Murat Çalış

ROBOTER MIT ROS

Bots konstruieren und mit Open Source programmieren

dpunkt.verlag

Murat Çalış wurde in Heidelberg geboren. Er ist Informatiker im öffentlichen Dienst. Nebenberuflich unterstützt er Unternehmen in den Bereichen Programmierung, Informationssicherheit und Automatisierung. In seiner Freizeit beschäftigt er sich leidenschaftlich mit Robotik und experimenteller Informatik. Sein aktuelles Projekt ist ein Roboter, der selbstständig lernt.

Papier plus PDF.

Zu diesem Buch – sowie zu vielen weiteren dpunkt.büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei dpunkt.plus+:

www.dpunkt.plus

Murat Çalış

Roboter mit ROS

Bots konstruieren und mit Open Source programmieren

dpunkt.verlag

Murat Çalış
mc@pirate-robotics.net

Ergänzende Informationen, Aktualisierungen und Erweiterungen des Roboters auf:
www.piraterobotics.net

Lektorat: Gabriel Neumann
Copy-Editing: Petra Kienle, Fürstenfeldbruck
Satz: Birgit Bäuerlein
Herstellung: Stefanie Weidner
Umschlaggestaltung: Helmut Kraus, *www.exclam.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-86490-567-4
PDF 978-3-96088-467-5
ePub 978-3-96088-468-2
mobi 978-3-96088-469-9

1. Auflage 2020
© 2020 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

PEFC zertifiziert
Das Papier für dieses Buch stammt aus nachhaltig bewirtschafteten Wäldern und kontrollierten Quellen
PEFC/04-31-0810 www.pefc.de

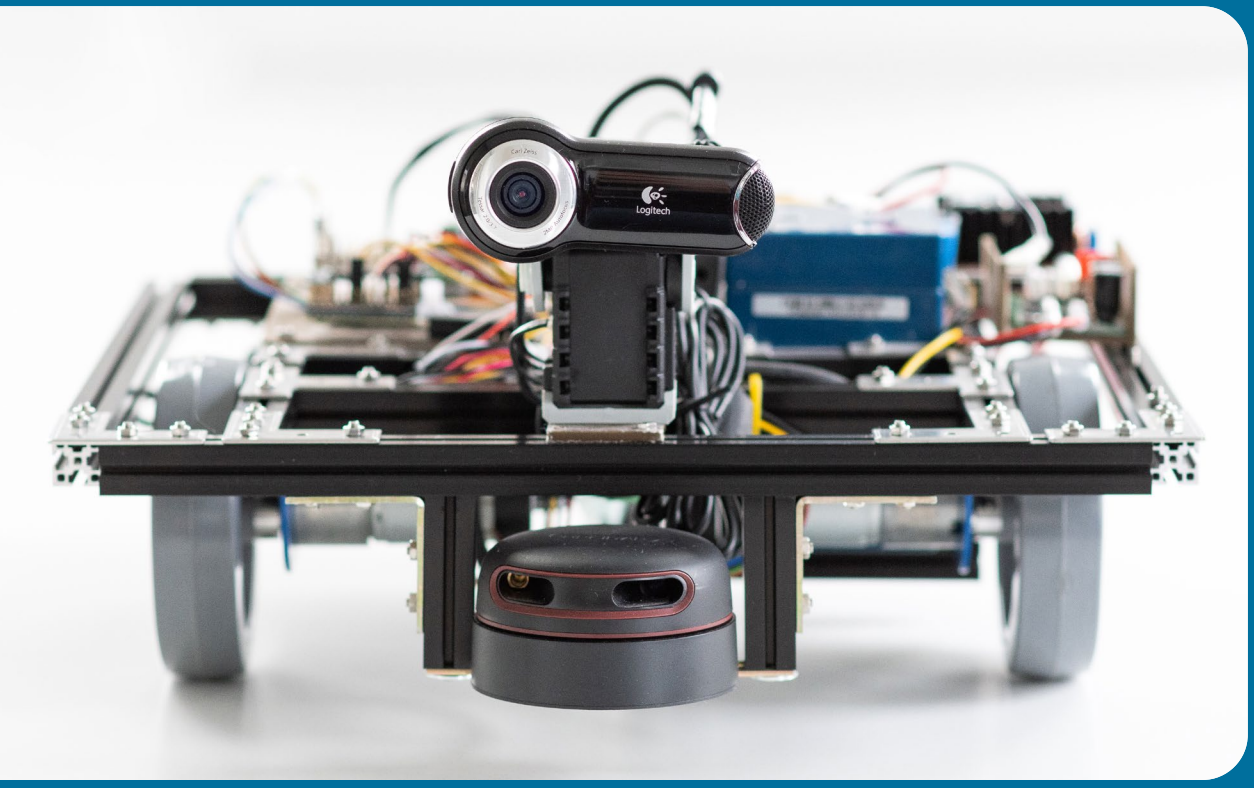
Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: *hallo@dpunkt.de*.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten.
Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

5 4 3 2 1 0

Inhaltsverzeichnis

1 ROS – Robot Operating System



Das Robot Operating System wurde entwickelt, um das Rad nicht jedes Mal neu zu erfinden. Es stehen etliche Pakete für ROS zur Verfügung, sodass Treiberentwicklungen der Vergangenheit angehören und man schneller mit den höheren Schichten der Robotik beginnen kann. Dazu gehören Gesichtserkennung, Objekterkennung, autonomes kollisionsfreies Fahren, Kartografierung, Spracherkennung und kollisionsfreie Kinematik, um nur einige zu nennen. Mittlerweile ist ROS ein De-facto-Standard in der Robotik. Die *NASA* verwendet ROS für Robonaut2 auf der *ISS*[1], um nur ein prominentes Beispiel zu nennen. ROS hat mittlerweile über 7,5 Millionen Codezeilen. Der Linux-Kernel 4.14 hat ca. 25 Millionen Zeilen. Wenn wir zehn bis 20 Jahre in die Zukunft schauen, sollte niemand mehr die hardwarenahen Schichten eines Roboters programmieren müssen.

Das Robot Operating System ist nicht, wie es der Name andeutet, ein Betriebssystem. ROS wird wie ein gewöhnliches Programm auf einem Betriebssystem installiert. Nach der Installation von ROS können eigene Robotik-Programme die Funktionalität und Bibliotheken von ROS nutzen.

ROS ist ein Robotik-Framework, basierend auf dem *publish/subscribe*-Prinzip. Darin kommunizieren Programme über ein Nachrichtensystem miteinander, vergleichbar der Interprozesskommunikation in herkömmlichen Anwendungen. Der Vorteil ist, dass der Absturz eines Programms nicht zwingend das gesamte ROS-System zum Absturz bringt.

ROS-Nachrichten werden per TCPROS, einem Protokoll basierend auf TCP/IP, übertragen und können mit Wireshark mitgelesen werden. Dies erleichtert nicht nur die Fehlersuche in verteilten ROS-Anwendungen, sondern ermöglicht auch einen Einblick in die Kommunikation zwischen den sogenannten ROS-Knoten.

Ein ROS-Netzwerk ist praktisch ungeschützt gegen Verbindungen aus dem lokalen Netzwerk, da es keine Authentifizierungsmöglichkeit wie bei HTTP gibt. Es ist daher empfehlenswert, ein VPN oder OpenVPN zum Schutz der Netzwerkkommunikation einzurichten.

Ursprünglich wurde ROS unter dem Namen Switchyard am Stanford Artificial Intelligence Laboratory entwickelt und später von Willow Garage weiterentwickelt, die auch den PR2-Roboter konstruiert haben. Seit April 2012 ist die *Open Source Robotics Foundation (OSRF)* für ROS verantwortlich. Über 3.000 Software-Pakete gibt es bereits.

1. *ISS – International Space Station*. Ein internationales Weltraumprojekt, an dem Europa, die USA, Russland, Japan und Kanada beteiligt ist. Seit 2000 ist die *ISS* permanent bemannt. Alexander Gerst war zuletzt als Kommandeur auf der Raumstation.

Mittlerweile ist ROS unter dem Begriff ROS-Industrial auch in der Produktion und in namhaften Robotern im Einsatz. Je mehr ROS in der industriellen Fertigung eingesetzt wird, desto mehr entsteht ein Bedarf an Spezialisten, die sich mit ROS auskennen.

Die skizzierten Eigenschaften von ROS bringen mit sich, dass ROS nichts für schwache Mikrocontroller ist. Wer einen Roboter bauen möchte, der Hindernissen ausweichen kann, braucht kein Robot Operating System. Dazu genügen ein Infrarotsensor, ein Arduino mit etwas Programmierlogik und ein fahrbarer Untersatz. Die Zeit, die man zum Erlernen des ROS-Systems benötigt, würde weit über die Zeit hinausgehen, die wir für die Entwicklung des eben genannten Roboters benötigen.

Meine ersten Erfahrungen mit ROS machte ich, als mein Bioloid-Premium Humanoide mit ROS aufgerüstet werden sollte. Nach langen Recherchen, welche Computerplattform es nun werden sollte, lag der *Raspberry Pi* 2012 auf meinem Tisch. Damals wurde der Kleinstrechner von ROS nicht unterstützt und man war gezwungen, die ROS-Dateien von Quellcode in Maschinencode zu kompilieren. Auf einem *Raspberry Pi* mit 256 MB RAM und mit einem wegen diverser Abstürze während des Kompilierens nicht übertakteten 700-MHz-Prozessor ist das eine langwierige Angelegenheit. Nachdem auch OpenCV für die Gesichtserkennung kompiliert war, trat die Ernüchterung bei drei bis vier Bildern pro Sekunde ein.

Die meiste Zeit programmiere ich auf einem separaten, leistungsstarken Entwicklungsrechner und teste die Ergebnisse dann auf dem kleinen *Pi*. So ist das auch heute noch, mit dem Unterschied, dass der *Raspberry Pi* zwischendurch von einem *Odroid U3* und dieser aktuell von einem *Odroid XU4* abgelöst wurde. Die Leistung hat sich innerhalb von fünf Jahren gefühlt verzehnfacht, denn es laufen mittlerweile sehr leistungshungrige ROS-Pakete auf dem Einplatinenrechner wie *Navigation*, *MoveIt!* und *OpenCV* mit akzeptablen sieben Bildern pro Sekunde.

Der Trend zu leistungsstarken Einplatinenrechnern wird weitergehen. Schon stehen Kleinstcomputer mit Intel-Atom-Prozessoren, wie der *Intel Joule* zur Verfügung. Der *Intel NUC* zählt dabei nicht zu den Kleinstrechnern, ist aber ein beliebter Robotik-Rechner mit akzeptabler Größe, in welchem Ubuntu und ROS gut zusammenspielen.

In den folgenden Kapiteln lernen wir Schritt für Schritt ROS kennen, indem wir:

- ROS installieren
- ROS-Grundlagen besprechen
- ROS-Hilfswerkzeuge einsetzen

1.1 Installation

Die Arbeit während der Entstehung unseres Roboters besteht meist aus 3D-Simulation, 3D-Konstruktion, Tests und das Ganze wieder von vorne. 3D verlangt viel Rechenleistung, doch heutzutage ist selbst ein Intel NUC mit einem i5-Prozessor im Stand, Gazebo-Simulationen nebst RViz für die Navigation mit erträglichen Leistungsmerkmalen darzustellen. Das bedeutet, wir könnten unseren Entwicklungsrechner auch im Roboter verwenden. Wer den Ein- und Ausbau nicht scheut, um den Computer an einen Monitor und eine Tastatur anzuschließen, kann das gerne tun und spart gleichzeitig eine Installation.

Für die Installation gibt es eine ausführliche Anleitung auf den Wiki-Seiten von ROS (*wiki.ros.org/ROS/Installation*). Wir unterscheiden zwischen einem leistungsstarken Entwicklungsrechner, einem Intel NUC und einem Odroid-XU4 – und auch den Raspberry Pi lasse ich nicht unter den Tisch fallen.

- **Betriebssystem für Entwicklungsrechner**
  Das Betriebssystem für den Entwicklungsrechner wählen wir gemäß den Empfehlungen auf *http://wiki.ros.org/Distributions*. Dort ist für die jeweilige ROS-Version auch die entsprechende Ubuntu-Version angegeben. In diesem Buch verwenden wir ROS Kinetic Kame auf dem Betriebssystem Ubuntu Desktop 16.04 *LTS*. Nach dem Download schreiben wir das Ubuntu-Image auf einen bootbaren USB-Stick mit *Rufus* (*https://rufus.akeo.ie/*, *http://releases.ubuntu.com/16.04/*).

- **Betriebssystem für Intel NUC**
  Ubuntu Server 16.04 *LTS* (*http://releases.ubuntu.com/16.04*). Die Abkürzung *LTS* steht für *Long Term Support* und zeichnet sich durch langfristige Unterstützung mit Security-Patches und Updates von Seiten des Distributors für das entsprechende Betriebssystem aus. LTS-Versionen haben unter Ubuntu eine gerade Zahl, also Ubuntu 14, 16 usw. Nach dem Download muss das Image auf einen bootbaren USB-Stick. Mit *Rufus* (*https://rufus.akeo.ie*) erstellt man einen solchen Boot-Stick in wenigen Minuten.

- **Betriebssystem für Odroid-XU4**
  Da es sich beim Odroid um eine ARM-Architektur handelt, ist die Betriebssystemwahl auf Versionen mit ARM-Unterstützung begrenzt. Das Image für Odroid wird auf eine SD-Karte geschrieben und im Odroid startet sofort ein fertig installiertes Ubuntu. Zum Entpacken gibt es *7Zip* für Windows und in Linux kann *unxz* das Image aus dem xz-Format entpacken. Das ausgepackte Image bekommt man auf die SD-Karte mit Win32DiskImager in Windows oder mit dem Konsolenprogramm dd in Linux (*https://odroid.in/ubuntu_16.04lts/ubuntu-16.04.3-4.9-minimal-odroid-xu4-20170824.img.xz*).

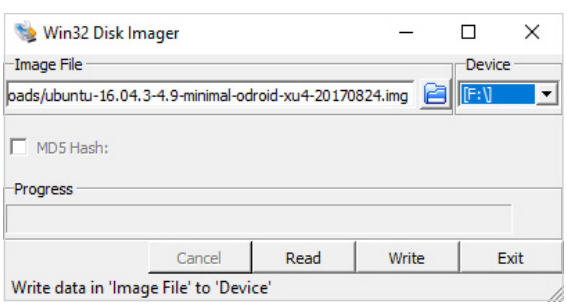
- **Betriebssystem für Raspberry Pi**
  Das Raspberry Pi hat ebenfalls eine ARM-Architektur. Das Image behandeln Sie so, wie es im Punkt für das Odroid-Board beschrieben ist. Ich habe das Server-Image von *Canonical* heruntergeladen. Auf der Downloadseite findet man Server-Images für Raspberry Pi 2 und 3. Für das aktuelle Raspberry Pi 4 steht noch kein Server-Image zur Verfügung. Das könnte sich aber bis zur Drucklegung dieses Buchs ändern. Dann wäre das Top-Modell von Raspberry Pi mit 4 GB RAM eine echte Alternative zu den genannten Systemen (*https://ubuntu.com/download/iot/raspberry-pi-2-3*).

- **ROS**
  Kinetic Kame. LTS-Variante unter den ROS-Distributionen, Unterstützung bis 2021. Von dieser Version installieren wir für AMD64, i386 oder armhf auf die gleiche Weise, nur mit unterschiedlicher Software-Ausstattung. So benötigen wir für die Programme auf dem Odroid keine grafische Benutzeroberfläche.

  Die Installationsanweisung in Schritt 1.4 (siehe Hyperlink Tabelle 1–1) unterscheidet sich für die jeweilige Hardwareplattform.

Hardware	Installation
Odroid-XU4	sudo apt install ros-kinetic-ros-base
Raspberry Pi	sudo apt install ros-kinetic-ros-base
NUC	sudo apt install ros-kinetic-ros-base
Entwicklung	sudo apt install ros-kinetic-desktop-full

Tab. 1–1 *http://wiki.ros.org/kinetic/Installation/Ubuntu*



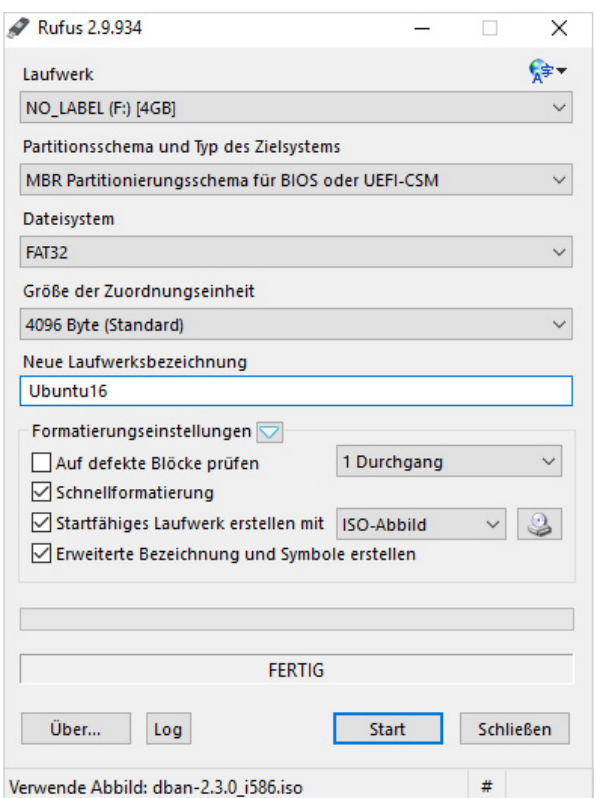


Abb. 1–1 Rufus (rechts) erstellt bootbare USB-Sticks und Win32 Disk Imager (oben) schreibt Image-Dateien auf eine SD-Karte.

Nachdem Sie Ubuntu auf Ihrem Entwicklungsrechner installiert haben, wollen wir als Nächstes ROS installieren. Die Rechner müssen für Internet konfiguriert sein und ein *ping* nach draußen sollte ohne Fehler funktionieren. Bevor wir loslegen, möchte ich auf ein paar Dinge aufmerksam machen, die Probleme bereiten können. Die Netzwerkkonfiguration ist wichtig für ROS, denn ROS basiert auf TCP/IP und TCP/IP wiederum ist das Netzwerkprotokoll. Hier ein Link mit Anleitungen, um das Netzwerk für ROS korrekt zu konfigurieren: *http://wiki.ros.org/ROS/NetworkSetup*. Das Benutzerkonto, mit dem Sie arbeiten, muss in all den Gruppen zugriffsberechtigt sein, die Sie benötigen, um Zugriff auf Schnittstellen zu bekommen. Wenn Sie mit der USB-Schnittstelle Daten an einen Mikrocontroller senden wollen, so müssen Sie in der Gruppe *dialout* sein, sonst gibt es eine Fehlermeldung.

Netzwerkkonfiguration, Uhrzeit und Gruppenzugehörigkeit prüfen!

Vorbedingung für einen fehlerfreien Ablauf ist die funktionierende Namensauflösung; außerdem sollten in */etc/hosts* die Zuordnungen IP – Name existieren und zwar für alle am Roboter beteiligten Computersysteme, wenn diese über das Netzwerk miteinander kommunizieren, sodass ein *ping hostname* ausgeführt werden kann.

Die Uhrzeit sollte synchronisiert laufen. Eine Bauanweisung an *catkin* erzeugt Fehlermeldungen, dass die Zeit in der Zukunft liegt, wenn die Uhrzeit des Computersystems auf 0 bzw. 1.1.1970 steht. Das passiert, wenn keine Knopfzellen-Batterie installiert ist, die die Uhrzeit am Laufen hält. Ein weiterer Indikator für Zeitprobleme ist, dass sich *tf*, jene Bibliothek für Transformationsberechnungen im dreidimensionalen Raum, über Diskrepanzen in den Zeitstempeln beschwert.

Diesen Benutzergruppen sollten Sie beitreten, damit keine Berechtigungsfehler auftreten, wenn USB-Schnittstellen oder Lautsprecher verwendet werden: *dialout, audio, video, plugdev, cdrom (sudo usermod <benutzer> -aG dialout,audio…)*.

1.1.1 Ubuntu-Repositorien anpassen

In Ubuntu mit grafischer Oberfläche:

1. Alt + F2 auf der Tastatur gleichzeitig drücken.
2. *software-properties-gtk* eingeben und Enter-Taste drücken.
3. Alle Haken im Reiter *Ubuntu-Anwendungen* außer bei Quelltext aktivieren.

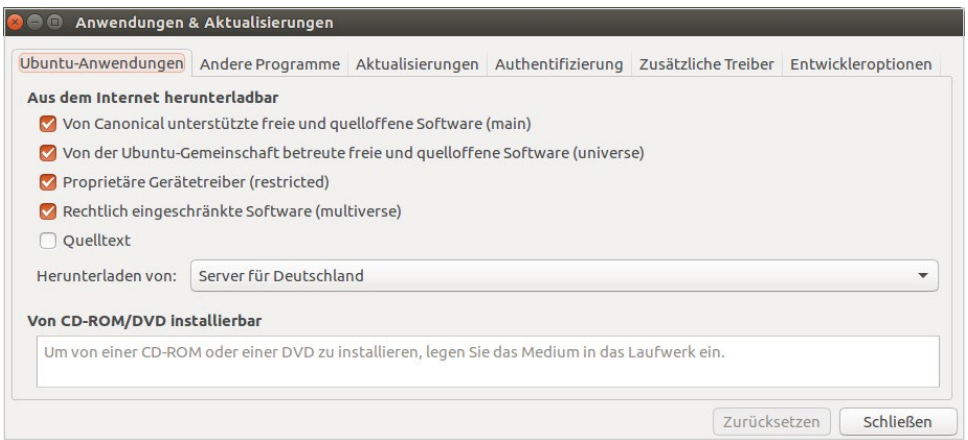


Abb. 1–2 Drittanbieter-Software akzeptieren (restricted, universe und multiverse).

In Ubuntu ohne grafische Oberfläche:

1. Öffnen Sie ein Terminal-Fenster.
2. Geben Sie Folgendes in der Konsole ein: *sudo vi /etc/apt/sources.list*
3. Überprüfen Sie, ob die Zeilen mit *deb*-Quellen aus *main*, *universe*, *restricted* und *multiverse* kein Kommentarzeichen[2] am Anfang enthalten. Standardmäßig ist bereits alles richtig eingestellt und eine Änderung in dieser Datei nicht notwendig.

1.1.2 Freiburg Mirror als Quelle angeben

```
sudo sh -c '. /etc/lsb-release && echo "deb http://packages.ros.org.ros.infor-
matik.uni-freiburg.de/ros/ubuntu $DISTRIB_CODENAME main" > /etc/apt/sources.
list.d/ros-latest.list'
```

2. Das Kommentarzeichen ist in den meisten Konfigurationsdateien eines Linux-Systems das Raute-Symbol (#). In Programmiersprachen können unterschiedliche Symbole verwendet werden.

1.1.3 Schlüssel importieren

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key
```

Wenn der obige Befehl fehlschlägt, gibt es die Möglichkeit, den Schlüssel mit folgendem Befehl manuell herunterzuladen und zu installieren.

```
wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -O - |
sudo apt-key add -
```

1.1.4 Installation

Bevor wir ROS installieren, lassen wir das System auf Aktualisierungen prüfen und ggf. installieren.

```
sudo apt update
sudo apt upgrade
```

Auf dem Entwicklungsrechner benötigen wir eine vollständige Desktopumgebung mit grafischen Werkzeugen.

```
sudo apt install ros-kinetic-desktop-full
```

Da wir auf dem Roboter keine grafischen Benutzeroberflächen benötigen, installieren wir die Basisausrüstung von ROS auf dem Intel NUC und dem Odroid XU4.

```
sudo apt install ros-kinetic-ros-base
```

1.1.5 Initialisierung mit rosdep

Im Folgenden wird für das aktuelle Benutzerkonto der versteckte Ordner .ros im Heimatverzeichnis angelegt – meist ist das /home/benutzername/.ros. Dort befinden sich später auch sämtliche log-Dateien, die von ROS während der Ausführung erstellt werden. Mit rosclean purge lassen sich alle log-Dateien im Ordner .ros löschen.

```
sudo rosdep init
rosdep update
```

Neues Benutzerkonto und ROS

Wenn ein neuer Benutzer angelegt wird, der auf dem Rechner ROS verwenden soll, muss dieser rosdep update ausführen und die ROS-Verzeichnisse, welche setup.bash-Skripte enthalten, in seine .bashrc eintragen.

1.1.6 Umgebungsvariablen setzen

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

1.1.7 rosinstall, Werkzeug für die Arbeitsbereichverwaltung

```
sudo apt install python-rosinstall python-rosinstall-generator python-wstool
build-essential
```

1.1.8 ROS-Arbeitsbereich erstellen

Als Nächstes erstellen wir einen Arbeitsbereich, der unsere eigenen Robotik-Werke enthalten wird, also unsere selbstgeschriebenen Programme. Aber auch Git-Repositorien von anderen Anbietern können wir dorthin herunterladen und verwenden. Obligatorisch ist der Name des Quelltextordners »*src*«, aber nicht der übergeordnete Ordner, dessen Namen ich gerne anders wähle, als er in wiki.ros.org mit »*catkin_ws*« vorgegeben wird. Der Grund dafür ist, dass ich oft zwischen Roboter-Computer und Entwicklungs-Computer hin und her kopiere. Wenn aber die Ordnernamen auf beiden Computern identisch sind, kann es passieren, dass man Quelle und Ziel verwechselt, was ärgerlich sein kann, wenn man vorher kein Backup gemacht hat.

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/
catkin_make
```

Der Befehl *catkin_make* erzeugt eine Datei *CMakeLists.txt*, wenn diese noch nicht existiert, und darüber hinaus die Ordner *build* und *devel*. Im *devel*-Verzeichnis befinden sich die *setup*-Dateien, die dem ROS-System bekanntgeben, wo es nach ROS-Paketen suchen soll.

Mit *catkin_make install*, dem Äquivalent zu *make install*, wird zusätzlich der Ordner *install* mit ausführbaren Binärdateien angelegt. Ansonsten macht *catkin_make* das, was *make* auch macht – es baut alle Programme, die im Arbeitsbereich vorliegen. Dieser Vorgang kann, abhängig von der Anzahl der ROS-Pakete im aktuellen Arbeitsbereich, sehr lange dauern.

Wer in unterschiedlichen Roboterprojekten arbeitet, hat die Möglichkeit, mehrere getrennte Arbeitsbereiche anzulegen. Das reduziert die Arbeit von *catkin_make* auf das, was der aktuelle Arbeitsbereich an ROS-Paketen enthält. Sie erstellen dazu wie oben beschrieben ein neues Verzeichnis mit einem Unterverzeichnis *src*. Im neuen Verzeichnis wird der Befehl *catkin_make* weitere benötigte Dateien und Verzeichnisse generieren, sodass Sie nur noch die *setup*-Datei im Verzeichnis *devel* ausführen

müssen, um den neuen Arbeitsbereich zu verwenden. Das Ausführen der *setup*-Datei eines Arbeitsbereichs blendet alle anderen Arbeitsbereiche für die Kompilierung auf dem System aus. Die anderen Arbeitsbereiche sind mit dem Befehl *roscd* dennoch erreichbar. Die Arbeit mit mehreren Arbeitsbereichen wird in ROS als Überlagerung (engl. *overlay*) bezeichnet.

build, devel und install werden von src generiert

Es kann vorkommen, dass die Ordner build, devel oder install inkonsistent werden. Diese Ordner außer *src* sollten Sie löschen und mit *catkin_make* wird aus dem Ordner *src* ein neuer *build* und *devel* generiert.

Abschließend tragen wir unseren Arbeitsbereich in unsere Terminalkonfiguration ein. Dadurch wird nach jedem Neustart der gewünschte Arbeitsbereich in jeder Konsole mit dem Befehl *roscd* erreichbar sein. Falls Sie den Ordnernamen von *catkin_ws* verändert haben, muss der entsprechende Name verwendet werden.

```
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Ob unsere Installation erfolgreich war, testen wir mit folgendem Befehl und beenden ihn anschließend mit Strg + C.

```
roscore
```

```
murat.calis@rosbox:~/catkin_ws$ roscore
... logging to /home/murat.calis/.ros/log/fc012286-9409-11e7-b6ac-080027408a1c/roslaunch-rosbox-32651.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://rosbox:42481/
ros_comm version 1.12.7


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.7

NODES

auto-starting new master
process[master]: started with pid [32662]
ROS_MASTER_URI=http://rosbox:11311/

setting /run_id to fc012286-9409-11e7-b6ac-080027408a1c
process[rosout-1]: started with pid [32675]
started core service [/rosout]
```

Abb. 1–3 roscore startet den ROS-Master auf port 11311 und erstellt die zugehörige log-Datei in ~/.ros.

Wer mehr über den Entstehungs- und Beendigungsprozess erfahren möchte, startet ein separates Fenster mit *netstat -tulpanc*. So kann man die einzelnen Prozesse beim Öffnen und Schließen der Ports beobachten.

1.1.9 Roboter Modell A installieren

Dieses Buch befasst sich mit zwei unterschiedlichen Robotern, Modell A und B. Der größte Unterschied zwischen beiden ist, dass Modell B zusätzlich eine motorisierte Kamera hat. Wir werden Modell A als Nächstes in unseren Arbeitsbereich herunterladen. Anschließend lernen wir ROS anhand dieses Roboter-Modells kennen.

- Wir installieren die Roboter-Programme im *src*-Verzeichnis des *catkin*-workspace.

```
cd /home/<benutzername>/catkin_ws/src
```

- Mit *git* laden wir die Dateien für das Roboter-Modell A in das *src*-Verzeichnis.

```
git clone https://bitbucket.org/piraterobotics/abot-kinetic.git
```

- Die Software-Abhängigkeiten, die im jeweiligen package.xml der Pakete aufgeführt sind, installieren wir mit *rosdep*. Der Schalter *-y* am Ende des Befehls bewirkt, dass alle Abfragen, die eine Benutzereingabe benötigen, automatisch mit *»yes«* bestätigt werden. Danach startet die Softwareverwaltung von Ubuntu mit dem Download und der Installation. Wechseln Sie vorher in das übergeordnete Verzeichnis, also *catkin_ws*, sonst bringt der nachfolgende Befehl eine Fehlermeldung.

```
rosdep install --from-paths src --ignore-src --rosdistro=kinetic –y
```

- Zuletzt führen wir das robotereigene Installationsskript aus. Es befindet sich im scripts-Verzeichnis des bringup-Pakets unseres Roboters. Mit roscd gelangen wir am schnellsten dahin. Sollte wider Erwarten der Befehl roscd das Paket abot_bringup nicht kennen, dann verwenden Sie den Linux-Befehl cd, um in das entsprechende Verzeichnis zu gelangen. Spätestens nach einem Neustart sollte roscd jedes ROS-Paket kennen, das wir installiert haben, sonst stimmt etwas nicht mit den Umgebungsvariablen, die wir in der .bashrc konfiguriert haben. Dort sollten am Ende der Datei die *source*-Befehle stehen, welche unsere Umgebungsvariablen konfigurieren. Sind wir im Skripte-Ordner angekommen, führen wir ein Python-Programm namens *install.py* aus.

  Das Installationsprogramm habe ich für die Einrichtung der Hardware sowie andere immer wiederkehrende Konfigurationsschritte erstellt. Ein Problem, das dieses Skript unter anderem löst, ist, dass es für die Mikrocontroller und Laserscanner jeweils einen symbolischen Link im Verzeichnis */udev* generiert. In den *launch*-Dateien verwenden wir dann nicht mehr die Gerätenamen, die vom Betriebs-

system generiert werden, sondern die symbolischen Namen, die immer auf die aktuell generierten Namen des Betriebssystems verweisen. So kann ein Gerät aus der USB-Buchse herausgezogen und wieder eingesteckt werden, ohne dass wir uns Sorgen machen müssen, dass der Gerätename vom Betriebssystem nun anders lautet, da wir ja mit einem symbolischen Link darauf zugreifen.

```
roscd abot_bringup/scripts
./install.py
```

Das Installationsprogramm benötigt Administratorrechte und wird nach dessen Ausführung ein Kennwort verlangen. Die symbolischen Links werden im Verzeichnis */udev* nur dann angezeigt, wenn Sie die Mikrocontroller oder einen Laserscanner mindestens einmal angeschlossen hatten.

1.2 ROS-Grundlagen

Zuerst prüfen wir die offiziellen Quellen zu ROS, um einen Überblick über das Framework zu gewinnen und um nicht bereits Geschriebenes in diesem Buch zu wiederholen. Empfehlenswert sind also folgende Quellen:

1. ROS-Konzepte
   http://wiki.ros.org/ROS/Concepts
2. ROS-Starthilfe
   http://wiki.ros.org/ROS/StartGuide
3. ROS-Tutorien
   http://wiki.ros.org/ROS/Tutorials
4. ROS-Spickzettel
   https://github.com/ros/cheatsheet/releases/

1.2.1 ROS-Dateisystem

Nachdem die Installation abgeschlossen ist, überprüfen wir die Funktionalität und Integrität unseres ROS-Dateisystems. Der Befehl *roscd* wechselt in das Verzeichnis eines ROS-Pakets und mit *rosls* listen wir den Verzeichnisinhalt eines ROS-Pakets auf.

```
roscd abot_navigation
rosls rospy_tutorial
```

Erscheint keine Fehlermeldung nach Ausführung beider Befehle, dann ist das ROS-Dateisystem korrekt konfiguriert. Insbesondere unsere eigenen Pakete sollten mit *roscd* erreichbar sein.

Sind die eigenen Pakete mit *roscd* nicht erreichbar, kann das mehrere Ursachen haben. ROS-Befehle suchen zuerst nach ROS-Variablen. Für *roscd* oder *rosls* ist die Variable *$CMAKE_PREFIX_PATH* relevant. Auf der Kommandozeile können wir herausfinden, was in *$CMAKE_PREFIX_PATH* enthalten ist.

```
echo $CMAKE_PREFIX_PATH
```

Ausgabe:

```
/home/<benutzername>/catkin_ws/src/opt/ros/kinetic/share
```

Nach einer frischen ROS-Installation steht meist der eigene ROS-Arbeitsbereich am Anfang, sofern einer erstellt wurde. Von einem Doppelpunkt getrennt folgt das Verzeichnis der installierten ROS-Pakete der jeweiligen ROS-Distribution.

Wenn die Ausgabe leer ist, dann wurden die *setup.bash*-Dateien nicht ausgeführt. Mit folgenden Befehlen kann man das nachholen, wobei die Reihenfolge eine Rolle spielt.

```
source /opt/ros/kinetic/setup.bash
source /home/<benutzername>/catkin_ws/devel/setup.bash
```

Normalerweise sollten beide Befehle in unserer *.bashrc* ganz unten aufgeführt sein. Wenn nicht, müssen diese dort hineinkopiert werden, damit jedes Terminal-Fenster diese setup.bash-Dateien ausführt, denn diese Befehle konfigurieren die nötigen Umgebungsvariablen für ROS.

Neben *$CMAKE_PREFIX_PATH* existiert eine gleichbedeutende Variable: *$ROS_PACKAGE_PATH*. Sie ist aus den Zeiten von *rosbuild*, also noch vor ROS Groovy und wird aus Kompatibilitätsgründen weiterhin gepflegt. Beide können in unserer *.bashrc* manuell angepasst werden. Das folgende Beispiel soll die Möglichkeiten erläutern.

```
export CMAKE_PREFIX_PATH=$CMAKE_PREFIX_PATH:/home/username/catkin_ws/src/…
```

Bei selbstkompilierten Paketen oder bei einem vollständig von Quellen gebautem ROS kann es notwendig sein, die *$CMAKE_PREFIX_PATH* bzw. *$ROS_PACKAGE_PATH* manuell anzupassen. Die folgenden Links dienen als Ergänzung zu den hier besprochenen Themen.

- *http://wiki.ros.org/catkin/conceptual_overview*
- *http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem*
- *http://wiki.ros.org/ROS/EnvironmentVariables*

1.2.2 ROS-Paket

Ein einzelnes ROS-Paket stellt die kleinstmögliche Organisationseinheit innerhalb des ROS-Frameworks dar. Ein Paket ist vergleichbar mit Linux-Software-Paketen. In einem Paket können Nodes, Programme, Bibliotheken, Konfigurationsdateien und mehr enthalten sein. Das Ziel ist eine ausreichende und nützliche, nicht aber überbordende Funktionalität, welche unübersichtlich oder schwer zugänglich wird. Wir können ROS-Pakete auch als Software-Module betrachten, da wir mit ROS modulare Software entwickeln. Ein neues Paket erstellen wir mit *catkin_create_pkg* standardmäßig im Ordner *src* unseres ROS-Arbeitsbereichs.

Die folgenden Ordner- und Dateinamen können in einem ROS-Paket vorkommen. Die Tabelle dient auch als Prüfliste bei der Arbeit mit Paketen. Ein häufiger Fehler ist, dass man die *CMakeLists.txt* nicht konfiguriert, während man *Services*, *Messages* etc. in den entsprechenden Ordnern bereits definiert hat. Die farbig gekennzeichnete Datei ist eine Pflichtdatei, die jedes ROS-Paket vorweisen muss, denn ROS-Programme suchen zuerst nach *package.xml*, um Abhängigkeiten aufzulösen oder zur Laufzeit benötigte Programme zu starten.

Ordner-/Dateiname	Beschreibung
CATKIN_IGNORE	Optionale leere Datei. Verhindert, dass dieses Paket von *catkin* kompiliert bzw. verarbeitet wird
CMakeLists.txt	Bauanleitung für *CMake*
package.xml	Software-Abhängigkeiten, Copyright, Autor, Version usw.
config	Ordner für Konfigurationsdateien (*xml*, *yaml*)
include/paket_name	Ordner für C++-Header-Dateien In *CMakeLists.txt* muss die Variable *INCLUDE_DIRS* auf den Speicherort dieser Header-Dateien verweisen

src/paket_name	Ordner für C++-Dateien und Python-Module
	In *CMakeLists.txt* muss für C++-Dateien folgende Definition angepasst werden:
	`add_executable( mein_programm src/paket_name/mein.cpp )`
	Bei Verwendung von *Python*-Modulen muss eine konfigurierte Datei *setup.py* im Paketverzeichnis erstellt werden und folgende Definition in *CMakeLists.txt* existieren:
	`catkin_python_setup()`

nodes	Python-Skripte, die eine Node-Funktionalität implementieren
scripts	Ausführbare Skript-Dateien, insbesondere Python-Skripte Implementiert ein Python-Skript eine Node-Funktionalität, dann kann es in *scripts* oder aber auch im Ordner *nodes* residieren

→

Ordner-/Dateiname	Beschreibung

srv	Service-Definitions-Dateien mit der Endung *.srv*
	Bei Verwendung von *srv*-Dateien müssen die folgenden Makros in *CMakeLists.txt* aktiviert werden:
	`add_service_files(…)`
	`generate_messages()`

msg	Message-Definitions-Dateien mit der Endung *.msg*.
	Bei Verwendung von *msg*-Dateien müssen die folgenden Makros in *CMakeLists.txt* aktiviert werden:
	`add_message_files(…)`
	`generate_messages()`

action	Ordner mit *.action*-Dateien.
	Bei Verwendung von *.action*-Dateien müssen die folgenden Makros in *CMakeLists.txt* aktiviert werden.
	`add_action_files(…)`
	`generate_messages()`

launch	Start-Dateien mit der Endung *.launch*.
urdf	Ordner mit *.urdf-*, *.xacro-* und *.gazebo*-Dateien.
meshes	Ordner mit *.dae-*, *.stl-*, *.jpeg-*, *.tiff*-Dateien.
cad	Ordner mit Konstruktionsdateien.
worlds	Ordner mit *.world*-Dateien wird von Gazebo verwendet.
models	Ordner mit *.dae-* und *.sdf*-Dateien wird von Gazebo verwendet. Der Ordner muss in der Umgebungsvariable `$GAZEBO_MODEL_PATH` enthalten sein.
buildings	Ordner mit *.sdf*-Dateien wird von Gazebo verwendet.

Tab. 1–2 Ordner- und Dateinamen in einem ROS-Paket

Der Befehl *rospack* liefert nützliche Informationen über die auf dem System installierten ROS-Pakete. Um alle installierten ROS-Pakete aufzulisten, verwendet man folgenden Befehl.

```
rospack list
```

Ist der Name des ROS-Pakets bekannt und möchte man wissen, ob das Paket auf dem System installiert ist, dann kann man sich mit dem nächsten Befehl schnell Gewissheit darüber verschaffen.

```
rospack find abot_description
```

Meist hängt ein ROS-Paket von diversen anderen Paketen ab. Die Abhängigkeiten überprüfen wir in folgendem Beispiel für das Programm *RViz*.

```
rospack depends rviz
```

1.2.3 ROS-Meta-Paket

Im Gegensatz zu einem einzelnen ROS-Paket ist ein ROS-Meta-Paket eine Ansammlung mehrerer loser ROS-Pakete in einem großen Gesamtpaket. Angenommen, wir entwickeln einen Roboter, der fahren, sehen und sprechen kann. Wir würden ein Paket für das Fahren, eines für das Sehen und zuletzt ein Paket für das Sprechen erstellen. Nun befinden sich diese Pakete innerhalb unseres *src*-Ordners und sobald weitere Pakete dazukommen, leidet die Übersichtlichkeit. Ein Meta-Paket bedeutet zugleich, dass ROS-Pakete in einem Unterordner von *src* gesammelt werden. Die Struktur unseres Beispielroboters könnte folgendermaßen aussehen, wenn wir es als ROS-Meta-Paket anlegen.

```
/home/<benutzername>/catkin_ws/src/meinbot-kinetic-master
/home/<benutzername>/catkin_ws/src/meinbot-kinetic-master/meinbot
/home/<benutzername>/catkin_ws/src/meinbot-kinetic-master/meinbot_fahren
/home/<benutzername>/catkin_ws/src/meinbot-kinetic-master/meinbot_sehen
/home/<benutzername>/catkin_ws/src/meinbot-kinetic-master/meinbot_sprechen
```

Die Anatomie eines ROS-Pakets kennen wir bereits. Interessant ist hier nur das ROS-Paket *meinbot*. Es vermittelt von seinem Namen her keine Funktion, die wir besprochen haben. Schauen wir also in den Ordner.

```
meinbot
 |__ CMakeLists.txt
 |__ package.xml
```

Das Verzeichnis von *meinbot*, welches ein Meta-Paket realisiert, enthält lediglich die Pflichtdatei *package.xml* und die Bauanleitung *CMakeLists.txt*. Der Inhalt von *CMakeLists.txt* verrät uns mehr über seine Funktion.

```
cmake_minimum_required(VERSION 2.8.3)
project(meinbot)
find_package(catkin REQUIRED)
catkin_metapackage()
```

Das Makro *catkin_metapackage()* weist *CMake* an, das Verzeichnis *meinbot* wie ein ROS-Meta-Paket zu behandeln. Werfen wir zuletzt noch einen Blick in die stark verkürzte Datei *package.xml*.

```
...
<buildtool_depend>catkin</buildtool_depend>
<run_depend>meinbot_fahren</run_depend>
<run_depend>meinbot_sehen</run_depend>
<run_depend>meinbot_sprechen</run_depend>

<export>
<metapackage />
</export>
```

Abgesehen von dem benötigten <*buildtool_depend*>-Element dürfen Meta-Pakete nur <*run_depend*>-Elemente haben. In diesen Elementen werden die losen ROS-Pakete zu einem großen, ganzen Meta-Paket geschnürt. Am Ende steht ein <*export*>-Element, das dieses ROS-Paket als ein ROS-Meta-Paket auszeichnet.

1.2.4 ROS-Master

Im Mittelpunkt steht der Master, welchen wir zuvor schon mal mit *roscore* gestartet hatten. Standardmäßig läuft dieser auf TCP-Port 11311 und wartet auf XMLRPC-Nachrichten von anderen ROS-Knoten, den *Nodes*. Die genaue Adresse der Ressource erfährt man mit:

```
echo $ROS_MASTER_URI
```

ROS-Variablen beginnen mit der Zeichenfolge ROS und können in der .bashrc überschrieben oder gesetzt werden. Ein ROS-Master auf einem entfernten Rechner lässt sich so einfach einrichten wie das Setzen der Variable $ROS_MASTER_URI. In der .bashrc könnte zum Beispiel folgende Export-Variable stehen:

```
export $ROS_MASTER_URI=http://192.168.2.123:11311
```

Startet nun ein Node, wird zuerst der Wert von $ROS_MASTER_URI abgefragt. Dann wird per XML/RPC eine Verbindung mit dem Master aufgebaut, um diesem bekanntzugeben, was man publizieren oder abonnieren möchte – nach dem publish/subscribe-Prinzip. Mit einem *ping* zwischen dem Node und dem Master sollte man vorab sicherstellen, dass es keine Verbindungsprobleme gibt. Wurde zuvor kein ROS-Master mit *roscore* oder *roslaunch* gestartet, bricht der Startprozess mit der Fehlermeldung ab, dass kein ROS-Master gefunden oder gestartet wurde.

Im Grunde ist der Master ein XML/RPC-Namensdienst, der allen *Nodes* die nötigen Informationen liefert, damit diese sich untereinander mit den richtigen *Nodes* über TCP/IP verbinden, um deren Nachrichten zu erhalten. Die Kopplung loser Programme bzw. *Nodes* durch den Master, damit diese Nachrichten untereinander austauschen können, ist ein wesentliches Merkmal von ROS.

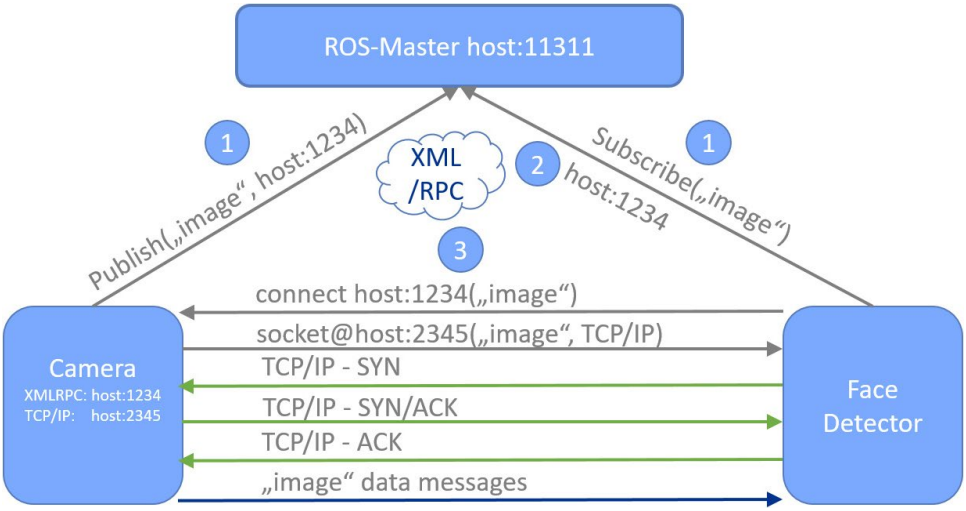


Abb. 1–4 ROS-Kommunikationskonzept (XML/RPC & TCP/IP)

In Abbildung 1–4 sehen wir zwei Nodes und einen Master. Verfolgen wir die Schritte 1 bis 3, wird deutlich, dass der Master stets eine kurzfristige Vermittlerrolle einnimmt. Der Master verkuppelt lediglich zwei Nodes miteinander und lässt diese Daten unter sich austauschen. Die Nodes sind lose verbunden, was so viel bedeutet wie, es kann ein Node jederzeit beendet werden und ein neuer Node kann jederzeit Nachrichten von anderen publizierenden Nodes abonnieren, ohne dass das Gesamtsystem beeinträchtigt wird.

Wird der Node *FaceDetector* beendet oder stürzt dieser ab, hat das keine Auswirkungen auf den Master und auch keine auf den *Camera-Node*. Startet der FaceDetector nach einer Weile wieder, wird die Verbindung wiederhergestellt und Daten werden übertragen, als wäre nichts gewesen.

Fällt der Node *Camera* aus oder wird dieser beendet, gibt es lediglich eine Warnung, aber keine Fehlermeldung und der *FaceDetector-Node* bricht deshalb auch nicht zusammen oder wird beendet, stattdessen wird gewartet, bis der Camera-Node wiederbelebt wird und Daten an das »image«-Topic ausgibt.

Dieses Merkmal macht ROS so beliebt. Es kann immer mal etwas ausfallen und trotzdem bleibt das Gesamtsystem stabil. Dezentrale Software-Module erleichtern darüber hinaus die Arbeit im Team. Jeder konzentriert sich auf seinen Teilbereich – ein Programm-Modul im Gesamtsystem.

- Mit roscore startet ein ROS-Master.
- Ein weiterer Aufruf von *roscore* führt zu einem Fehler, da es unter der verwendeten IP-Adresse nur einen Master geben kann.
- Andere ROS-Computer können sich einem ROS-Master anschließen, indem Sie die *ROS_MASTER_URI* und *ROS_MASTER_IP* setzen.

1.2.5 ROS-Nodes

Ein Node ist eine Instanz eines ausführbaren Programms. Das bedeutet, dass man ein und dasselbe Programm mehrmals gleichzeitig starten kann, indem man unterschiedliche Namen beim Aufruf verwendet.

Programme innerhalb des ROS-Systems werden nicht wie gewöhnliche Systemprogramme oder Anwendungen aufgerufen. Stattdessen wird ein Startprogramm namens rosrun verwendet, um das Programm zu einem vollwertigen Node zu machen. Wir haben zuvor gelernt, dass der ROS-Master sämtliche Ports öffnet und Sockets bereitstellt, damit ein Node erreichbar ist und mit anderen Nodes kommunizieren kann. Wir werden also nie Sockets programmieren oder Ports öffnen müssen, denn das übernimmt ROS für uns.

Im folgenden Befehl wird mit *rosrun* aus dem Paket rqt_image_view das gleichnamige Programm rqt_image_view gestartet. Im darauffolgenden Befehl ist ein Beispiel mit dem Parameter __name. Der letzte Befehl zeigt die nötigen Informationen eines Node an. Wenn man Nachrichten von einem Node erhalten oder umgekehrt Nachrichten an ein Node senden möchte, dann stehen in den Informationen die Nachrichtendefinitionen, die von den jeweiligen Themen (topics) verwendet werden.

```
rosrun rqt_image_view rqt_image_view
rosrun rviz rviz __name:=mein_test_name
rosnode info mein_test_name
```

Tabulator-Taste verwenden!

In Linux kann man die Konsole dazu bewegen, nach einer Tastatur-Eingabe alles auszugeben, was im Kontext möglich ist. Wenn man »*rosrun rqt_*« eingibt, aber nicht weiß, wie es weiter geht, einfach zweimal hintereinander die Tabulator-Taste drücken und schon kommen mögliche Eingabe-Vorschläge von der Konsole.

Sobald der mit *rosrun* gestartete *Node* läuft, kann man mit dem Befehl *rosnode list* prüfen, welchen *Node*-Namen die Programm-Instanz erhalten hat. Weitere Möglichkeiten gibt die Konsole mit *rosnode* und zwei aufeinanderfolgenden Tabulator-Eingaben aus.

Der Befehl *rosrun* sucht in allen Verzeichnissen, die in der Variable *$CMAKE_PREFIX_PATH* enthalten sind, nach ausführbaren Programmen.

Unsere Programme werden wir jedoch nicht einzeln per *rosrun* starten, sondern später mit *roslaunch* in einer Starter-Datei bündeln. Schließlich werden wir viele kleine Programme, die wir nun als *Nodes* bezeichnen, zu einem großen Cortex verknüpfen.

Sobald wir beginnen, eigene *Nodes* zu programmieren, werden diese hauptsächlich *Topics*, *Services*, *Service-Clients*, *Action-Clients*, *Action-Services* implementieren. Somit generieren wir Schnittstellen für andere *Nodes*. Dabei beschränkt die Wahl einer der soeben genannten Funktionen nicht die anderen. Wir können also *Topics*, *Services* und *Actions* gleichzeitig und mehrfach vorkommend in einem *Node* implementieren.

1.2.6 ROS-Topics

Damit ein *Node* mit anderen *Nodes* kommunizieren kann, publiziert oder abonniert dieser ein oder mehrere *Topics*. *Topics* sind Themen-Bezeichnungen, unter denen es fortwährend Nachrichten (engl. *Messages*) zu senden oder empfangen gibt. Will ein *Node* zur Gesichtserkennung Nachrichten von einer Kamera erhalten, so abonniert dieser *Node* für die Gesichtserkennung ein entsprechendes *Topic* – zum Beispiel »/camera/image_raw«. Nachdem ein *Topic* abonniert wurde, fließen Nachrichten vom publizierenden zum abonnierenden *Node*, also in nur eine Richtung.

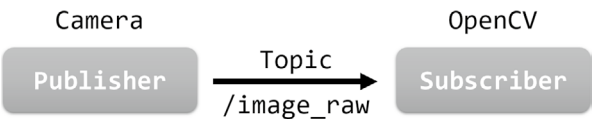


Abb. 1–5 Ein Publisher, ein Topic und ein Subscriber

Die in der obigen Abbildung dargestellte 1:1-Kommunikationsform kann auch in Form einer 1:n- oder n:n-Kommunikation vorkommen. Wenn ein *Topic* für mehr als nur einen Abonnenten interessant ist, lesen mehrere *Subscriber* denselben Nachrichtenkanal. Darüber hinaus kann ein *Subscriber* beliebig viele *Topics* abonnieren, auch wenn diese von unterschiedlichen *Publishern* stammen.

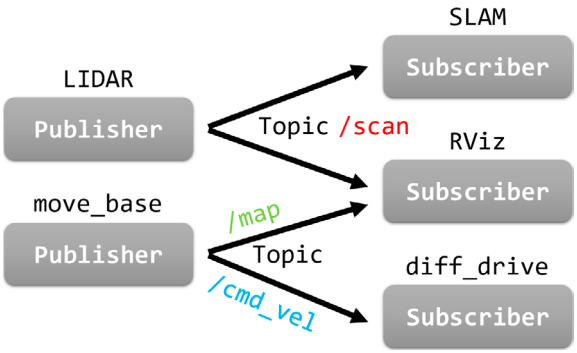


Abb. 1–6 Viele Publisher, viele Topics und viele Subscriber

Der Befehl, der in ROS wohl am häufigsten benutzt wird, ist der zum Anzeigen der *Topics*.

```
rostopic list
```

Zum Abonnieren bzw. Ausgeben der Nachrichten in der Konsole verwenden wir *echo*.

```
rostopic echo /camera/image_raw
```

Für das Publizieren muss nicht gleich ein Node programmiert werden, denn der pub-Befehl erfüllt diese Funktion bereits in der Konsole. In nachfolgendem Befehl geht es um Nachrichten, die mit einer Rate *-r* in Hertz, also zehnmal in der Sekunde an das *Topic* mit der Bezeichnung */cmd_vel* gesendet werden. Die publizierten Nachrichten enthalten dann Daten im Format geometry_msgs/Twist. Mehr Information darüber erfährt man mit rosmsg show geometry_msgs/Twist. Die beiden aufeinanderfolgenden Minussymbole ermöglichen negative Werteangaben. Ein Tupel beginnt immer mit einem einfachen hochgestellten Komma und muss damit auch beendet werden. Die Einheit für Fortbewegungen in einem Drei-Achsen-System wird von ROS in Meter pro Sekunde vorgegeben. Also bewirkt der folgende Befehl, dass sich das Fahrzeug mit 0,2 Meter/Sekunde auf der x-Achse rückwärts bewegt und mit 1,57 Radiant/Sekunde gegen den Uhrzeigersinn dreht.

```
rostopic pub -r 10 /cmd_vel geometry_msgs/Twist -- '{linear: {x: -0.2, y:
0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.57}}'
```

Nachrichten in der Konsole publizieren

Oft ist es nicht trivial, die grammatikalisch korrekte Schreibweise zum Publizieren von Nachrichten in der Konsole zu erahnen. Da wir bereits zuvor schon von der `Tabulator`-Taste Hilfe erhalten haben, wird diese uns einmal mehr behilflich sein. Man gebe den Befehl bis zur Bezeichnung des Topics ein (rostopic pub /cmd_vel) und verwende dann die `Tabulator`-Taste und schon wird der gesamte Befehl samt Vorgaben ausgedruckt. Nur noch die Werte einsetzen und fertig.

1.2.7 ROS-Messages

Zuvor haben wir von Nachrichten (engl. *Messages*) im Zusammenhang mit *Nodes* gesprochen. Sie sind die Datenströme, die zwischen zwei oder mehr *Nodes* transportiert werden. *Messages* werden in *.msg*-Dateien definiert. Dort steht der Datentyp mit Bezeichner. ROS unterstützt einfache Datentypen und mehrdimensionale Arrays, aus denen man komplexe Datenstrukturen konstruieren kann. Der Ordnername für Messagedateien lautet *msg*.

Messages fließen immer nur in eine Richtung. Um eine bidirektionale Kommunikation zwischen zwei *Nodes* zu etablieren, benötigt man ein *Service* oder eine *Action*.

Die Kompatibilität der ausgetauschten Nachrichten gewährleisten *Subscriber* durch Überprüfen der MD5-Quersumme, die aus der Nachrichtendefinition berechnet wird. Jeder *Subscriber* prüft also zuerst, ob die MD5-Quersumme korrekt ist, um dann die empfangenen Nachrichten zu verarbeiten.

Fehlerhafte Nachrichtenübertragung

Eine Fehlermeldung in der MD5-Quersumme kann auf eine inkompatible Nachrichtendefinition aus einer älteren ROS-Version hindeuten.

Detaillierte Auskunft über Nachrichtendefinitionen erhalten wir mit rosmsg. Schauen wir uns die weiter oben verwendete Nachrichtendefinition geometry_msgs/Twist etwas genauer an. Der Name ist aufgeteilt in Paket (geometry_msgs) und Format (Twist).

Die Einteilung in Namensräume dient hauptsächlich der Kollisionsvermeidung mit anderen identischen Formatnamen. Es kann also geometry_msgs/Twist und turtlebot/Twist geben, die eine vermeintlich ähnliche Datenstruktur verkörpern.

Der folgende Befehl gibt eine solche Datenstruktur auf der Konsole aus. Wenn man nicht weiß, wie die Datentypbezeichnung lautet, kann man eine Linux-Pipe (das