteht immer aus dem Schlüsselwort **rosrun**, gefolgt vom Paketnamen, gefolgt vom Namen des auszuführenden Knotens. Auf diese Art können Knoten auch mehrfach in einem Robotersystem vorkommen; Voraussetzung ist, dass diese in Unterschiedlichen Paketen enthalten sind.

```
rosrun turtlesim turtlesim_node
```

Befehl zum Ausführen eines ROS-Programms. Der Befehl rosrun funktioniert von jedem Ort aus dem Terminal heraus.

Name des ROS-Pakets, in dem sich ein ausführbarer Knoten / Programm befindet.

Name des ROS-Knotens der gestartet wird. Ist kein Knoten mit dem Namen in ROS bekannt, so wird mit einer Fehlermeldung abgebrochen.

Nachrichten in ROS, also ROS-Topics, können mittels der Kommandozeile schnell untersucht werden. Ein zentrales Programm ist dabei rostopic. In der Vorlesung erproben wir den Umgang mit dem Befehl rostopic im Zusammenhang mit der Simulation turtlesim.

```
rostopic info /cmd_vel
```

Schlüsselwort zum Start des Terminal-Programms rostopic

Option zum Aufruf rostopic. Folgende Optionen sind Möglich: bw, delay, echo, find, hz, info, list, pub, type

Name des ROS-Topics welches untersucht oder bearbeitet werden soll

Übung

1. Starten Sie einen roscore in einem Terminal

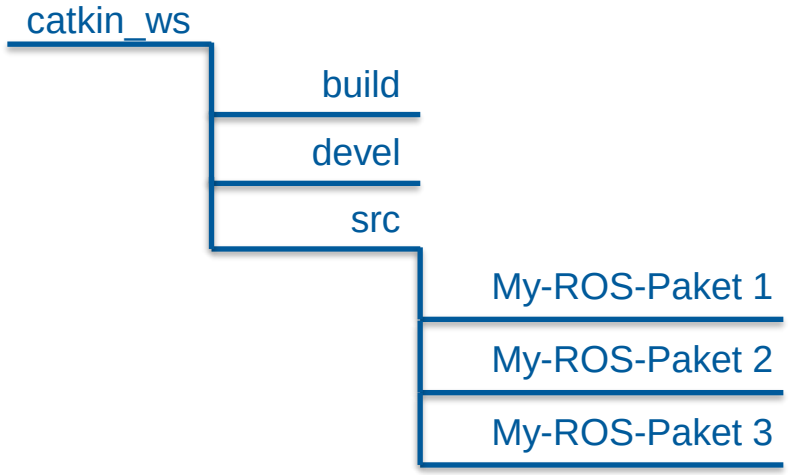
2. Starten Sie den Konten **turtle_sim** aus dem ROS-Paket **turtlesim** mittels **rosrun**. Was müssen Sie ins Terminal eingeben?

3. Starten Sie den Knoten **turtle_teleop_key** aus dem ROS-Paket **turtlesim** mittels **rosrun.**

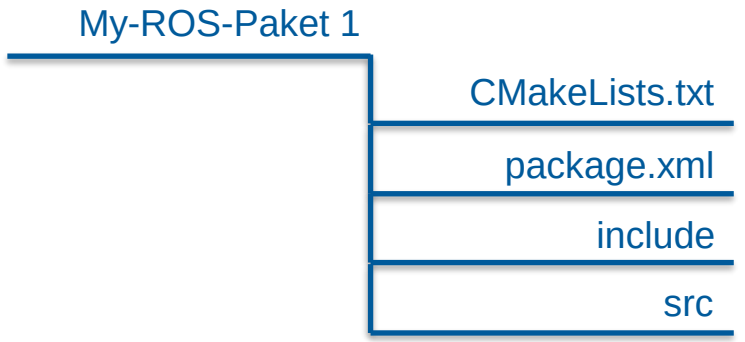
4. Finden Sie mittels des Befehls **rostopic** heraus, welche Nachrichten der Knoten zur Verfügung stellt.

5. Lassen Sie die Schildkröte mittels **rostopic** im Kreis fahren. Hierzu müssen Sie den Knoten **turtle_teleop_key** beenden.

Standardmäßig erfolgt die Installation von ROS im Ordner `/opt`, direkt unter dem Wurzelverzeichnis (/). Dort liegt der Source-Code und auch die ausführbaren Dateien, die zum Betrieb von ROS notwendig sind. Um das Verzeichnis opt zu bearbeiten sind root-Rechte notwendig.
Eigene Programme und entwickelte Anwendungen werden daher im catkin-Workspace (meist kurz `catkin_ws`) abgelegt.



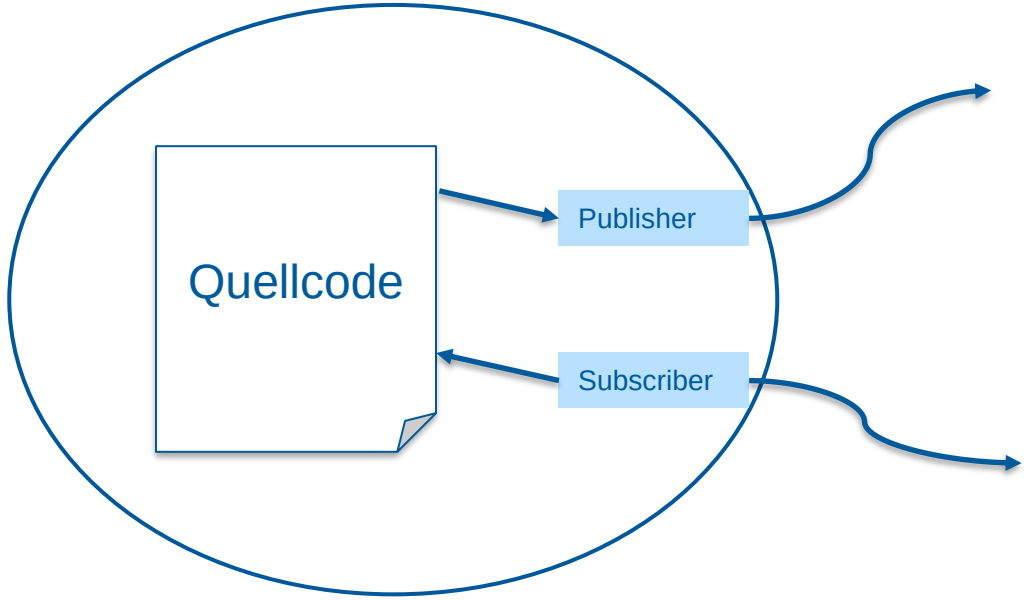
Jedes einzelne ROS-Paket beinhaltet zwei Dateien: CMakeLists.txt und package.xml sind für das Übersetzen der Knoten zuständig. Die Ordner `include` und `src` sind dafür gedacht dort Quellcode abzulegen.



Zur Kommunikation über ROS-Topics sind zwei Bausteine im Quellcode notwendig: **ROS-Publisher** haben die Aufgabe Nachrichten zu versenden. Im Gegenzug empfangen **ROS-Subscriber** ROS-Topics über das System.

Das Prinzip hinsichtlich Publisher und Subscriber ist über alle Programmiersprachen in ROS ähnlich – natürlich ist die Syntax verschieden. Jede Nachricht innerhalb eines ROS-Systems hat zwei Eigenschaften, wodurch diese sich unterscheidet und zugeordnet werden kann: Der **Typ** der Nachricht (zum Beispiel std_msgs/String) und der vergebene **Topic-Name** müssen übereinstimmen.

Haben zwei Nachrichten den Gleichen Namen, aber unterschiedliche Datentypen, so gibt ROS bei der Verwendung eine kontinuierliche Warnung über das Terminal aus.



Neben ROSTOPICS stellen ROSSERVICES eine weitere Kommunikationsmöglichkeit zwischen den Knoten her: Der Aufruf eines Services gibt meistens auch einen Wert zurück

```
rosservice info /spawn
```

Schlüsselwort zum Start des Terminal-Programms rosservice

Option zum Aufruf rosservice. Folgende Optionen sind Möglich:
call, find, **info**, list, type, uri

Name des ROS-SERVICES welcher untersucht oder bearbeitet werden soll

Übung

1. Starten Sie einen roscore in einem Terminal
2. Starten Sie den Konten **turtle_sim** aus dem ROS-Paket **turtlesim** mittels **rosrun**. Was müssen Sie ins Terminal eingeben?
3. Überprüfen Sie mittels **rosservice list** welche Dienste zur Verfügung stehen.
4. Erzeugen Sie eine zusätzliche Schildkröte. Überprüfen Sie im Anschluss die Topics, die dieser Schildkröte zugeordnet sind. Welchen Namen erhält die Schildkröte?
5. Entfernen Sie die Schildkröte wieder mittels eines Services.

Nun muss der bestehende Quellcode noch in ein ausführbares Programm übersetzt werden. Hierzu nutzt ROS für C++ Anwendungen standardmäßig den GCC-Compiler. Da eine Vielzahl von Optionen beim Übersetzen verwendet werden kann, ist der Compiler nochmal über Cmake gekapselt. Dieses wiederum ist letztlich über catkin, der-Dateiverwaltung von ROS gekapselt.

Für den folgenden Befehl muss in den Ordner **catkin_ws** gewechselt werden.

```
$ catkin_make
```

catkin_make übersetzt alle Programme im Workspace. Danach können die Programme wie gewohnt von jedem Ort im Terminal aus über rosrun gestartet werden.

Die ROS-Installation verwendet `catkin_make` zum übersetzen von C++-Code und zum Erzeugen von ROS-Paketen. Allerdings ist `catkin_make` häufig wenig praxistauglich: Um den Workspace zu übersetzen müssen Sie catkin_make immer im Workspace in das Terminal eingeben; das ist lästig und erfordert das Springen in der Ordnerstuktur.

Eine Alternative hierzu ist `catkin build`**:** Die Python-basierte Ergänzung zu catkin_make kann einen Workspace auch übersetzen, egal von welcher Position im Dateibaum aus. Die Installation ist mit wenigen Schritten erledigt.

Folgende Befehle geben Sie in das Terminal ein zur Installation :

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

```
$ sudo apt-get update
$ sudo apt-get install python-catkin-tools
```

Mittels dem folgenden Befehl können Sie nun den Workspace auch bauen. Achtung: Sie können nicht zwischen den beiden Build-Systemen wechseln. Wenn dann müssen Sie zuerst die Ordner **build** und **devel** im Workspace löschen.

```
$ catkin build
```

```
ros@vm:~/catkin_ws$ ls
build  devel  logs  src
```

Einführung in ROS

Hello Ros in C++

Der C++-Code auf der rechten Seite veröffentlicht eine Stringliteral über ROS. Nach dem Beginn der Main-Funktion wird zunächst der Knoten mittels der Funktion `ros::init( … )` in der ROS-Umgebung über den ROSCORE angemeldet.

Weitere Initialisierungen, wie das Anlegen eines Publishers zum Versenden von Nachrichten aus dem Programm sind notwendig.

Das Programm besitzt eine `while-Schleife`, welche kontinuierlich mit 10 Hz den vorgegebenen String versendet. Zusätzlich wird im Terminal noch der Text mittels `cout` ausgegeben.

Aufgabe: Versuchen Sie das Programm nachzuvollziehen. Denken Sie daran, dass Programm mittels `catkin_make` zu übersetzen vor dem Ausführen.

```cpp
#include <ros/ros.h>
#include <std_msgs/String.h>


int main(int argc, char **argv)
{
   // initialisierung des ros knotens
   ros::init(argc, argv, "my_string_publisher");

   // initialisierung des node handles zur Kommunikation mit dem ROSCORE
   ros::NodeHandle n;

   // Initialisierung für den ROS-Publisher zum veröffentlichen von String-Nachrichten
   ros::Publisher string_pub = n.advertise<std_msgs::String>("hello_world_topic", 1);

   // Initialisierung der Wiederholfrequenz des Programms
   ros::Rate loop_rate(10);

   // Programm wird solange ausgeführt, bis die Funktion ros::ok() false zurückliefert
   while(ros::ok())
   {
      std_msgs::String msg;               // Anlegen der Standard-Nachricht String
      msg.data = "Hello World";           // Füllen der Nachricht
      string_pub.publish(msg);            // Absenden der Nachricht

      // Ausgabe der Nachricht im Terminal
      std::cout << "Publish: " << msg.data << std::endl;

      loop_rate.sleep();                  // Warten bis zum nächsten Durchlauf der Schleife
   }

   return 0;
}
```

Einführungsbeispiel von Github: https://github.com/christianpfitzner/hello_ros

Einführung in ROS

Hello Ros in C++

```
#include <ros/ros.h>
#include <std_msgs/String.h>
```

Jedes C++ Programm beginnt zunächst mit dem Einbinden von Bibliotheken. In diesem Beispielprogramm werden zwei eingebunden: `ros/ros.h` muss immer dann hinzugefügt werden, wenn es sich um einen C++-basierten ROS-Knoten handelt. Der Include `std_msgs/String.h` ist eine vordefinierte Nachricht für die Kommunikation zwischen den Knoten -> ein ROS-Topic.

```
int main(int argc, char **argv)
{
   // initialisierung des ros knotens
   ros::init(argc, argv, "my_string_publisher");
   …
```

Jedes C++ Programm muss die Funktion `main` enthalten, so auch ein C++-Knoten. `argc` und `argv` sind Übergabeargumente. Mit der Funktion `ros::init` erhält der Knoten seinen Namen zur Betriebslaufzeit. Den Namen sollten Sie sehen, wenn Sie `$ rostopic list`

```
   // initialisierung des node handles zur Kommunikation mit dem ROSCORE
   ros::NodeHandle n;

   // Initialisierung für den ROS-Publisher zum veröffentlichen von String-Nachrichten
   ros::Publisher string_pub = n.advertise<std_msgs::String>("hello_world_topic", 1);
```

Für die Kommunikation mit dem ROSCORE und anderen ROS-Knoten wird ein ros::NodeHandle benötigt. In diesem Fall wird ein String versendet über ein Objekt ros::Publisher.

Wir sehen uns hier nochmal im Detail an, wie ein ROS-Publisher angelegt und auch initialisiert wird. Meistens werden Publisher direkt nach der Init-Funktion und dem Anlegen eines NodeHandles angelegt.

```
ros::Publisher string_pub = n.advertise<std_msgs::String>("hello_world_topic", 1);
```

Publisher-Objekt im Namespace ros

Variablenname für den Publisher, welcher frei wählbar ist

Die Funktion advertise des NodeHandles initialisiert das ros::Publisher-Objekt

Eine C++ Template-Funktion besteht immer aus zwei <>-Klammern. Hier muss angegeben warden von welchem Datentyp die Nachricht ist.

Topic-Name unter dem die Nachricht versendet wird.

Queue-Size (kann meistens auf 1 belassen werde)

Einführung in ROS

Hello Ros in C++

```
// Programm wird solange ausgeführt, bis die Funktion ros::ok() false zurückliefert
while(ros::ok())
{
   std_msgs::String msg;                // Anlegen der Standard-Nachricht String
   msg.data = "Hello World";            // Füllen der Nachricht
   string_pub.publish(msg);             // Absenden der Nachricht

   loop_rate.sleep();                   // Warten bis zum nächsten Durchlauf der Schleife
}
```

Die Hauptschleife des Programms wird erst dann verlassen, wenn die Funktion `ros::ok()` den Wert `false` zurückliefert – zum Beispiel durch das Beenden des Knoten durch <strg> + <c>.

In der Schleife wird in diesem Programm eine Nachricht vom Typ std_msgs::String erzeugt (Achtung: Header muss eingebunden sein). Die Definition der Nachricht sehen Sie im ROS-Wiki: http://docs.ros.org/en/melodic/api/std_msgs/html/msg/String.html.
Zum Versenden muss letztlich noch die Funktion publish des Objekt `string_pub` aufgerufen werden.

Die Funktion `sleep()` bremst die While-Schleife, da diese ansonsten mit möglichst hoher Geschwindigkeit unnötig die CPU belastet.

Übung

1. Ändern Sie das Programm ab, so dass der Knoten keinen String sondern
    1. Bool -> http://docs.ros.org/en/melodic/api/std_msgs/html/msg/Bool.html
    2. UInt16 -> http://docs.ros.org/en/melodic/api/std_msgs/html/msg/UInt16.html
    3. ColorRGBA -> http://docs.ros.org/en/melodic/api/std_msgs/html/msg/ColorRGBA.html
    4. Header -> http://docs.ros.org/en/melodic/api/std_msgs/html/msg/Header.html versendet, beachten Sie, dass Sie das Programm nach einer Änderung wieder neu Übersetzen und Starten müssen.
2. Ändern Sie die Nachricht zur Laufzeit des Programms in irgendeiner Weise und überprüfen Sie das Ergebnis mittels der bekannten ROS-Topic-Befehle.

Einführung in ROS

Hello Ros in Python

Das Programmbeispiel auf der rechten Seite zeigt die gleiche Funktion: Ein String wird mittels ROS unter dem Topic-Namen `hello_world_topic` veröffentlicht.

Hauptkomponenten in diesem Programm ist eine `while`-Schleife, welche mit 10 Hz Zykluszeit den String versendet.

Der Code am Anfang der Funktion ist zur Initialisierung des Knotens, sowie zum Anlegen eines Publishers vorhanden.

Aufgabe:

Versuchen Sie das Programm nachzuvollziehen. Machen Sie kleine Änderungen. Ändern Sie den Inhalt der Nachricht, oder die Frequenz mit der die Nachricht veröffentlicht wird.

```python
#!/usr/bin/env python

import rospy
from std_msgs.msg import String

def talker():
    # initialisierung des ros knotens
    rospy.init_node('my_string_publisher_python', anonymous=True)
    # Initialisierung des Publishers
    pub = rospy.Publisher('hello_world_topic', String, queue_size=10)
    # Dauer einer Schleife festlegen
    rate = rospy.Rate(10) # 10hz
    # Schleife mit Abbruchkriterium
    while not rospy.is_shutdown():
        # Vorbereiten der Nachricht
        hello_str = "hello world"
        pub.publish(hello_str)
        # Warten bis Schleifenzeit erreicht ist
        rate.sleep()

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

Einführung in ROS

Standarddatentypen für ROS_MSG

Ein großer Vorteil von ROS ist das zur Verfügung stellen von standardisierten Schnittstellen zwischen ausführbaren Programmen, auch wenn diese in unterschiedlichen Programmiersprachen geschrieben sind. Diese Folie enthält eine Auflistung aller vorhandener Nachrichten im ROS-Paket std_msgs. Die ausführliche Dokumentation hierzu findet sich im ROS-Wiki: http://wiki.ros.org/std_msgs

Auf Basis von diesen und auch weiteren Nachrichten können auch eigene Nachrichtentypen aufgebaut werden.

std_msgs/String Message

File: std_msgs/String.msg

Raw Message Definition

```
string data
```

Die ausführliche Dokumentation zum Datentyp std_msgs/String

ROS Message Types

Bool
Byte
ByteMultiArray
Char
ColorRGBA
Duration
Empty
Float32
Float32MultiArray
Float64
Float64MultiArray
Header
Int16
Int16MultiArray
Int32
Int32MultiArray
Int64
Int64MultiArray
Int8
Int8MultiArray
MultiArrayDimension
MultiArrayLayout
String
Time
UInt16
UInt16MultiArray
UInt32
UInt32MultiArray
UInt64
UInt64MultiArray
UInt8
UInt8MultiArray

Für die Services in ROS stehen standardmäßig nur drei Typen zur Verfügung. Die ausführliche Dokumentation findet sich im ROS-Wiki.

Empty ist wie der Name sagt ein leerer Service. Er kann dann zum Einsatz kommen, wenn bei einem Service direkt keine Information ausgetauscht werden muss. Es reicht, wenn es bei einem anderen ROS-Knoten ankommt.

```
# empty request - this is just a comment
---
# empty response - this is just a comment
```

SetBool versendet eine booleschen Wert. Als Ergebnis erhält der Service zum einen ob der Wert erfolgreich gesetzt werden konnte, und ggf. noch eine Nachricht in Form eines Strings.

```
bool data      # e.g. for hardware enabling / disabling
---
bool success   # indicate successful run of triggered service
string message # informational, e.g. for error messages
```

Trigger versendet einen leeren Service-Request. Als Antwort erhält der Service einen booleschen Wert, ob das Triggern erfolgreich war, und dazu eine Nachricht in Form eines Strings.

```
---
bool success   # indicate successful run of triggered service
string message # informational, e.g. for error messages
```

Einführung in ROS

Service-Server

Der C++-ROS-Knoten ist ein Beispiel für einen einfachen Service-Server: Er kann ROS-Service-Anfragen empfangen, bearbeiten und je nach gewähltem Datentyp auch eine Antwort zurücksenden.

Dieses Programm besitzt keine while-Schleife. Durch den Aufruf von ros::spin() beendet sich das Programm jedoch auch nicht.

Das Codebeispiel finden Sie auf Github.

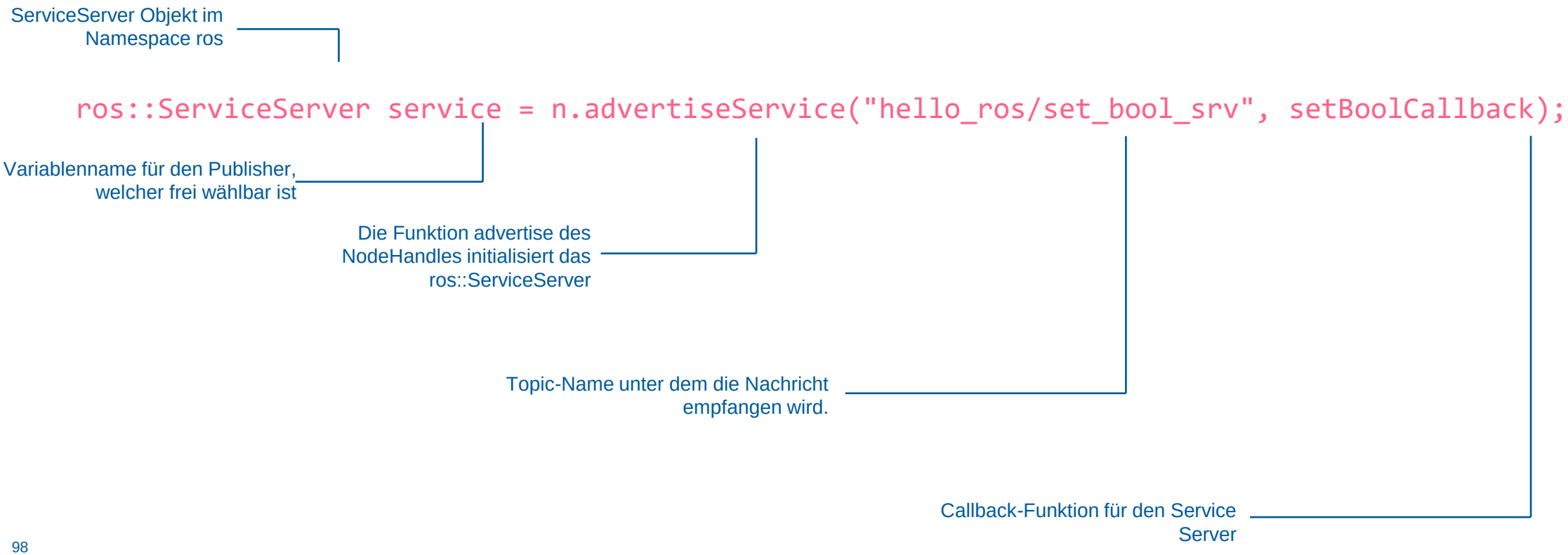
Aufgabe:

Versuchen Sie das Programm nachzuvollziehen. Denken Sie daran, dass Programm mittels catkin_make zu übersetzen vor dem Ausführen. Ändern Sie den Knoten ab, so dass er eine Service-Anfrage vom Typ std_srvs/Trigger empfangen kann.

```cpp
#include <ros/ros.h>
#include <std_srvs/SetBool.h>

bool g_variable = false;

bool setBoolCallback(std_srvs::SetBool::Request  &req, std_srvs::SetBool::Response &res)
{
  /* hier wird abgefragt, ob die Variable bereits den neuen Wert besitzt
     Falls das der Fall ist, so gibt der Service einen Fehler zurück */
  if(g_variable == req.data)
  {
    res.success = false;
    res.message = std::string("Variable already has the requested value");
    ROS_INFO_STREAM("[Server] Variable already has the requested value");

    return true;
  }

  g_variable = req.data;

  // Ausgabe im Terminal
  ROS_INFO_STREAM("[Server] New value of variable is: " << g_variable);

  // Füllen der Response Nachricht
  res.success = req.data;
  res.message = std::string("Variable was successfully set to " + static_cast<int>(g_variable));

  return true;
}


int main(int argc, char **argv)
{
  // initialisierung des ros Knotens
  ros::init(argc, argv, "service_server");

  // Initialisierung des node handles zur Kommunikation mit dem ROSCORE
  ros::NodeHandle n;

  // Initialisierung für den ROS-ServiceServers zum Behandeln von eingehenden Service-Anfragen
  ros::ServiceServer service = n.advertiseService("hello_ros/set_bool_srv", setBoolCallback);

  // Ausgabe im Terminal, dass der Knoten läuft.
  ROS_INFO_STREAM("Service server is ready.");

  // das Programm wird ausgeführt und lauscht dabei auf eingehende Service Nachrichten.
  ros::spin();

  return 0;
}
```

Einführung in ROS

Anlegen eines ROS Service Servers in C++



Einführung in ROS

Service-Client

Der dazugehörige Client erzeugt in der `while`-Schleife eine Serviceanfrage. Die Anfrage eines Service entsteht immer im Element

`srv.request. ….`

die dazugehörige Antwort

`srv.response. …`

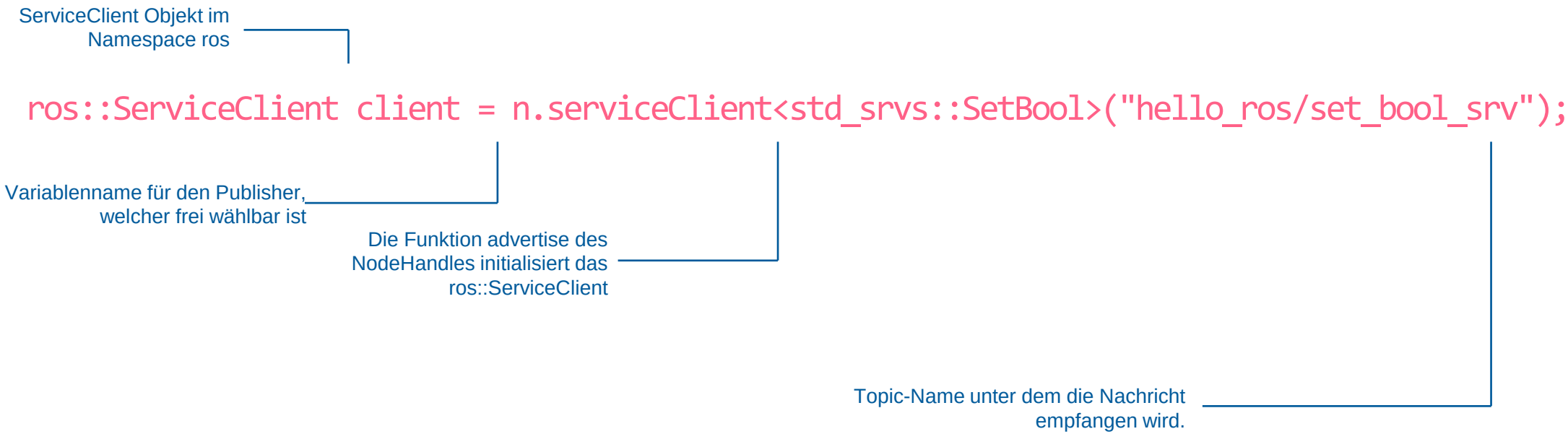
füllt der Service-Server. Die Antwort wird in diesem Code-Beispiel im Terminal ausgegeben

Aufgabe:

Wenn Sie den Service-Server bereits auf `std_srvs/Trigger` angepasst haben, passen Sie diesen Knoten auch an, so dass eine sinnvolle Kommunikation entstehen kann.

```cpp
#include <ros/ros.h>
#include <std_srvs/SetBool.h>



int main(int argc, char **argv)
{
  // initialisierung des ros Knotens
   ros::init(argc, argv, "service_client");

   // Initialisierung des node handles zur Kommunikation mit dem ROSCORE
   ros::NodeHandle n;

  // Initialisierung für den ROS-ServiceClient zum Versenden von Service-Nachrichten
   ros::ServiceClient client = n.serviceClient<std_srvs::SetBool>("hello_ros/set_bool_srv");
   std_srvs::SetBool srv;

   // Loop-Rate mit
   ros::Rate loop_rate(0.2);


   unsigned int i=0;
   while(ros::ok())
   {
      if(      i == 0) srv.request.data = true;
      else if (i == 1) srv.request.data = false;
      else if (i == 2) srv.request.data = false;

      if (client.call(srv))
      {
         ROS_INFO_STREAM("Variable changed to "          << static_cast<int>(srv.response.success));
         ROS_INFO_STREAM("Message from service server: " << srv.response.message);
      }
      else
      {
         ROS_ERROR("Failed to call service hello_ros/set_bool_srv");
      }

      loop_rate.sleep();


      i++;              // Iterator um eins hochzählen
      i = i%3;          // Iterator mit dem Modulo 3 verrechnen.
   }

   return 0;
}
```

Anlegen eines ROS Service Clients in C++



Start eines oder mehrerer ROS-Knotens mittels ROSLAUNCH

Der Start von **mehreren Knoten** erfordert bisher auch immer ein neues Terminal zu öffnen. In einem realen Robotersystem können ohne Probleme bis zu 100 Knoten laufen. Aus diesem Grund bietet ROS die Möglichkeit mehrere Knoten über ein so genanntes Launch-File zu starten. Der Aufruf ist analog zu `rosrun`. Beachten Sie, dass Sie beim Start eines Launch-Files keinen extra ROSCORE zuvor gestartet haben müssen; falls keiner vorhanden ist, wird dieser mitgestartet.

```
roslaunch turtlesim turtlesim_node
```

Befehl zum Ausführen eines ROS-Programms. Der Befehl roslaunch funktioniert von jedem Ort aus dem Terminal heraus.

Name des ROS-Pakets, in dem sich das Launch-File befindet

Name der Launch-Datei

Das folgende Beispiel starte einmal Turtlesim, sowie das dazugehörige Programm zum Steuern der Schildkröte über die Tastatur.

```
<launch>
    <node pkg="turtlesim"   type="turtlesim_node"    name="turtlesim_node" />
    <node pkg="turtlesim"   type="turtle_teleop_key" name="keyboard_ctrl" />
</launch>
```

Die Datei heißt `turtle.launch` – wichtig ist, dass die Dateiendung `.launch` zum Einsatz kommt.

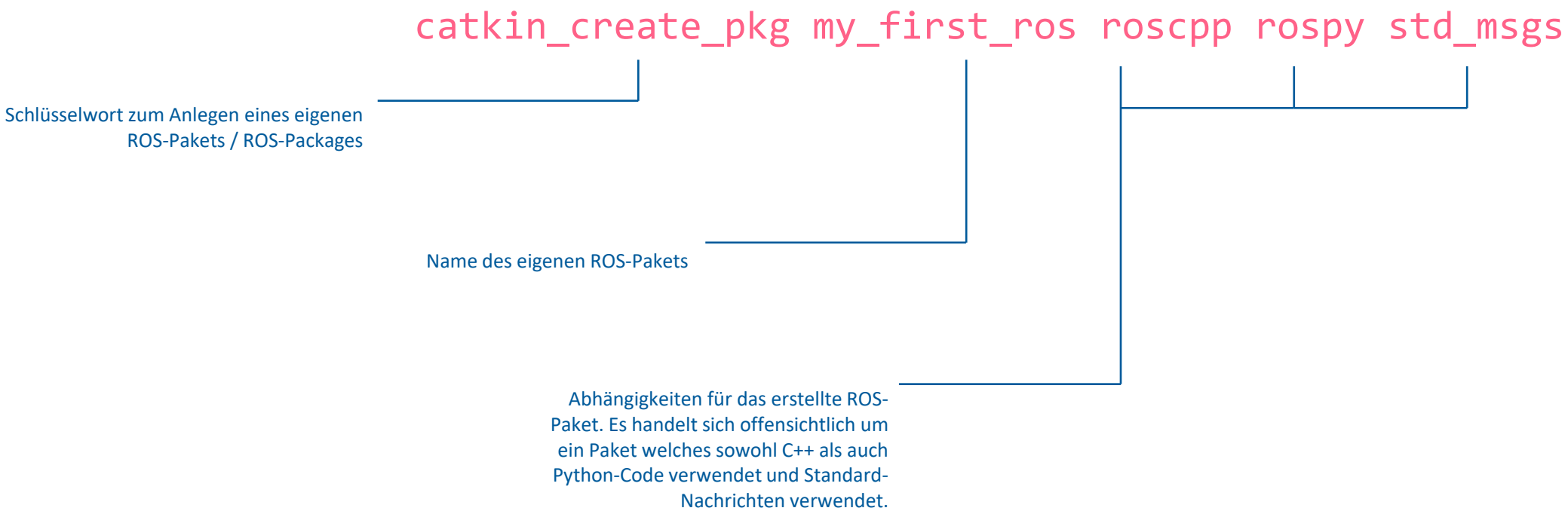
Neben dem einfachen Starten von mehreren ROS-Knoten können auch Parameter gleichzeitig an verschiedene Knoten übergeben werden, oder Namensräume gebildet werden. Die ausführliche Dokumentation findet sich im ROS-Wiki.

Zur Erstellung von Launch-Files empfehle ich in VSCode folgendes Plugin zu installieren: `pojar.ros-snippets`. Dieses bietet eine Syntaxvervollständigung und Code-Snippets an, so dass sich in wenigen Sekunden auch komplexere Dateien erstellen lassen.

Aufgabe:

Schreiben Sie eine Launch-Datei zum Aufruf des bekannten Hello_ROS-Beispiels. Erzeugen Sie hierzu im Ordner des ROS-Pakets den Ordner Namens launch.

Eigene Pakete lassen sich in wenigen Schritten initialisieren: Das Schlüsselwort `catkin_create_pkg`, gefolgt vom gewünschten Paketnamen, und beliebig vielen Abhängigkeiten stellt eine Ordnerstruktur und einige essentielle Dateien als Template im Ordner `catkin_ws/src` zur Verfügung.



Package.xml

Die Datei `package.xml` beinhaltet Abhängigkeiten zu anderen ROS-Paketen. Sie wird automatisch erstellt, wenn jemand ein neues ROS-Paket anlegt.

Neben dem Namen – hello_ros – und den Abhängigkeiten – hier zum Beispiel zu `std_msgs`, `roscpp` und `rospy` – beinhaltet die Datei auch den Autor und Maintainer des Pakets, wie die aktuelle Version und im Idealfall eine kurze Beschreibung.

Wenn beim **Anlegen eines Paketes** alle notwendigen Abhängigkeiten korrekt angegeben wurden, dann besteht zunächst kein Bedarf diese Datei anzupassen. Änderungen können nachher händisch durchgeführt werden.

```xml
<?xml version="1.0"?>
<package>
  <name>hello_ros</name>
  <version>1.0.0</version>
  <description>The hello_ros package</description>

  <!-- One maintainer tag required, multiple allowed, one person per tag -->
  <!-- Example:  -->
  <!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
  <maintainer email="christian.pfitzner@thi.de">Prof. Dr. Christian Pfitzner</maintainer>


  <license>BSD</license>


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

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>rospy</build_depend>

  <run_depend>std_msgs</run_depend>
  <run_depend>rospy</run_depend>

  <export>


  </export>
</package>
```

package.xml aus dem hello_ros-Beispiel

CMakeLists.txt

Beim Anlegen eines neuen ROS-Pakets entsteht auch eine Datei mit der Bezeichnung `CMakeLists.txt`. Der Dateiname sollte nicht verändert werden. CMakeLists ist eine Datei, welche von Cmake und ROS dazu verwendet wird ausführbare Programme auf Basis von Quellcode zu generieren.

Die Datei ist als Template zu sehen. An einigen entscheidenden Stellen müssen händisch nach dem Erstellen eines Paketes Änderungen und Ergänzungen gemacht werden.

Die vollständige Dokumentation zu CMake im Kontext von ROS findet sich im ROS-Wiki.

```
cmake_minimum_required(VERSION 2.8.3)
project(hello_ros)
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
catkin_python_setup()

add_service_files(
   FILES
   SumTwoNumbers.srv
)
generate_messages(
   DEPENDENCIES
)

catkin_package(
  CATKIN_DEPENDS message_runtime
)

include_directories(
  ${catkin_INCLUDE_DIRS}
)

## Declare a C++ executable
add_executable(publisher_node                   src/string_publisher.cpp)
add_executable(subscriber_node                  src/string_subscriber.cpp)
add_executable(service_server_node              src/service_server.cpp)
add_executable(service_client_node              src/service_client.cpp)

## Specify libraries to link a library or executable target against
target_link_libraries(publisher_node                    ${catkin_LIBRARIES} )
target_link_libraries(subscriber_node                   ${catkin_LIBRARIES} )
target_link_libraries(service_server_node               ${catkin_LIBRARIES} )
target_link_libraries(service_client_node               ${catkin_LIBRARIES} )
```

CMakeLists.txt aus dem hello_ros-Beispiel

ROS_LOG

Üblicherweise werden Ausgaben in Programmen in C++ mittels der Funktion `printf()` oder `std::cout` getätigt. ROS bietet hierzu eine Erweiterung an: ROS_LOG beinhaltet neben der Ausgabe im Terminal auch die Möglichkeit diese Ausgabe in einer Logdatei umzuleiten und ein so genanntes Log-Level einzustellen.

INFO

```
ROS_INFO("Hello %s", "World");
ROS_INFO_STREAM("Hello " << "World");
```

```
rospy.loginfo(msg, *args, **kwargs)
```

DEBUG

```
ROS_DEBUG("Hello %s", "World");
ROS_DEBUG_STREAM("Hello " << "World");
```

```
rospy.logwarn(msg, *args, **kwargs)
```

WARNING:

```
ROS_WARNING("Hello %s", "World");
ROS_WARNING_STREAM("Hello " << "World");
```

```
rospy.logdebug(msg, *args, **kwargs)
```

ERROR

```
ROS_ERROR("Hello %s", "World");
ROS_ERROR_STREAM("Hello " << "World");
```

```
rospy.logerr(msg, *args, **kwargs)
```

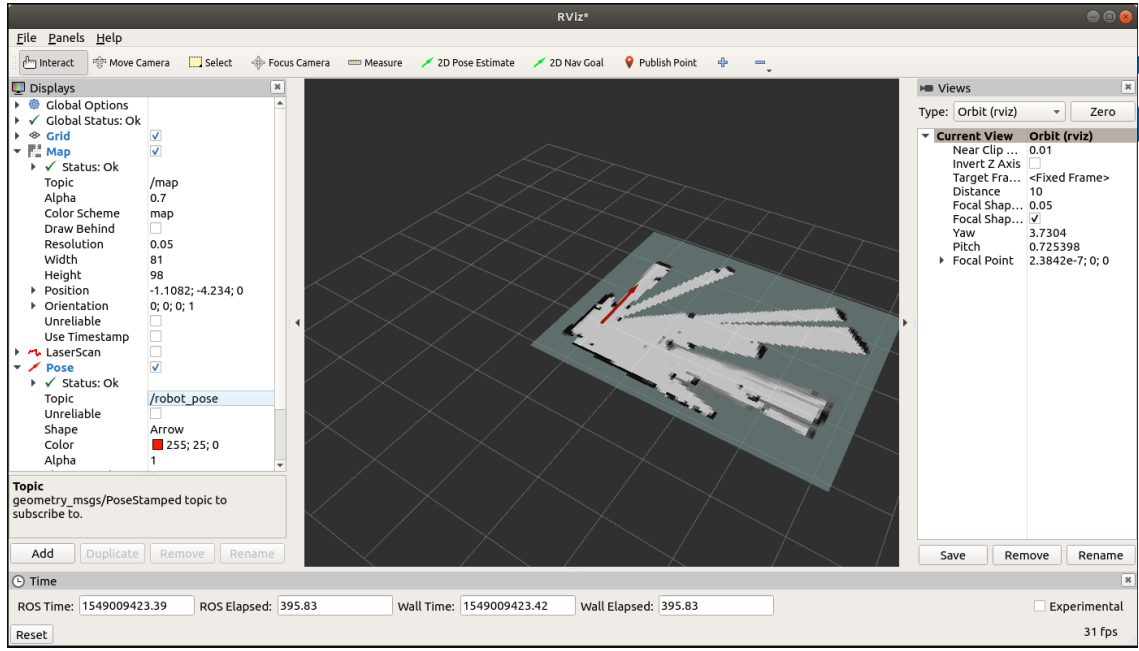
Die Vollständige Dokumentation zu ROS_LOG finden Sie im Wiki für C++ oder Python.

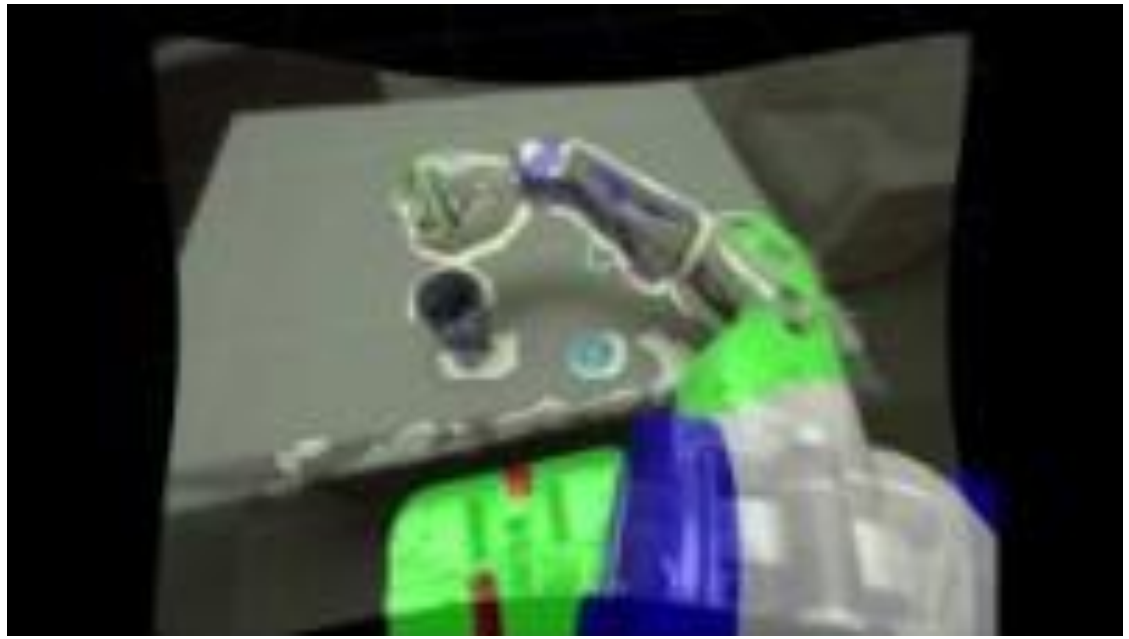
Einführung in ROS

RVIZ

Viele der Nachrichten sind zu **komplex** um sie im Terminal auszuwerten oder zu debuggen. Aus diesem Grund stellt ROS auch eine **Visualisierung** von vielen Nachrichten zur Verfügung. RVIZ (ROS Visualization) kann typische **Sensornachrichten** – zum Beispiel Kameras, LIDAR, Ultraschalldistanzsensoren, 3D-Kameras – Umgebungskarten oder Pfade, sowie die aktuelle Position eines Roboters visualisieren.

`Rviz` wird mittels des Befehls rviz im Terminal gestartet. Die vollständige Dokumentation findet sich im ROS-Wiki.





https://www.youtube.com/watch?v=i--Sd4xH9ZE&feature=emb_logo

Aufgabe:

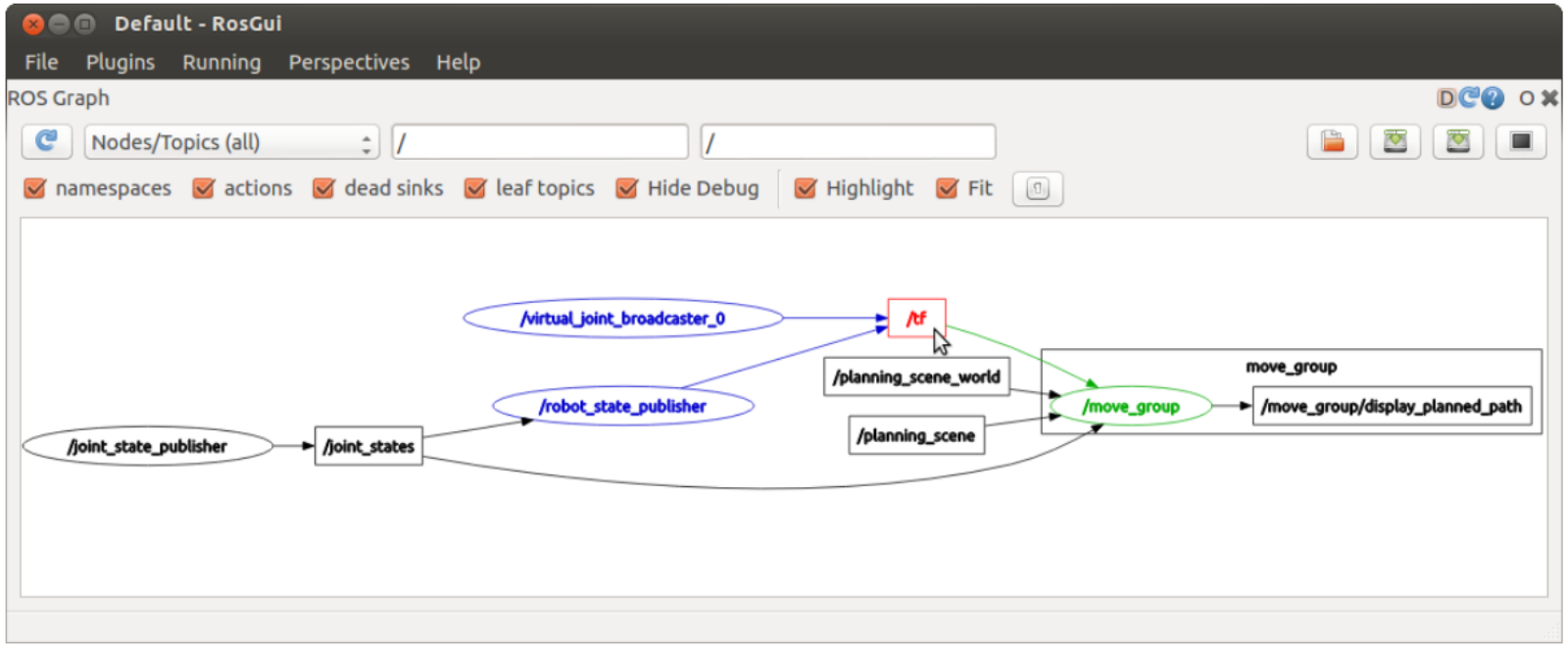
Zeigen Sie sich die Position der Schildkröte von Turtlesim in RVIZ an.

Mehrere Knoten interagieren in einem Robotersystem bei ROS miteinander. Das debuggen über das Terminal hat seine Grenzen, wenn die Anzahl der Knoten besonders groß (> 10 wird). ROS bietet hierzu die grafische Oberfläche RQT-Graph zur Verfügung. Diese zeigt alle aktiven Knoten, sowie die Kommunikation über ROSTOPICs grafisch dar. Nicht enthalten in dem Tool sind die Verbindungen über Services.

Zum Start muss lediglich `rqt-graph` im Terminal eingegeben werden. Die komplette Dokumentation hierzu findet sich im ROS-Wiki.

Knoten werden hier als Ellipsen dargestellt. Die Nachrichten zwischen Knoten dagegen als Rechtecke.

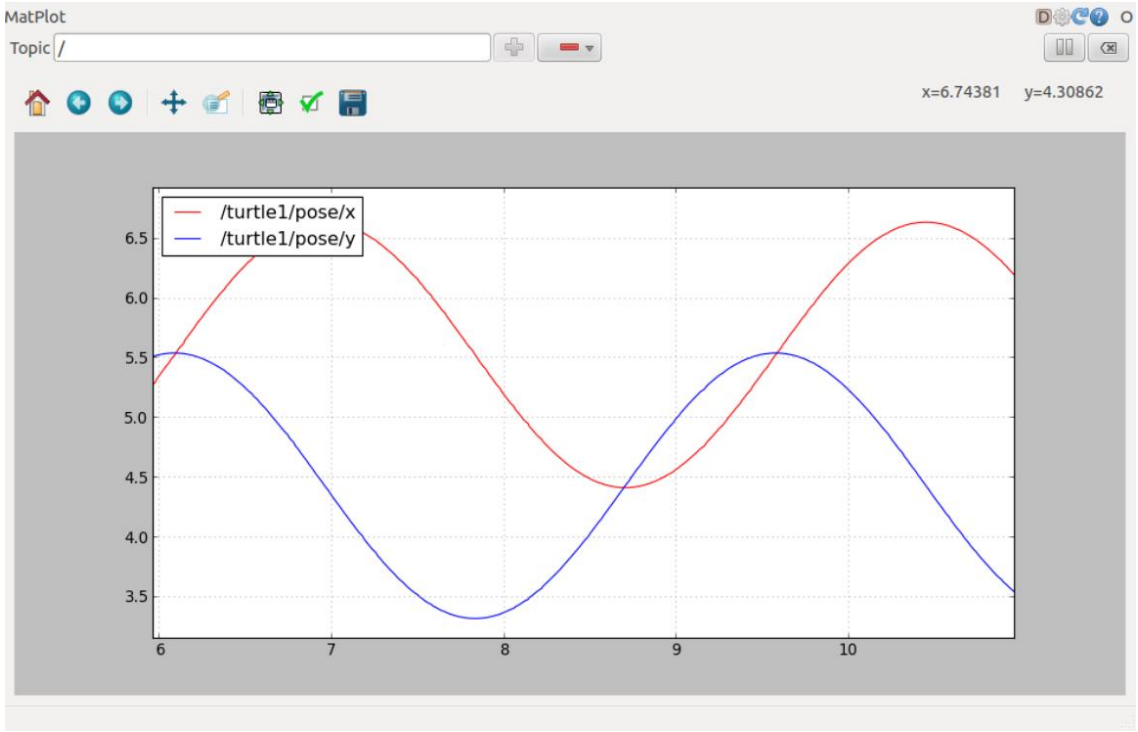
Über das Menü im oberen Teil der Anzeige können verschiedene Knoten herausgefiltert werden, um für Übersichtlichkeit zu sorgen.



Einführung in ROS

RQT-PLOT

Mittels einfacher Dateioperationen könnten Daten aus rostopic oder rosservice in eine CSV-Datei zum Plotten von Werten gespeichert werden. Der einfachere Weg läuft über das Werkzeug rqt_plot, welches auch über das Terminal gestartet werden kann. Die umfangreiche Dokumentation hierzu findet sich im ROS-Wiki.



Aufgabe:

Visualisieren Sie die aktuelle Position einer Schildkröte mit der x- und y-Koordinate in turtlesim über die Zeit in rqt_plot .

Einführung in ROS

Turtlesim

Im Folgenden soll ein Knoten auf Basis eines Templates programmiert werden, welcher eine Schildkröte mittels eines Service erzeugt, und diese im Anschluss im Kreis fahren lässt.

Die Vorlage können Sie über Github in den Workspace von ROS herunterladen.

https://github.com/christianpfitzner/turtle_control

oder direkt in den catkin-Workspace:

```
$ git clone https://github.com/christianpfitzner/turtle_control.git
```

Weitere Aufgaben:

Programmieren Sie eine Spiralbewegung der Schildkröte, oder eine mäanderförmige Bewegung, die zum Beispiel die Bewegung eines Staubsaugerroboters zur Reinigung einer quadratischen Fläche nachahmt.

```cpp
int main(int argc, char **argv)
{
  if(argc<2)
  {
    std::cout << "Too less parameters. Expected: " << argv[0] << " <turtlename>" << std::endl;
    return -1;
  }


  /**
   * Name des eigenen Knotens
   */
  char node[64];
  sprintf(node, "%s%s", argv[1], "_node");
  printf("Neuer Knoten: %s\n", node);

  ros::init(argc, argv, node);
  ros::NodeHandle n;

  …
```

Programmierübung in C++

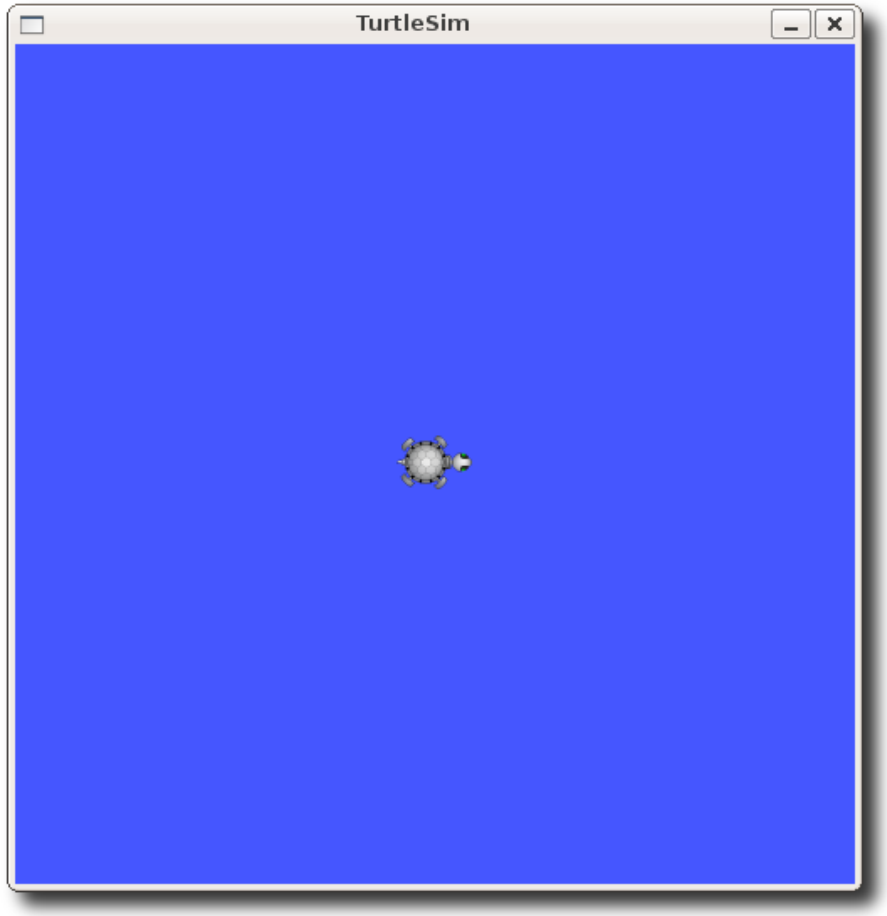
```python
import rospy                            # general include to proceed with ros and python
import sys                              # neccesary for sys command
from geometry_msgs.msg import Twist     # twist message
from turtlesim.msg import Pose          # pose message
from turtlesim.srv import Spawn         # spawn service from turtlesim


# callback function to be performed on each pose callback
def pose_callback_function(msg):
    rospy.loginfo(rospy.get_caller_id() + "received: %s", msg)


# main function to initialize the rosnode and perform turtle circle loop
def node(turtlename):
    # initialisierung des ros knotens
    rospy.init_node('turtle_control_node', anonymous=True)

    # generate the topics based on the turtles name
    turtle_twist_topic = turtlename + '/cmd_vel'
    turtle_pose_topic  = turtlename + '/pose'
    …
```

Programmierübung in Python

Einführung in ROS

Turtlesim

Turtlesim ist eine einfache Simulationsumgebung, die bei der Basisinstallation von ROS mit installiert wird. Die Schildkröte ist das Wahllogo von ROS und mit jeder neuen ROS-Version kommt auch ein neues Schildkrötenmaskotchen hinzu. Die Schildkröten sind ausgestattet mit einem Differentialantrieb – dazu später noch mehr -- , und können daher sich auf der Stelle drehen, sowie geradeaus, oder Kurvenfahrten durchführen.



Übung:

Untersuchen Sie alle Nachrichten in Turtlesim. Wie können Sie die Schildkröte mit einem Befehl im Terminal bewegen? Lassen Sie die Schildkröte eine Kreisbewegung machen.

Einführung in ROS

Turtlesim: Anfahren eines Punktes

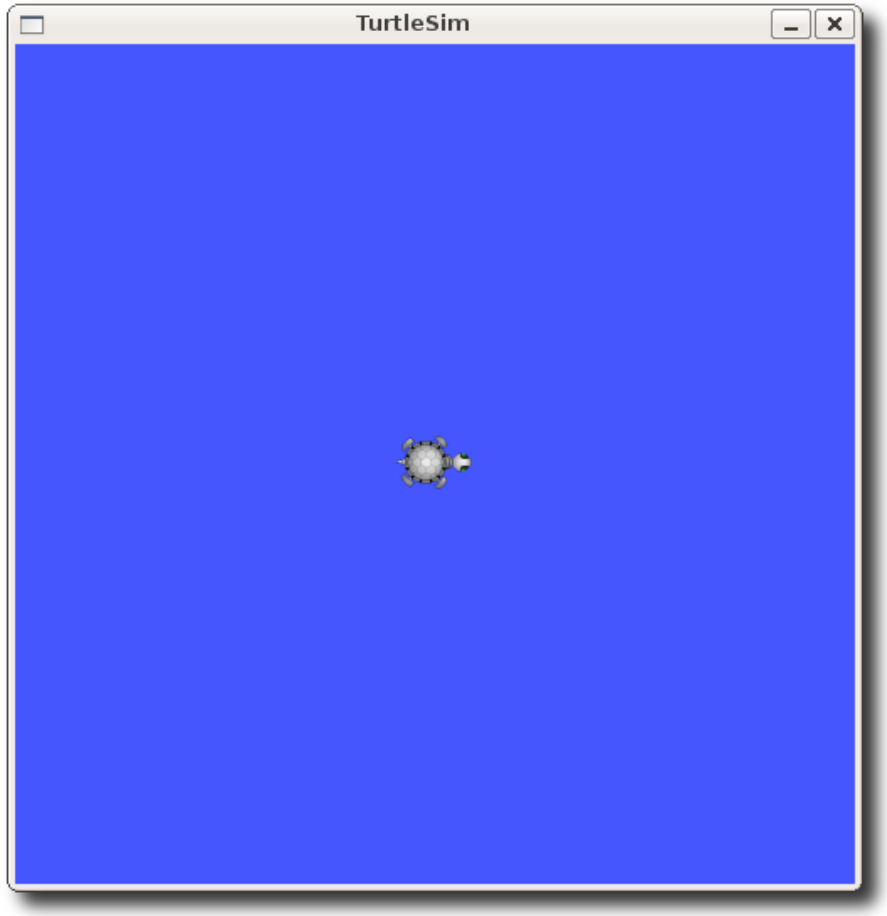
In einer weiteren Programmierübung sollen Sie die Schildkröte nun einen Punkt in der Bildebene anfahren lassen.

Hierzu benötigen Sie die aktuelle Position der Schildkröte $\mathbf{p} = (x \;\; y)^T$ – diese ist über ein ROSTOPIC verfügbar – sowie einen vorgegebenen Punkt $\mathbf{p}' = (x' \;\; y')^T$ im Koordinatensystem von Turtlesim. Die Gleichungen für die lineare Geschwindigkeit ergibt sich aus der euklidischen Distanz zwischen Roboter-Pose und Ziel. Ein Faktor K ist als P-Regler zu verstehen.

$$v_x = K\sqrt{(x - x')^2 + (y - y')^2}$$

Die Orientierung berechnet sich mittels des Arcus-Tangens.

$$\theta = \mathrm{atan}(\frac{y' - y}{x' - x})$$



Die wichtigsten ROS-Befehle auf einen Blick

Die wichtigsten ROS-Befehle auf einen Blick

ROS Kinetic 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/rossrv

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
```

Logging Tools

rosbag

A set of tools for recording and playing back of ROS topics.

Commands:

rosbag record	Record a bag file with specified topics.
rosbag play	Play content of one or more bag files.
rosbag compress	Compress one or more bag files.
rosbag decompress	Decompress one or more bag files.
rosbag filter	Filter the contents of the bag.

Examples:

Record select topics:

```
$ rosbag record topic1 topic2
```

Replay all messages without waiting:

```
$ rosbag play -a demo_log.bag
```

Replay several bag files at once:

```
$ rosbag play demo1.bag demo2.bag
```

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
```

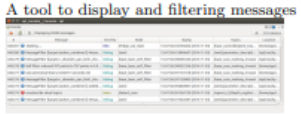
Die wichtigsten ROS-Befehle auf einen Blick

Die wichtigsten ROS-Befehle auf einen Blick

Logging Tools

rqt_console

A tool to display and filtering messages published on rosout.



Usage:

```
$ rqt_console
```

rqt_bag

A tool for visualizing, inspecting, and replaying bag files.

Usage, viewing:

```
$ rqt_bag bag_file.bag
```

Usage, bagging:

```
$ rqt_bag *press the big red record button.*
```

rqt_logger_level

Change the logger level of ROS nodes. This will increase or decrease the information they log to the screen and rqt_console.

Usage:

```
viewing $ rqt_logger_level
```

Introspection & Command Tools

rqt_topic

A tool for viewing published topics in real time.

Usage:

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

rqt_msg, rqt_srv, and rqt_action

A tool for viewing available msgs, srvs, and actions.

Usage:

```
$ rqt
Plugin Menu->Topic->Message Type Browser
Plugin Menu->Service->Service Type Browser
Plugin Menu->Action->Action Type Browser
```

rqt_top

A tool for ROS specific process monitoring.

Usage:

```
$ rqt
Plugin Menu->Introspection->Process Monitor
```

rqt_publisher, and rqt_service_caller

Tools for publishing messages and calling services.

Usage:

```
$ rqt
Plugin Menu->Topic->Message Publisher
Plugin Menu->Service->Service Caller
```

rqt_reconfigure

A tool for dynamically reconfiguring ROS parameters.

Usage:

```
$ rqt
Plugin Menu->Configuration->Dynamic Reconfigure
```

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
```

Development Environments

rqt_shell, and rqt_py_console

Two tools for accessing an xterm shell and python console respectively.

Usage:

```
$ rqt
Plugin Menu->Miscellaneous Tools->Shell
Plugin Menu->Miscellaneous Tools->Python Console
```

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
```

rqt_plot

A tool for plotting data from ROS topic fields.

Examples:

To graph the data in different plots:

```
$ rqt_plot /topic1/field1 /topic2/field2
```

To graph the data all on the same plot:

```
$ rqt_plot /topic1/field1,/topic2/field2
```

To graph multiple fields of a message:

```
$ rqt_plot /topic1/field1:field2:field3
```

rqt_image_view

A tool to display image topics.

Usage:

```
$ rqt_image_view
```

ROS Kinetic Catkin Workspaces

Create a catkin workspace

Setup and use a new catkin workspace from scratch.

Example:

```
$ source /opt/ros/kinetic/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.gi
$ 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)
```

ROS2

ROS vs. ROS2

Anwendung	ROS	ROS2
Plattform	Getestet auf Ubuntu Beibehalten auf anderen Linux-Versionen sowie auf OS X	Aktuell getestet und unterstützt auf Ubuntu Xenial, OS X El Capitan sowie auf Windows 10
C++	C++03 // nutzt nicht die Kapazitäten von C++11 bei seinen API-Schnittstellen	Nutzt hauptsächlich C+++11 Planen und beginnen Sie mit der Nutzung von C++14 und C+++17
Python	Ziel Python 2	>= Python 3.5
Middleware	Personalisiertes Serialisierungsformat (Transportprotokoll + zentraler Feststellungsmechanismus)	Aktuell beruhen alle Implementierungen dieser Schnittstelle auf dem DDS-Standard.
Synchronisierung von Dauer und Zeitmessung	Dauer und Zeittypen werden in Client-Bibliotheken definiert und in C++ und Python programmiert.	Diese Typen werden als Nachrichten definiert und sind somit in allen Programmiersprachen einheitlich.
Komponenten mit Lebenszyklus	Jeder Knoten hat generell eine eigene Hauptfunktion.	Lebenszyklus kann bei Tools, wie z. B. roslaunch, verwendet werden, um ein System aus mehreren Komponenten deterministisch zu starten.
Multi-Thread-Modell	Der Entwickler kann sich nur zwischen Mono-Thread- und Multi-Thread-Ausführung entscheiden.	Modelle mit Zwischenstufen der Ausführung sind erhältlich und personalisierte Ausführungsprogramme können einfach implementiert werden.
Mehrfache Knoten	Es ist nicht möglich, mehr als einen Knoten in einem Prozess zu erstellen.	Es ist möglich, mehrere Knoten in einem Prozess zu erstellen.
roslaunch	Die roslaunch-Dateien werden in XML definiert und besitzen sehr begrenzte Kapazitäten.	Die Startdateien werden in Python programmiert, sodass komplexere Logiken wie bedingte Anweisungen usw. verwendet werden können.

Einfacher Publisher in ROS2

Das Code-Beispiel auf der rechten Seite zeigt einen einfachen Publisher-Knoten in ROS2: Der Framework fordert Anwender deutlich mehr dazu objektorientiert zu programmieren. Ein ROS-Knoten muss zwangsweise von der Basisklasse `rclcpp::Node` erben. Das kann zum einen als Einschränkung verstanden werden, schafft aber auch einen notwendigen Standard, so dass das Einlesen in einen Knoten einfacher wird.

Aufgabe:

Vergleichen Sie den Publisher-Knoten zwischen ROS1 und ROS2. Welche Unterschiede können Sie erkennen?

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

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

using namespace std::chrono_literals;

/* This example creates a subclass of Node and uses std::bind() to register a
* member function as a callback from the timer. */

class MinimalPublisher : public rclcpp::Node
{
  public:
    MinimalPublisher()
    : Node("minimal_publisher"), count_(0)
    {
      publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
      timer_ = this->create_wall_timer(500ms, std::bind(&MinimalPublisher::timer_callback, this));
    }

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

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