

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/314101187>

Programming for Robotics - Introduction to ROS

Presentation · February 2017

DOI: 10.13140/RG.2.2.14140.44161

CITATIONS

0

READS

12,813

4 authors, including:



Péter Fankhauser

ANYbotics

45 PUBLICATIONS 924 CITATIONS

[SEE PROFILE](#)



Martin Wermelinger

ETH Zurich

12 PUBLICATIONS 88 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:



ANYmal Research [View project](#)



Hybrid Locomotion: Exploiting the Advantages of Wheeled-Legged Robots on Varying Terrain [View project](#)



Programming for Robotics

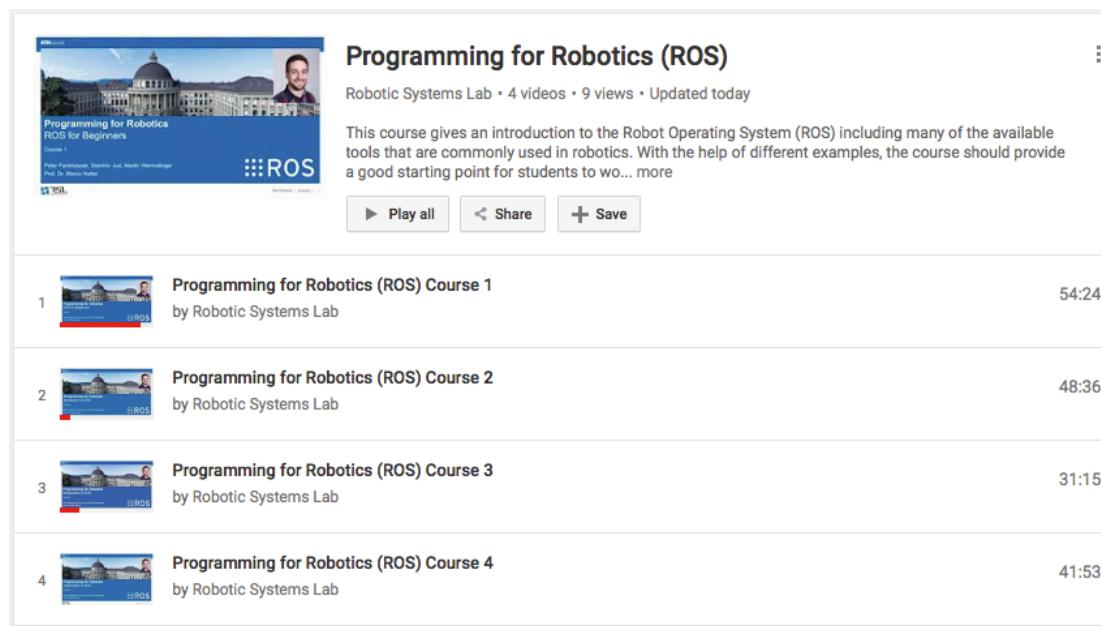
Introduction to ROS

Péter Fankhauser, Dominic Jud, Martin Wermelinger, Marco Hutter



Presentation videos

<https://www.youtube.com/playlist?list=PLE-BQwvVGf8HOvwXPgtDfWoxd4Cc6ghiP>



Programming for Robotics (ROS)

Robotic Systems Lab • 4 videos • 9 views • Updated today

This course gives an introduction to the Robot Operating System (ROS) including many of the available tools that are commonly used in robotics. With the help of different examples, the course should provide a good starting point for students to wo... more

Play all Share Save

- 1 Programming for Robotics (ROS) Course 1 by Robotic Systems Lab 54:24
- 2 Programming for Robotics (ROS) Course 2 by Robotic Systems Lab 48:36
- 3 Programming for Robotics (ROS) Course 3 by Robotic Systems Lab 31:15
- 4 Programming for Robotics (ROS) Course 4 by Robotic Systems Lab 41:53

Course Material & Exercises

<http://www.rsl.ethz.ch/education-students/lectures/ros.html>

Course material

Topics	Material
20.2.	<ul style="list-style-type: none"> – ROS architecture & philosophy – ROS master, nodes, and topics – Console commands – Catkin workspace and build system – Launch-files – Gazebo simulator
23.2.	<ul style="list-style-type: none"> – ROS package structure – Integration and programming with Eclipse – ROS C++ client library (roscpp) – ROS subscribers and publishers – ROS parameter server – RViz visualization
24.2.	<ul style="list-style-type: none"> – TF Transformation System – rqt User Interface – Robot models (URDF) – Simulation descriptions (SDF)
27.2.	<ul style="list-style-type: none"> – ROS services – ROS actions (actionlib) – ROS time – ROS bags – Debugging strategies
2.3.	<ul style="list-style-type: none"> – Lecture 1 (PDF, 3.2 MB)  Updated 22.02.2017 – Exercise 1 (PDF, 290 KB)  Updated 22.02.2017 – Lecture 2 (PDF, 4.1 MB)  Updated 24.02.2017 – Exercise 2 (PDF, 210 KB)  Updated 22.02.2017 – Husky Highlevel Controller Template (ZIP, 3 KB)  Updated 22.02.2017 – Example Solution Exercise 2 (ZIP, 5 KB)  Updated 24.02.2017 – Lecture 3 (PDF, 5.5 MB)  Updated 24.02.2017 – Exercise 3 (PDF, 227 KB)  Updated 23.02.2017 – ROS Worlds (ZIP, 1 KB)  Updated 13.02.2017 – Lecture 4 (PDF, 966 KB)  Updated 26.02.2017 – Exercise 4 (PDF, 384 KB)  Updated 26.02.2017 – ROS Bag (BAG, 158.9 MB)  Updated 13.02.2017 – Exercise 5 (PDF, 127 KB)  Updated 13.02.2017

Overview

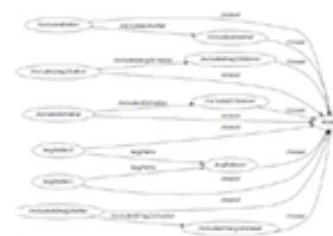
- **Part 1**
 - ROS architecture & philosophy
 - ROS master, nodes, and topics
 - Console commands
 - Catkin workspace and build system
 - Launch-files
 - Gazebo simulator
- **Part 2**
 - ROS package structure
 - Integration and programming with Eclipse
 - ROS C++ client library (roscpp)
 - ROS subscribers and publishers
 - ROS parameter server
 - RViz visualization
- **Part 3**
 - TF Transformation System
 - rqt User Interface
 - Robot models (URDF)
 - Simulation descriptions (SDF)
- **Part 4**
 - ROS services
 - ROS actions (actionlib)
 - ROS time
 - ROS bags
 - Debugging strategies
- **Part 5**
 - Case study

Overview Part 1

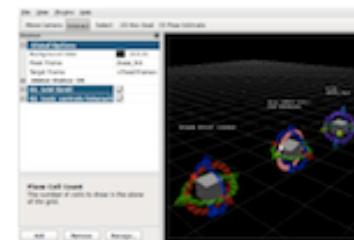
- ROS architecture & philosophy
- ROS master, nodes, and topics
- Console commands
- Catkin workspace and build system
- Launch-files
- Gazebo simulator

What is ROS?

ROS = Robot Operating System



+



+



+

ros.org

Plumbing

- Process management
- Inter-process communication
- Device drivers

Tools

- Simulation
- Visualization
- Graphical user interface
- Data logging

Capabilities

- Control
- Planning
- Perception
- Mapping
- Manipulation

Ecosystem

- Package organization
- Software distribution
- Documentation
- Tutorials

History of ROS

- Originally developed in 2007 at the Stanford Artificial Intelligence Laboratory
- Since 2013 managed by OSRF
- Today used by many robots, universities and companies
- De facto standard for robot programming



ros.org

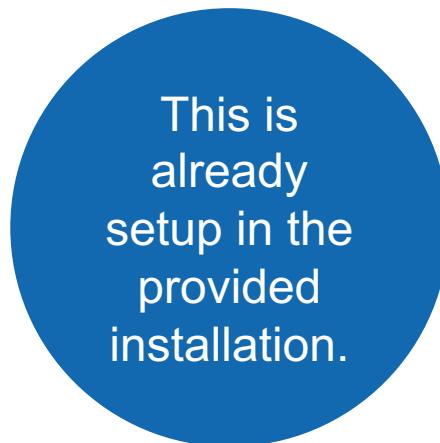
ROS Philosophy

- **Peer to peer**
Individual programs communicate over defined API (ROS *messages, services, etc.*).
- **Distributed**
Programs can be run on multiple computers and communicate over the network.
- **Multi-lingual**
ROS modules can be written in any language for which a client library exists (C++, Python, MATLAB, Java, etc.).
- **Light-weight**
Stand-alone libraries are wrapped around with a thin ROS layer.
- **Free and open-source**
Most ROS software is open-source and free to use.

ROS Workspace Environment

- Defines context for the current workspace
- Default workspace loaded with

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



Overlay your *catkin* workspace with

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

See setup with

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

Check your workspace with

```
> echo $ROS_PACKAGE_PATH
```

More info

<http://wiki.ros.org/indigo/Installation/Ubuntu>
<http://wiki.ros.org/catkin/workspaces>

ROS Master

- Manages the communication between nodes
- Every node registers at startup with the master

ROS Master

Start a master with

```
> roscore
```

More info

<http://wiki.ros.org/Master>

ROS Nodes

- Single-purpose, executable program
- Individually compiled, executed, and managed
- Organized in *packages*

Run a node with

```
> rosrun package_name node_name
```

See active nodes with

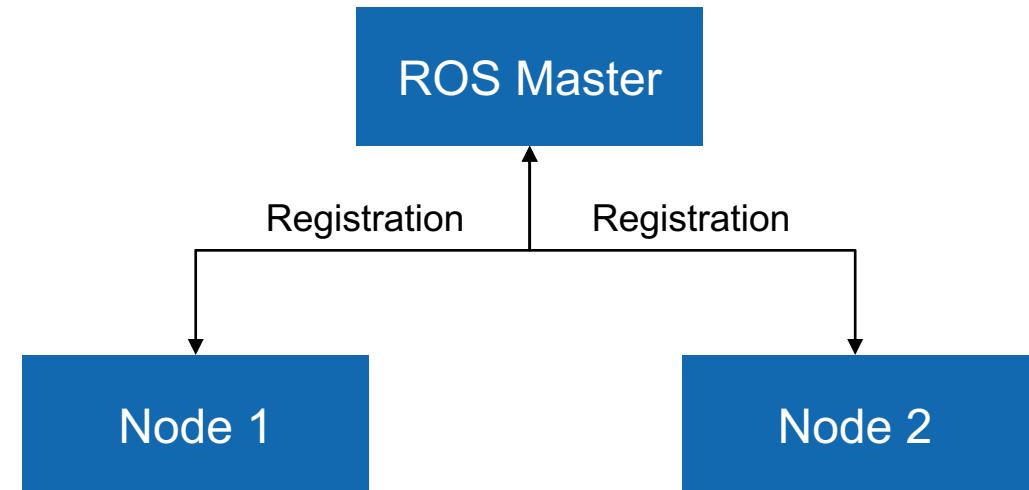
```
> rosnodes list
```

Retrieve information about a node with

```
> rosnodes info node_name
```

More info

<http://wiki.ros.org/rosnodes>



ROS Topics

- Nodes communicate over *topics*
 - Nodes can *publish* or *subscribe* to a topic
 - Typically, 1 publisher and n subscribers
- Topic is a name for a stream of *messages*

List active topics with

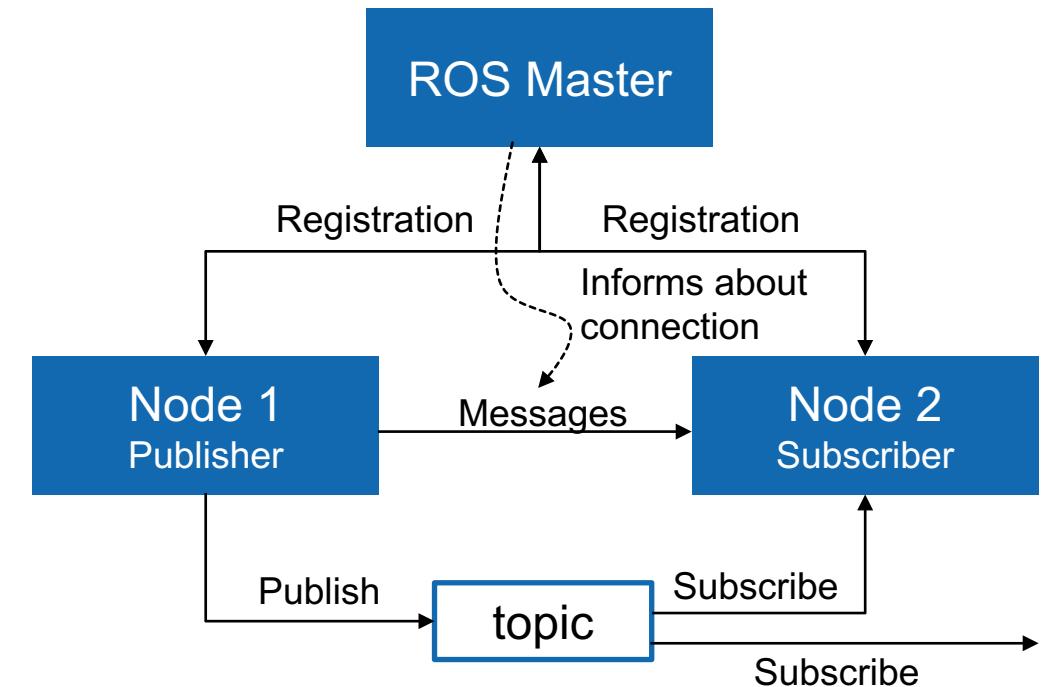
```
> rostopic list
```

Subscribe and print the contents of a topic with

```
> rostopic echo /topic
```

Show information about a topic with

```
> rostopic info /topic
```



More info
<http://wiki.ros.org/rostopic>

ROS Messages

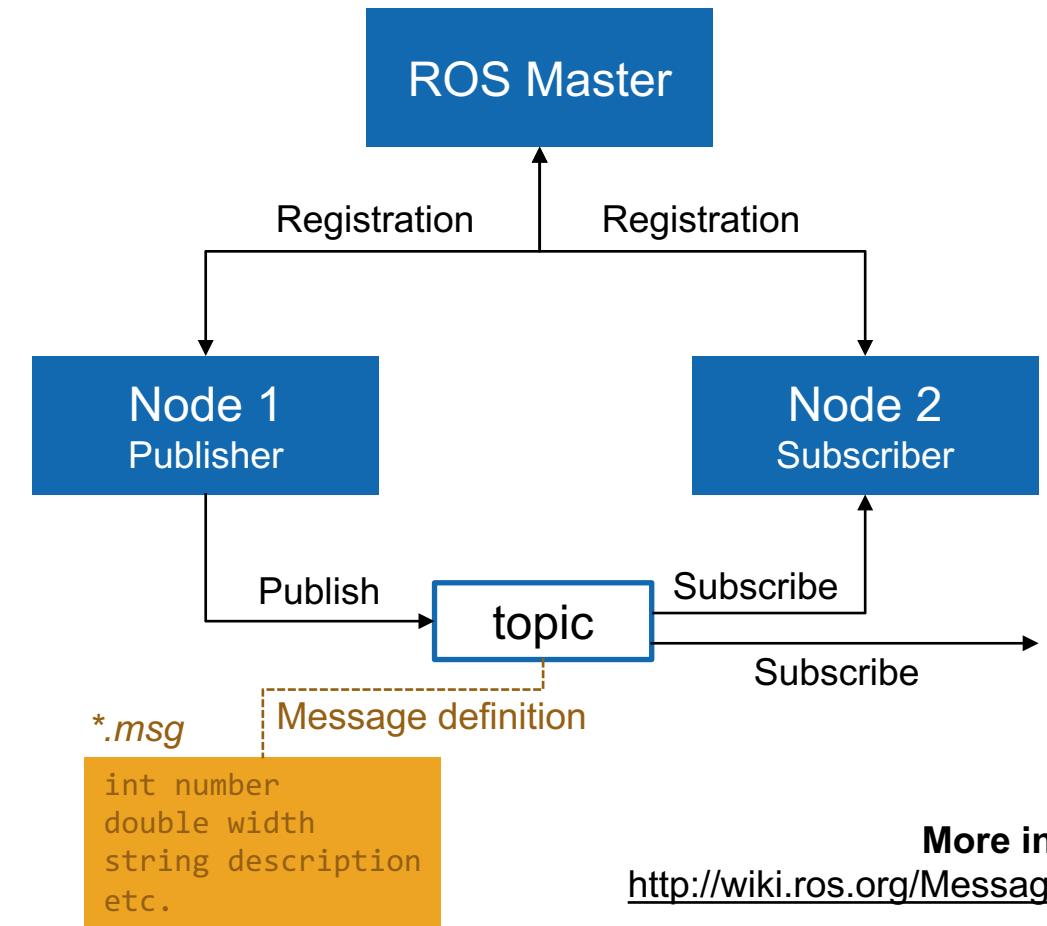
- Data structure defining the *type* of a topic
- Comprised of a nested structure of integers, floats, booleans, strings etc. and arrays of objects
- Defined in **.msg* files

See the type of a topic

```
> rostopic type /topic
```

Publish a message to a topic

```
> rostopic pub /topic type args
```



More info
<http://wiki.ros.org/Messages>

ROS Messages

Pose Stamped Example

geometry_msgs/Point.msg

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

sensor_msgs/Image.msg

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

geometry_msgs/PoseStamped.msg

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

Example

Console Tab Nr. 1 – Starting a *roscore*

Start a *roscore* with

```
> roscore
```

```
student@ubuntu:~/catkin_ws$ roscore
... logging to /home/student/.ros/log/6c1852aa-e961-11e6-8543-000c297bd368/roslaunch-ubuntu-6696.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://ubuntu:34089/
ros_comm version 1.11.20

SUMMARY
=====

PARAMETERS
* /rosdistro: indigo
* /rosversion: 1.11.20

NODES

auto-starting new master
process[master]: started with pid [6708]
ROS_MASTER_URI=http://ubuntu:11311/

setting /run_id to 6c1852aa-e961-11e6-8543-000c297bd368
process[rosout-1]: started with pid [6721]
started core service [/rosout]
```

Example

Console Tab Nr. 2 – Starting a *talker* node

Run a talker demo node with

```
> rosrun roscpp_tutorials talker
```

```
student@ubuntu:~/catkin_ws$ rosrun roscpp_tutorials talker
[ INFO] [1486051708.424661519]: hello world 0
[ INFO] [1486051708.525227845]: hello world 1
[ INFO] [1486051708.624747612]: hello world 2
[ INFO] [1486051708.724826782]: hello world 3
[ INFO] [1486051708.825928577]: hello world 4
[ INFO] [1486051708.925379775]: hello world 5
[ INFO] [1486051709.024971132]: hello world 6
[ INFO] [1486051709.125450960]: hello world 7
[ INFO] [1486051709.225272747]: hello world 8
[ INFO] [1486051709.325389210]: hello world 9
```

Example

Console Tab Nr. 3 – Analyze *talker* node

See the list of active nodes

```
> rosnode list
```

Show information about the *talker* node

```
> rosnode info /talker
```

See information about the *chatter* topic

```
> rostopic info /chatter
```

```
student@ubuntu:~/catkin_ws$ rosnode list
/rosout
/talker
```

```
student@ubuntu:~/catkin_ws$ rosnode info /talker
```

```
-----
--  
Node [/talker]  
Publications:  
  * /chatter [std_msgs/String]  
  * /rosout [rosgraph_msgs/Log]
```

```
Subscriptions: None
```

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

```
student@ubuntu:~/catkin_ws$ rostopic info /chatter
Type: std_msgs/String
```

```
Publishers:  
  * /talker (http://ubuntu:39173/)
```

```
Subscribers: None
```

Example

Console Tab Nr. 3 – Analyze *chatter* topic

Check the type of the *chatter* topic

```
> rostopic type /chatter
```

```
student@ubuntu:~/catkin_ws$ rostopic type /chatter
std_msgs/String
```

Show the message contents of the topic

```
> rostopic echo /chatter
```

```
student@ubuntu:~/catkin_ws$ rostopic echo /chatter
data: hello world 11874
---
data: hello world 11875
---
data: hello world 11876
```

Analyze the frequency

```
> rostopic hz /chatter
```

```
student@ubuntu:~/catkin_ws$ rostopic hz /chatter
subscribed to [/chatter]
average rate: 9.991
    min: 0.099s max: 0.101s std dev: 0.00076s window: 10
average rate: 9.996
    min: 0.099s max: 0.101s std dev: 0.00069s window: 20
```

Example

Console Tab Nr. 4 – Starting a *listener* node

Run a listener demo node with

```
> rosrun roscpp_tutorials listener
```

```
student@ubuntu:~/catkin_ws$ rosrun roscpp_tutorials listener
[ INFO] [1486053802.204104598]: I heard: [hello world 19548]
[ INFO] [1486053802.304538827]: I heard: [hello world 19549]
[ INFO] [1486053802.403853395]: I heard: [hello world 19550]
[ INFO] [1486053802.504438133]: I heard: [hello world 19551]
[ INFO] [1486053802.604297608]: I heard: [hello world 19552]
```

Example

Console Tab Nr. 3 – Analyze

See the new *listener* node with

```
> rosnode list
```

Show the connection of the nodes over the chatter topic with

```
> rostopic info /chatter
```

```
student@ubuntu:~/catkin_ws$ rosnode list
/listener ←
/rosout
/talker
```

```
student@ubuntu:~/catkin_ws$ rostopic info /chatter
Type: std_msgs/String

Publishers:
* /talker (http://ubuntu:39173/) ←

Subscribers:
* /listener (http://ubuntu:34664/) ←
```

Example

Console Tab Nr. 3 – Publish Message from Console

Close the *talker* node in console nr. 2 with Ctrl + C

Publish your own message with

```
> rostopic pub /chatter std_msgs/String  
"data: 'ETH Zurich ROS Course'"
```

```
student@ubuntu:~/catkin_ws$ rostopic pub /chatter std_msgs/String "data: 'ETH  
Zurich ROS Course'"  
publishing and latching message. Press ctrl-C to terminate
```

Check the output of the *listener* in console nr. 4

```
[ INFO] [1486054667.5011872]: I heard: [hello world 28201]  
[ INFO] [1486054667.604322265]: I heard: [hello world 28202]  
[ INFO] [1486054667.704264199]: I heard: [hello world 28203]  
[ INFO] [1486054667.804389058]: I heard: [hello world 28204]  
[ INFO] [1486054707.646404558]: I heard: [ETH Zurich ROS Course]
```

catkin Build System

- *catkin* is the ROS build system to generate executables, libraries, and interfaces
- We suggest to use the *Catkin Command Line Tools*

→ Use `catkin build` instead of `catkin_make`

Navigate to your catkin workspace with

```
> cd ~/catkin_ws
```

Build a package with

```
> catkin build package_name
```

! Whenever you build a **new** package, update your environment

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

The `catkin` command line tools are pre-installed in the provided installation.

More info

<http://wiki.ros.org/catkin/Tutorials>
<https://catkin-tools.readthedocs.io/>

catkin Build System

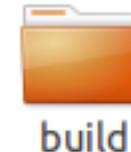
The catkin workspace contains the following spaces

Work here



The *source* space contains the source code. This is where you can clone, create, and edit source code for the packages you want to build.

Don't touch



The *build* space is where CMake is invoked to build the packages in the source space. Cache information and other intermediate files are kept here.

Don't touch



The *development (devel)* space is where built targets are placed (prior to being installed).

If necessary, clean the entire build and devel space with

```
> catkin clean
```

More info

<http://wiki.ros.org/catkin/workspaces>

catkin Build System

The catkin workspace setup can be checked with

```
> catkin config
```

For example, to set the *CMake build type* to Release (or Debug etc.), use

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

More info

http://catkin-tools.readthedocs.io/en/latest/verbs/catkin_config.html

http://catkin-tools.readthedocs.io/en/latest/cheat_sheet.html

```
student@ubuntu:~/catkin_ws$ catkin config
-----
Profile:           default
Extending:        [env] /opt/ros/indigo:/home/student/catkin_ws/devel
Workspace:        /home/student/catkin_ws

Source Space:     [exists] /home/student/catkin_ws/src
Log Space:        [exists] /home/student/catkin_ws/logs
Build Space:      [exists] /home/student/catkin_ws/build
Devel Space:      [exists] /home/student/catkin_ws/devel
Install Space:   [unused] /home/student/catkin_ws/install
DESTDIR:          [unused] None

Devel Space Layout: linked
Install Space Layout: None

Additional CMake Args: -GEclipse CDT4 - Unix Makefiles -DCMAKE_CXX_COMPILER=gcc
ILER_ARG1=-std=c++11 -DCMAKE_BUILD_TYPE=Release
Additional Make Args: None
Additional catkin Make Args: None
Internal Make Job Server: True
Cache Job Environments: False

Whitelisted Packages: None
Blacklisted Packages: None

Workspace configuration appears valid.
```

Already
setup in the
provided
installation.

Example

Open a terminal and browse to your git folder

```
> cd ~/git
```

Clone the Git repository with

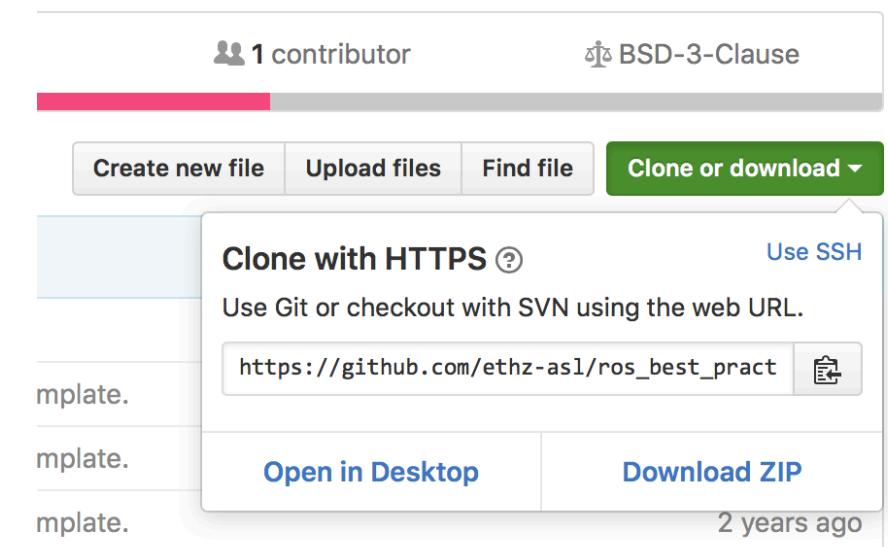
```
> git clone https://github.com/ethz-asl/ros_best_practices.git
```

Symlink the new package to your catkin workspace

```
> ln -s ~/git/ros_best_practices/ ~/catkin_ws/src/
```

Note: You could also directly clone to your catkin workspace, but using a common git folder is convenient if you have multiple catkin workspaces.

https://github.com/ethz-asl/ros_best_practices



Example

Go to your catkin workspace

```
> cd ~/catkin_ws
```

Build the package with

```
> catkin build ros_package_template
```

Re-source your workspace setup

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

Launch the node with

```
> roslaunch ros_package_template  
    ros_package_template.launch
```

```
NOTE: Forcing CMake to run for each package.  
-----  
[build] Found '1' packages in 0.0 seconds.  
[build] Updating package table.  
Starting >>> catkin_tools_prebuild  
Finished <<< catkin_tools_prebuild [ 1.0 seconds ]  
Starting >>> ros_package_template  
Finished <<< ros_package_template [ 4.1 seconds ]  
[build] Summary: All 2 packages succeeded!  
[build] Ignored: None.  
[build] Warnings: None.  
[build] Abandoned: None.  
[build] Failed: None.  
[build] Runtime: 5.2 seconds total.  
[build] Note: Workspace packages have changed, please re-source setup files to use them.  
student@ubuntu:~/catkin_ws$
```

```
  /ros_package_template/subscriber_topic: /temperature  
* /rosdistro: indigo  
* /rosversion: 1.11.20  
  
NODES  
/  
  ros_package_template (ros_package_template/ros_package_template)  
  
auto-starting new master  
process[master]: started with pid [27185]  
ROS_MASTER_URI=http://localhost:11311  
  
setting /run_id to e43f937a-ed52-11e6-9789-000c297bd368  
process[rosout-1]: started with pid [27198]  
started core service [/rosout]  
process[ros_package_template-2]: started with pid [27201]  
[ INFO] [1486485095.843512614]: Successfully launched node.
```

ROS Launch

- *launch* is a tool for launching multiple nodes (as well as setting parameters)
- Are written in XML as **.launch* files
- If not yet running, launch automatically starts a roscore

Browse to the folder and start a launch file with

```
> roslaunch file_name.launch
```

Start a launch file from a package with

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

More info

<http://wiki.ros.org/roslaunch>

Example console output for
`roslaunch roscpp_tutorials talker_listener.launch`

```
student@ubuntu:~/catkin_ws$ roslaunch roscpp_tutorials talker_listener.launch
... logging to /home/student/.ros/log/794321aa-e950-11e6-95db-000c297bd368/roslaunch
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://ubuntu:37592/

SUMMARY
========
PARAMETERS
  * /rosdistro: indigo
  * /rosversion: 1.11.20

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

auto-starting new master
process[master]: started with pid [5772]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 794321aa-e950-11e6-95db-000c297bd368
process[rosout-1]: started with pid [5785]
started core service [/rosout]
process[listener-2]: started with pid [5788]
process[talker-3]: started with pid [5795]
[ INFO] [1486044252.537801350]: hello world 0
[ INFO] [1486044252.638886504]: hello world 1
[ INFO] [1486044252.738279674]: hello world 2
[ INFO] [1486044252.838357245]: hello world 3
```

ROS Launch File Structure

talker_listener.launch

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

- **launch**: Root element of the launch file
- **node**: Each `<node>` tag specifies a node to be launched
- **name**: Name of the node (free to choose)
- **pkg**: Package containing the node
- **type**: Type of the node, there must be a corresponding executable with the same name
- **output**: Specifies where to output log messages (screen: console, log: log file)

More info

<http://wiki.ros.org/roslaunch/XML>

<http://wiki.ros.org/roslaunch/Tutorials/Roslaunch%20tips%20for%20larger%20projects>

! Notice the syntax difference
for self-closing tags:
`<tag></tag>` and `<tag/>`

ROS Launch Arguments

- Create re-usable launch files with `<arg>` tag, which works like a parameter (default optional)

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

- Use arguments in launch file with

```
$(arg arg_name)
```

- When launching, arguments can be set with

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

range_world.launch (simplified)

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

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

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

More info

<http://wiki.ros.org/roslaunch/XML/arg>

ROS Launch

Including Other Launch Files

- Include other launch files with `<include>` tag to organize large projects

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

- Find the system path to other packages with

```
$(find package_name)
```

- Pass arguments to the included file

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

range_world.launch (simplified)

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

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

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

More info

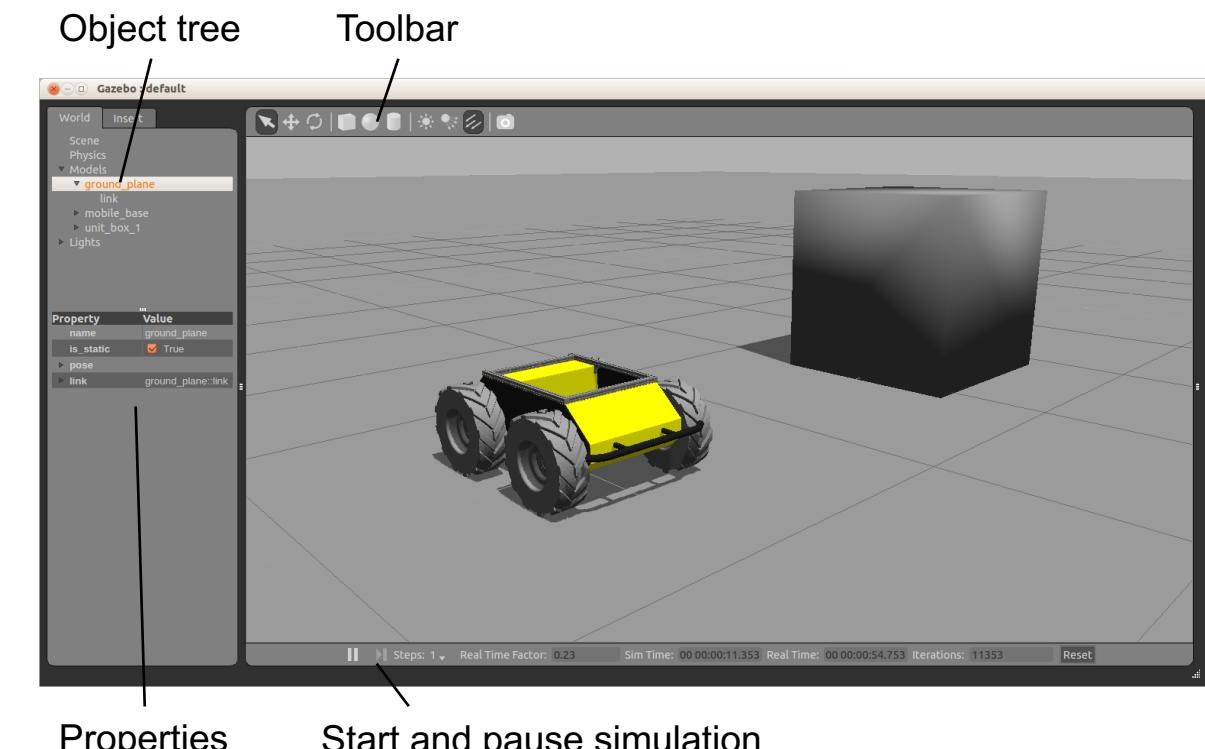
<http://wiki.ros.org/roslaunch/XML/include>

Gazebo Simulator

- Simulate 3d rigid-body dynamics
- Simulate a variety of sensors including noise
- 3d visualization and user interaction
- Includes a database of many robots and environments (*Gazebo worlds*)
- Provides a ROS interface
- Extensible with plugins

Run Gazebo with

```
> rosrun gazebo_ros gazebo
```



More info

<http://gazebosim.org/>

<http://gazebosim.org/tutorials>

Overview Part 2

- ROS package structure
- Integration and programming with Eclipse
- ROS C++ client library (roscpp)
- ROS subscribers and publishers
- ROS parameter server
- RViz visualization

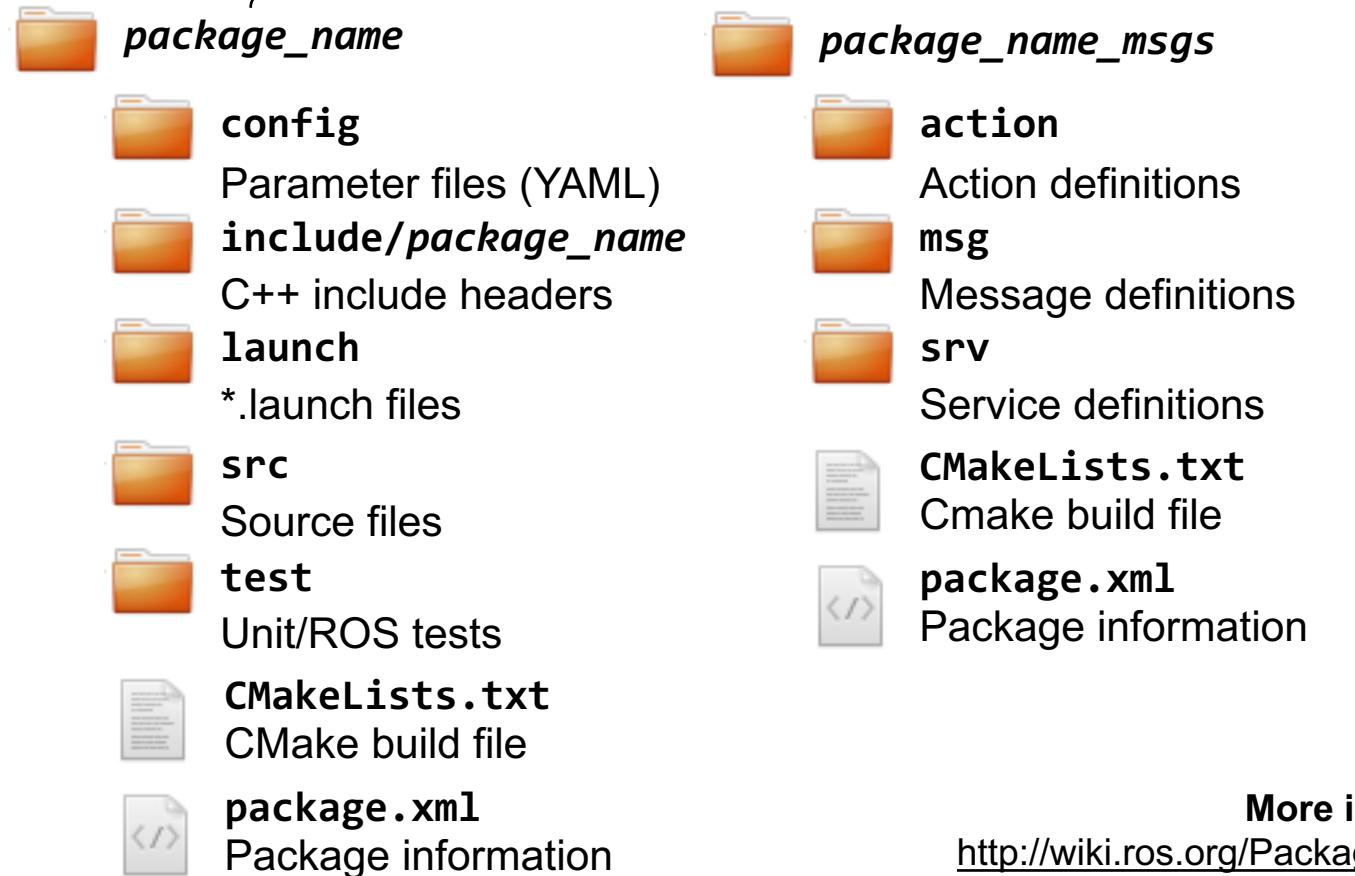
ROS Packages

- ROS software is organized into *packages*, which can contain source code, launch files, configuration files, message definitions, data, and documentation
- A package that builds up on/requires other packages (e.g. message definitions), declares these as *dependencies*

To create a new package, use

```
> catkin_create_pkg package_name  
  {dependencies}
```

Separate message definition packages from other packages!



More info

<http://wiki.ros.org/Packages>

ROS Packages

package.xml

- The package.xml file defines the properties of the package
 - Package name
 - Version number
 - Authors
 - Dependencies on other packages**
 - ...

package.xml

```
<?xml version="1.0"?>
<package format="2">
  <name>ros_package_template</name>
  <version>0.1.0</version>
  <description>A template for ROS packages.</description>
  <maintainer email="pfankhauser@e...>Peter Fankhauser</maintainer>
  <license>BSD</license>
  <url type="website">https://github.com/ethz-asl/ros_best_pr...</url>
  <author email="pfankhauser@ethz.ch">Peter Fankhauser</author>

  <buildtool_depend>catkin</buildtool_depend>

  <depend>roscpp</depend>
  <depend>sensor_msgs</depend>
</package>
```

More info

<http://wiki.ros.org/catkin/package.xml>

ROS Packages

CMakeLists.xml

The CMakeLists.txt is the input to the CMakebuild system

1. Required CMake Version (`cmake_minimum_required`)
2. Package Name (`project()`)
3. Find other CMake/Catkin packages needed for build (`find_package()`)
4. Message/Service/Action Generators (`add_message_files()`,
`add_service_files()`, `add_action_files()`)
5. Invoke message/service/action generation (`generate_messages()`)
6. Specify package build info export (`catkin_package()`)
7. Libraries/Executables to build
(`add_library()`/`add_executable()`/`target_link_libraries()`)
8. Tests to build (`catkin_add_gtest()`)
9. Install rules (`install()`)

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(ros_package_template)

## Use C++11
add_definitions(--std=c++11)

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

More info

<http://wiki.ros.org/catkin/CMakeLists.txt>

ROS Packages

CMakeLists.xml Example

```
cmake_minimum_required(VERSION 2.8.3)
project(husky_highlevel_controller)
add_definitions(--std=c++11)

find_package(catkin REQUIRED
    COMPONENTS roscpp sensor_msgs
)

catkin_package(
    INCLUDE_DIRS include
    # LIBRARIES
    CATKIN_DEPENDS roscpp sensor_msgs
    # DEPENDS
)

include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(${PROJECT_NAME} src/${PROJECT_NAME}_node.cpp
src/HuskyHighlevelController.cpp)

target_link_libraries(${PROJECT_NAME} ${catkin_LIBRARIES})
```

Use the same name as in the package.xml

We use C++11 by default

List the packages that your package requires to build (have to be listed in package.xml)

Specify build export information

- INCLUDE_DIRS: Directories with header files
- LIBRARIES: Libraries created in this project
- CATKIN_DEPENDS: Packages dependent projects also need
- DEPENDS: System dependencies dependent projects also need (have to be listed in package.xml)

Specify locations of header files

Declare a C++ executable

Specify libraries to link the executable against

Setup a Project in Eclipse

- Build the Eclipse project files with additional build flag

```
> catkin build package_name -G"Eclipse CDT4 - Unix Makefiles"  
-DCMAKE_CXX_COMPILER_ARG1=-std=c++11
```

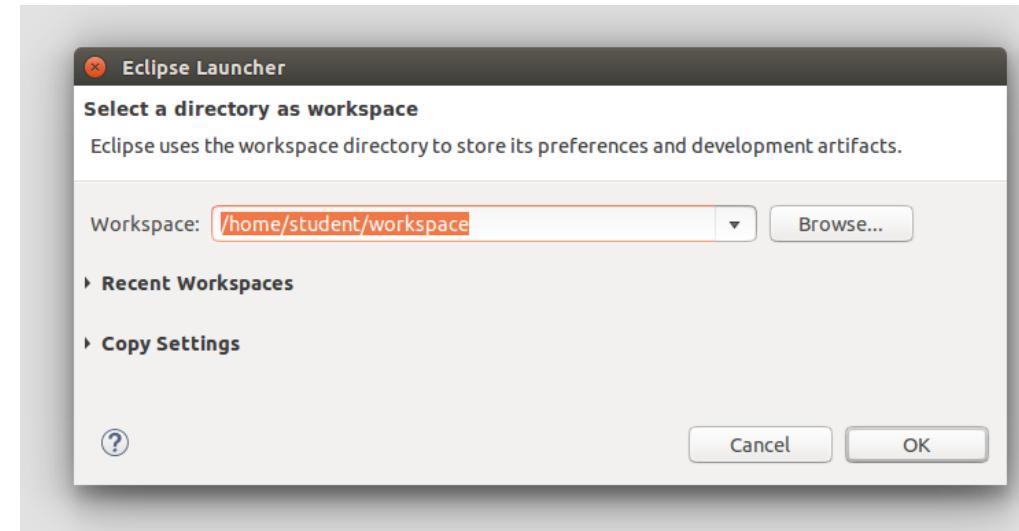
- The project files will be generated in `~/catkin_ws/build`



The build flags are already setup in the provided installation.

Setup a Project in Eclipse

- Start Eclipse and set the workspace folder

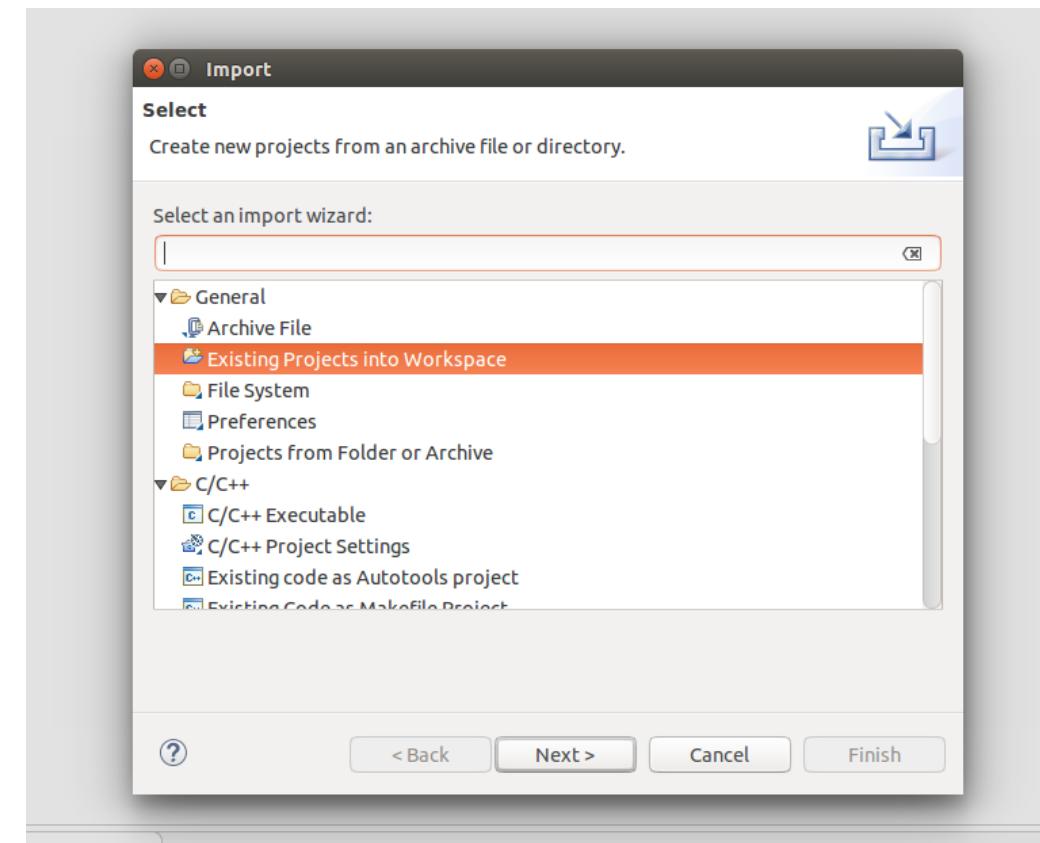


The Eclipse workspace is already set in the provided installation.

Setup a Project in Eclipse

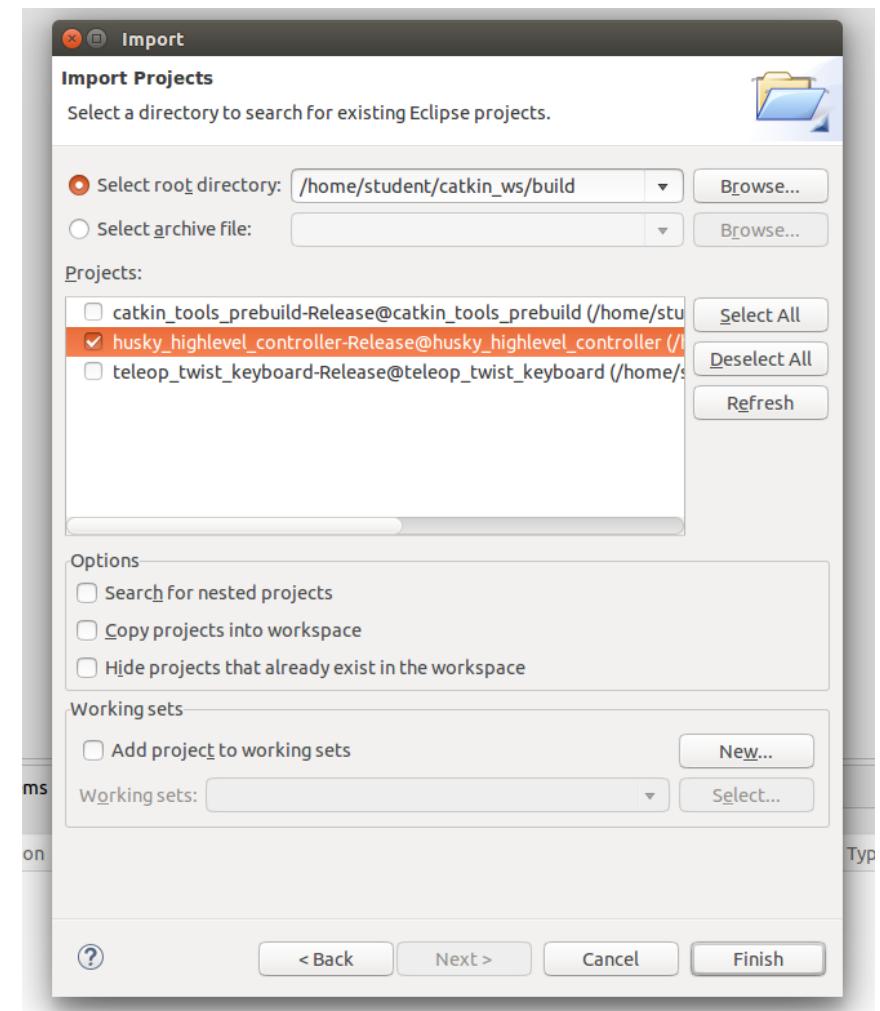
- Import your project to Eclipse

File → Import → General
→ Existing Projects into Workspace



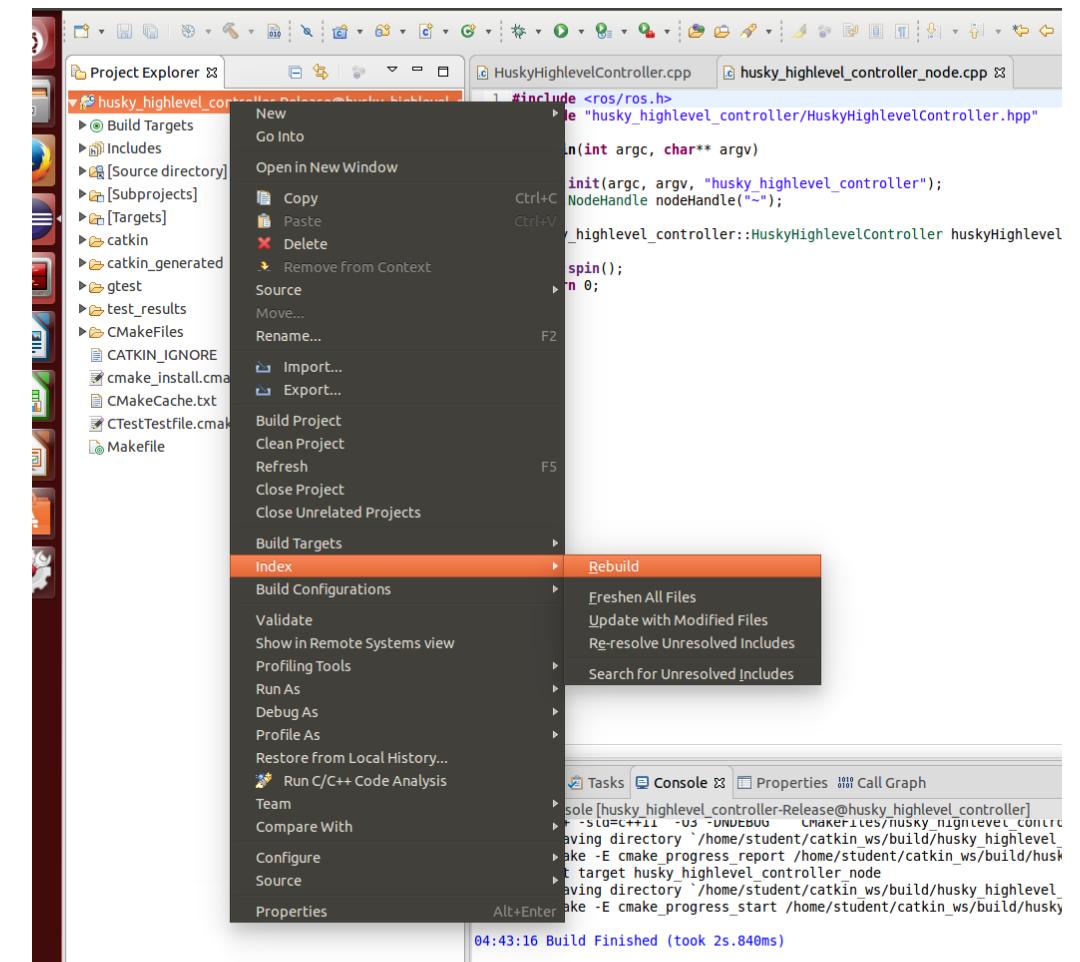
Setup a Project in Eclipse

- The project files can be imported from the `~/catkin_ws/build` folder



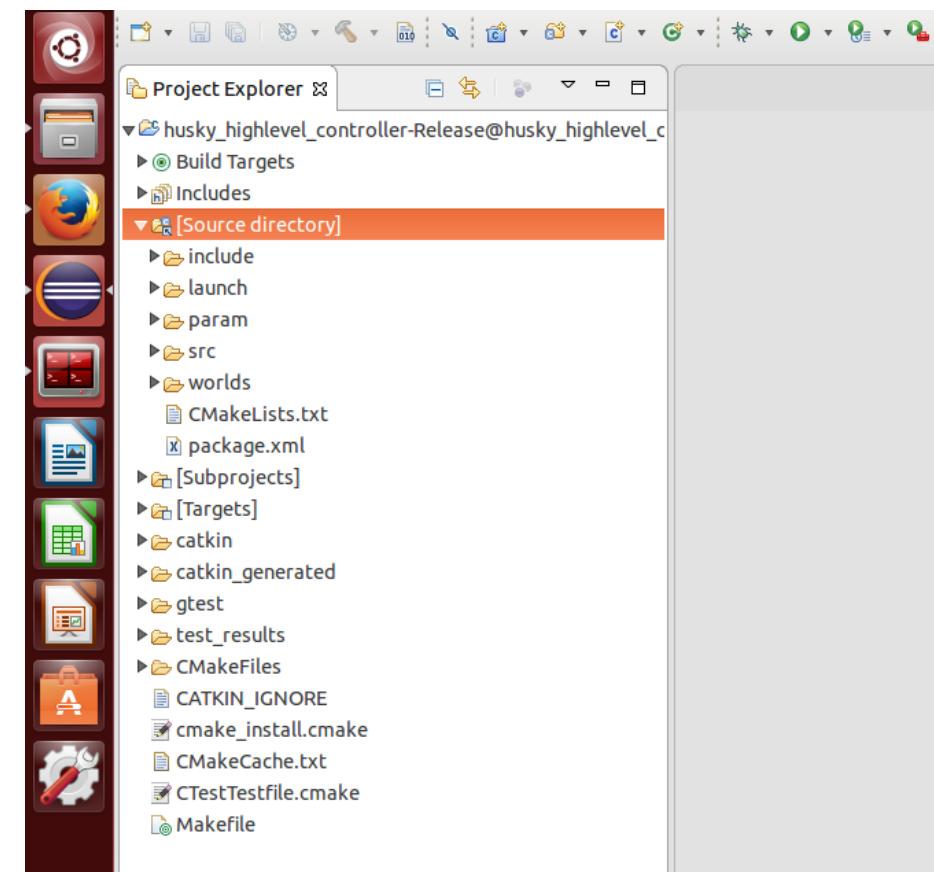
Setup a Project in Eclipse

- Rebuild the C/C++ index of your project by
Right click on Project → Index → Rebuild
- Resolving the includes enables
 - Fast navigation through links (Ctrl + click)
 - Auto-completion (Ctrl + Space)
 - Building (Ctrl + B) and debugging your code in
Eclipse



Setup a Project in Eclipse

- Within the project a link [Source directory] is provided such that you can edit your project
- Useful Eclipse shortcuts
 - Ctrl + Space: Auto-complete
 - Ctrl + /: Comment / uncomment line or section
 - Ctrl + Shift + F: Auto-format code using code formatter
 - Alt + Arrow Up / Arrow Down: Move line or selection up or down
 - Ctrl + D: Delete line



ROS C++ Client Library (*roscpp*)

hello_world.cpp

```
#include <ros/ros.h>
int main(int argc, char** argv)
{
    ros::init(argc, argv, "hello_world");
    ros::NodeHandle nodeHandle;
    ros::Rate loopRate(10);

    unsigned int count = 0;
    while (ros::ok()) {
        ROS_INFO_STREAM("Hello World " << count);
        ros::spinOnce();
        loopRate.sleep();
        count++;
    }

    return 0;
}
```

ROS main header file include

ros::init(...) has to be called before calling other ROS functions

The node handle is the access point for communications with the ROS system (topics, services, parameters)

ros::Rate is a helper class to run loops at a desired frequency

ros::ok() checks if a node should continue running
Returns false if SIGINT is received (Ctrl + C) or ros::shutdown() has been called

ROS_INFO() logs messages to the filesystem

ros::spinOnce() processes incoming messages via callbacks

More info

<http://wiki.ros.org/roscpp>

<http://wiki.ros.org/roscpp/Overview>

ROS C++ Client Library (*roscpp*)

Node Handle

- There are four main types of node handles

- Default (public) node handle:

```
nh_ = ros::NodeHandle();
```

- Private node handle:

```
nh_private_ = ros::NodeHandle("~");
```

- Namespaced node handle:

```
nh_eth_ = ros::NodeHandle("eth");
```

- Global node handle:

```
nh_global_ = ros::NodeHandle("/");
```

Recommended

Not recommended

For a *node* in *namespace* looking up *topic*,
these will resolve to:

/namespace/topic

/namespace/node/topic

/namespace/eth/topic

/topic

More info

<http://wiki.ros.org/roscpp/Overview/NodeHandles>

ROS C++ Client Library (*roscpp*)

Logging

- Mechanism for logging human readable text from nodes in the console and to log files
- Instead of `std::cout`, use e.g. `ROS_INFO`
- Automatic logging to console, log file, and `/rosout` topic
- Different severity levels (Info, Warn, Error etc.)
- Supports both `printf`- and stream-style formatting

```
ROS_INFO("Result: %d", result);
ROS_INFO_STREAM("Result: " << result);
```

- Further features such as conditional, throttled, delayed logging etc.

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

! To see the output in the console, set the output configuration to screen in the launch file

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

More info

<http://wiki.ros.org/rosconsole>

<http://wiki.ros.org/roscpp/Overview/Logging>

ROS C++ Client Library (*roscpp*)

Subscriber

- Start listening to a topic by calling the method `subscribe()` of the node handle

```
ros::Subscriber subscriber =
nodeHandle.subscribe(topic, queue_size,
                     callback_function);
```

- When a message is received, callback function is called with the contents of the message as argument
- Hold on to the subscriber object until you want to unsubscribe

`ros::spin()` processes callbacks and will not return until the node has been shutdown

listener.cpp

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

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

int main(int argc, char **argv)
{
    ros::init(argc, argv, "listener");
    ros::NodeHandle nodeHandle;

    ros::Subscriber subscriber =
        nodeHandle.subscribe("chatter",10, chatterCallback);
    ros::spin();
    return 0;
}
```

More info

<http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers>

ROS C++ Client Library (*roscpp*)

Publisher

- Create a publisher with help of the node handle

```
ros::Publisher publisher =  
nodeHandle.advertise<message_type>(topic,  
queue_size);
```

- Create the message contents
- Publish the contents with

```
publisher.publish(message);
```

More info

<http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers>

talker.cpp

```
#include <ros/ros.h>  
#include <std_msgs/String.h>  
  
int main(int argc, char **argv) {  
    ros::init(argc, argv, "talker");  
    ros::NodeHandle nh;  
    ros::Publisher chatterPublisher =  
        nh.advertise<std_msgs::String>("chatter", 1);  
    ros::Rate loopRate(10);  
  
    unsigned int count = 0;  
    while (ros::ok()) {  
        std_msgs::String message;  
        message.data = "hello world " + std::to_string(count);  
        ROS_INFO_STREAM(message.data);  
        chatterPublisher.publish(message);  
        ros::spinOnce();  
        loopRate.sleep();  
        count++;  
    }  
    return 0;  
}
```

ROS C++ Client Library (*roscpp*)

Object Oriented Programming



my_package_node.cpp

```
#include <ros/ros.h>
#include "my_package/MyPackage.hpp"
int main(int argc, char** argv)
{
    ros::init(argc, argv, "my_package");
    ros::NodeHandle nodeHandle("~");

    my_package::MyPackage myPackage(nodeHandle);

    ros::spin();
    return 0;
}
```



MyPackage.hpp



MyPackage.cpp

class MyPackage

Main node class
providing ROS interface
(subscribers, parameters,
timers etc.)



Algorithm.hpp



Algorithm.cpp

class Algorithm

Class implementing the
algorithmic part of the
node

*Note: The algorithmic part of the
code could be separated in a
(ROS-independent) library*



Specify a function handler to a method from within the class as

```
subscriber_ = nodeHandle_.subscribe(topic, queue_size,
&ClassName::methodName, this);
```

More info

[http://wiki.ros.org/roscpp_tutorials/Tutorials/
UsingClassMethodsAsCallbacks](http://wiki.ros.org/roscpp_tutorials/Tutorials/UsingClassMethodsAsCallbacks)

ROS Parameter Server

- Nodes use the *parameter server* to store and retrieve parameters at runtime
- Best used for static data such as configuration parameters
- Parameters can be defined in launch files or separate YAML files

List all parameters with

```
> rosparam list
```

Get the value of a parameter with

```
> rosparam get parameter_name
```

Set the value of a parameter with

```
> rosparam set parameter_name value
```

config.yaml

```
camera:  
  left:  
    name: left_camera  
    exposure: 1  
  right:  
    name: right_camera  
    exposure: 1.1
```

package.launch

```
<launch>  
  <node name="name" pkg="package" type="node_type">  
    <rosparam command="load"  
      file="$(find package)/config/config.yaml" />  
  </node>  
</launch>
```

More info

<http://wiki.ros.org/rosparam>

ROS Parameter Server

C++ API

- Get a parameter in C++ with

```
nodeHandle.getParam(parameter_name, variable)
```

- Method returns true if parameter was found,
false otherwise
- Global and relative parameter access:

- Global parameter name with preceding /

```
nodeHandle.getParam("/package/camera/left/exposure", variable)
```

- Relative parameter name (relative to the node handle)

```
nodeHandle.getParam("camera/left/exposure", variable)
```

- For parameters, typically use the private node handle
`ros::NodeHandle("~")`

```
ros::NodeHandle nodeHandle("~");
std::string topic;
if (!nodeHandle.getParam("topic", topic)) {
    ROS_ERROR("Could not find topic
              parameter!");
}
```

More info

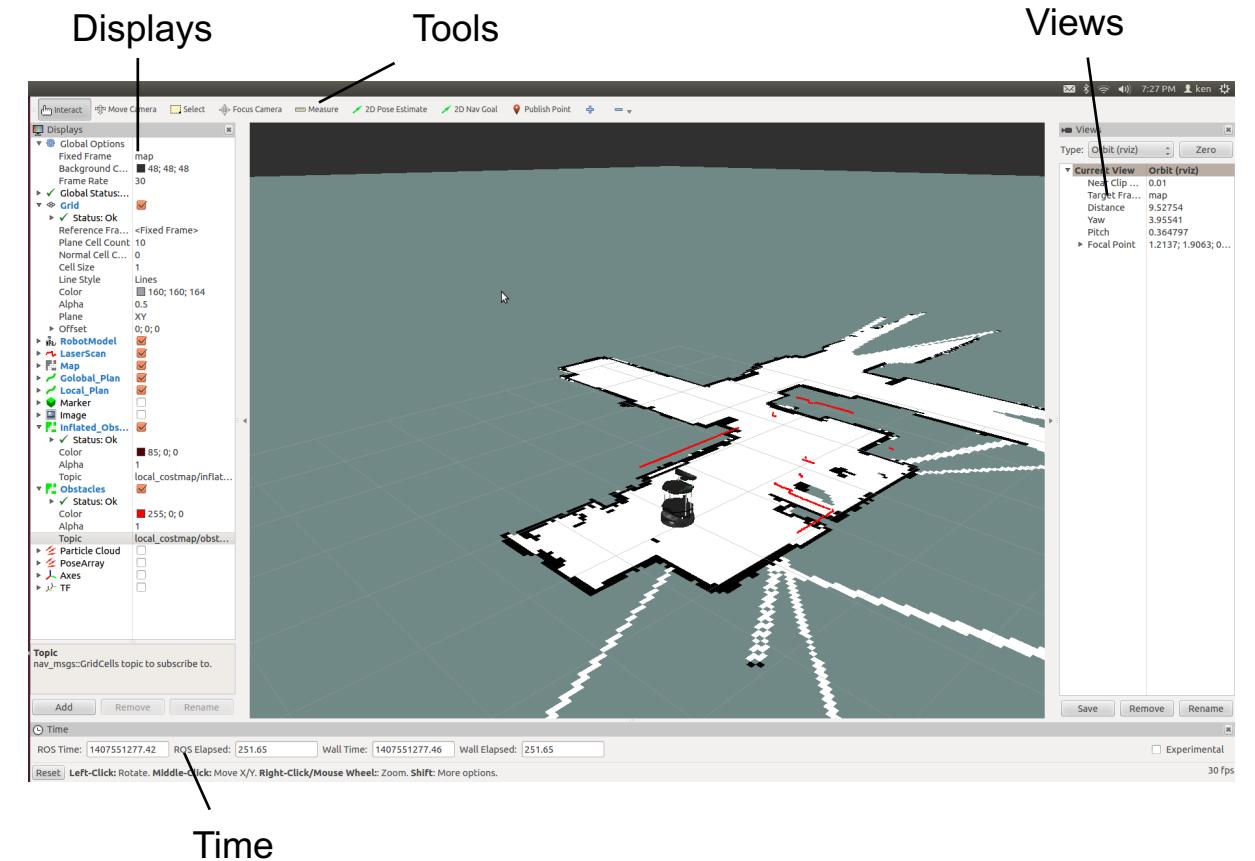
<http://wiki.ros.org/roscpp/Overview/Parameter%20Server>

RViz

- 3D visualization tool for ROS
- Subscribes to topics and visualizes the message contents
- Different camera views (orthographic, top-down, etc.)
- Interactive tools to publish user information
- Save and load setup as RViz configuration
- Extensible with plugins

Run RViz with

```
> rosrun rviz rviz
```

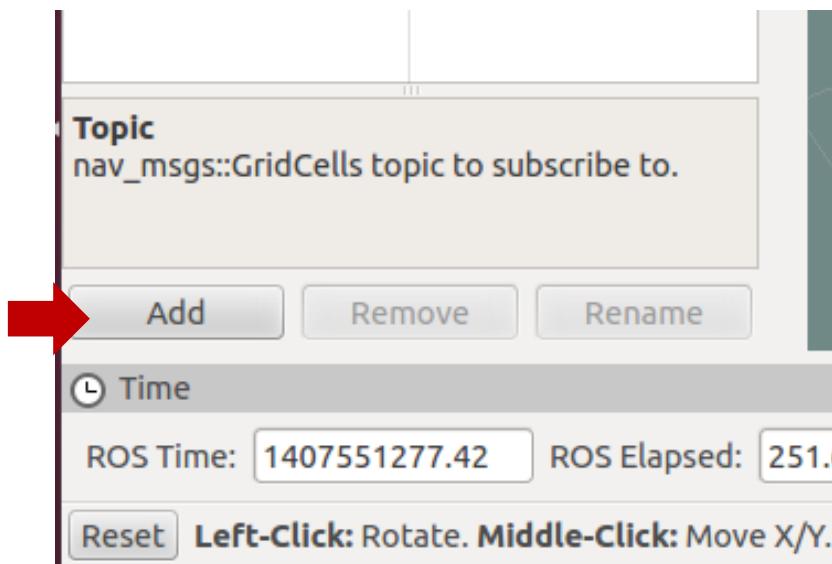


More info

<http://wiki.ros.org/rviz>

RViz

Display Plugins



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

RViz

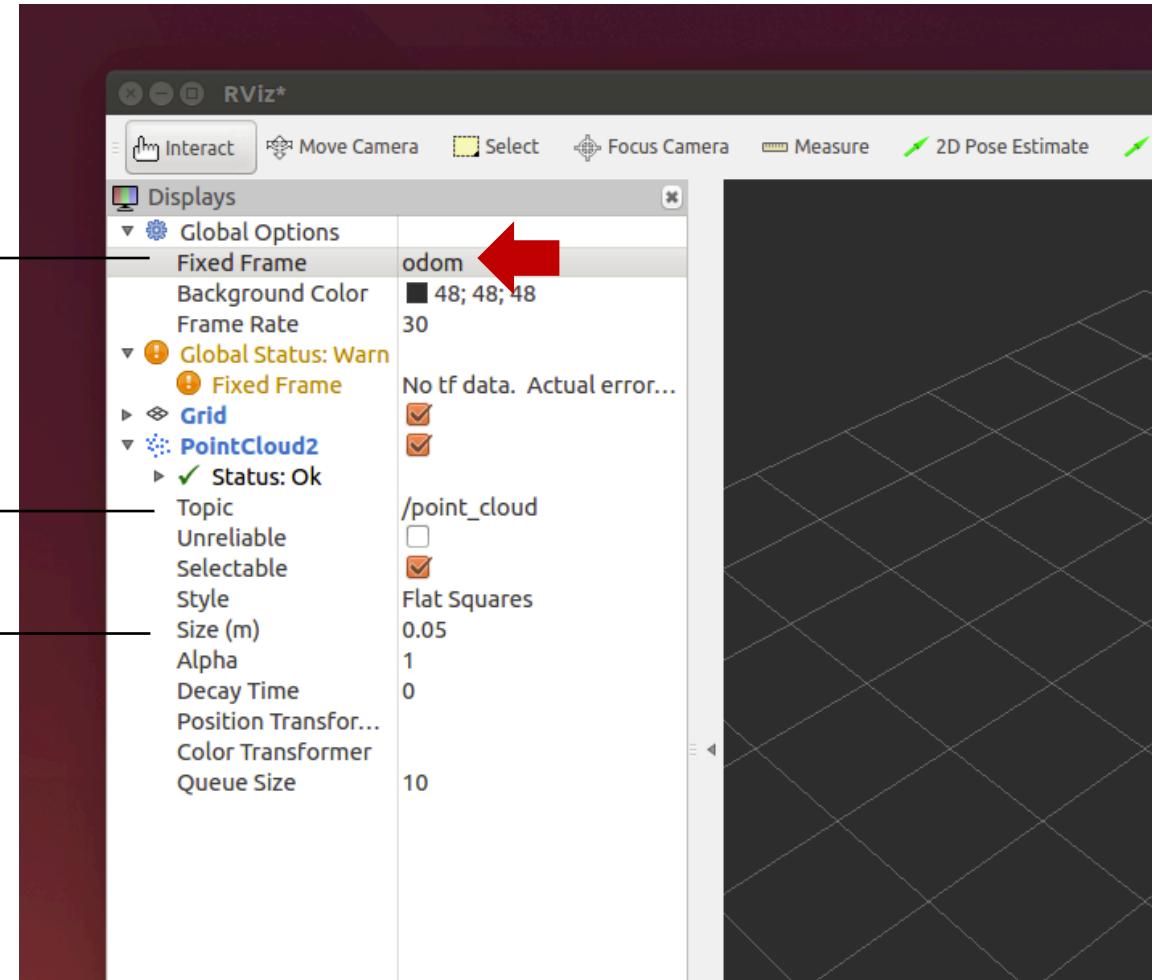
Visualizing Point Clouds Example

!

Frame in which the data is displayed (has to exist!)

Choose the topic for the display

Change the display options (e.g. size)

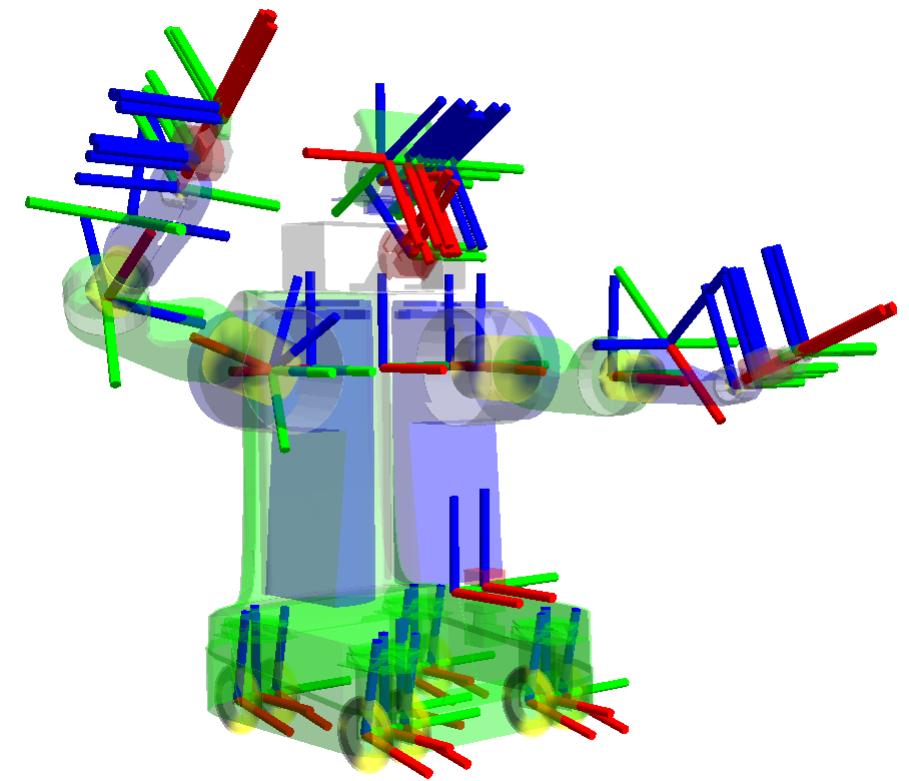
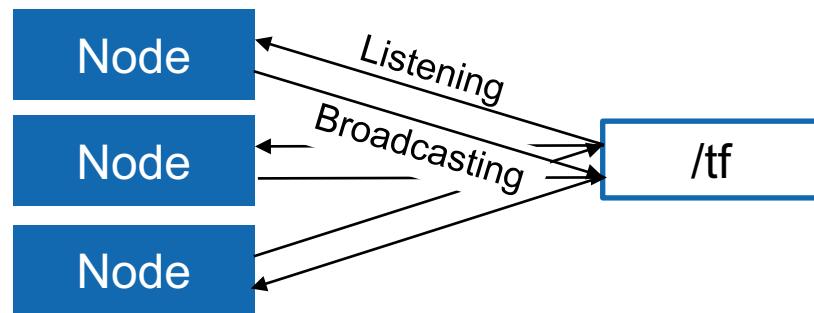


Overview Part 3

- TF Transformation System
- rqt User Interface
- Robot models (URDF)
- Simulation descriptions (SDF)

TF Transformation System

- Tool for keeping track of coordinate frames over time
- Maintains relationship between coordinate frames in a tree structure buffered in time
- Lets the user transform points, vectors, etc. between coordinate frames at desired time
- Implemented as publisher/subscriber model on the topics `/tf` and `/tf_static`



More info
<http://wiki.ros.org/tf2>

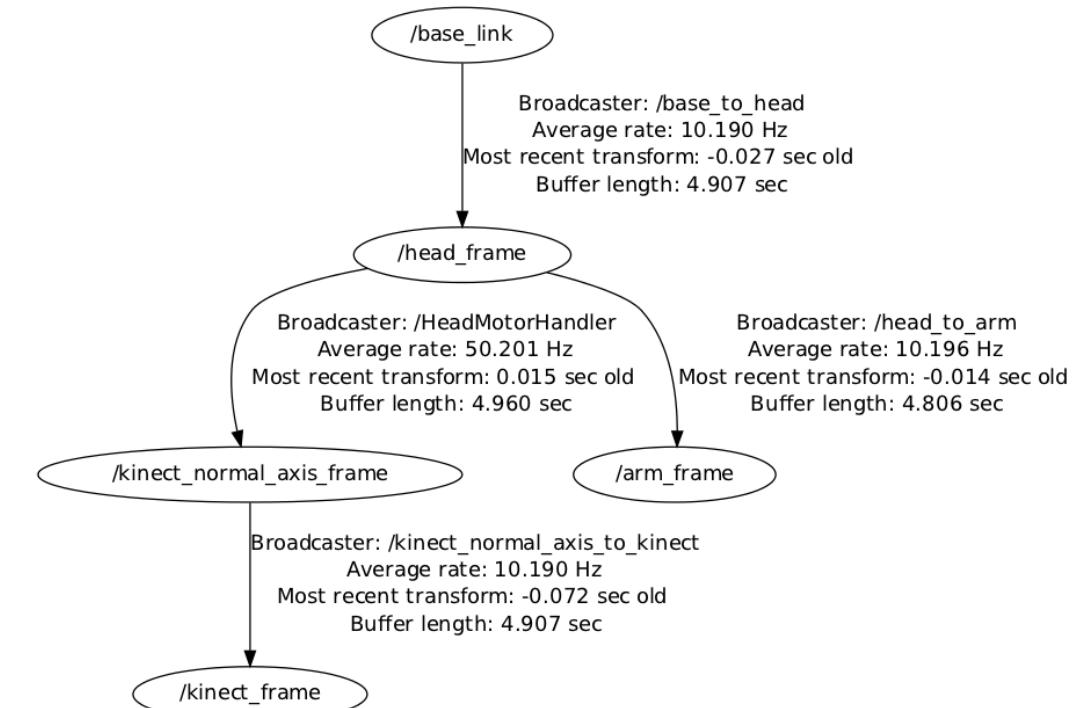
TF Transformation System

Transform Tree

- TF listeners use a buffer to listen to all broadcasted transforms
- Query for specific transforms from the transform tree

[tf2_msgs/TFMessage.msg](#)

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



TF Transformation System

Tools

Command line

Print information about the current transform tree

```
> rosrun tf tf_monitor
```

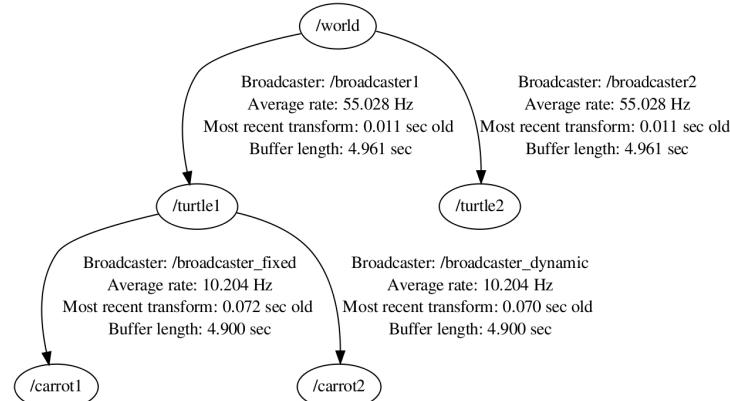
Print information about the transform between two frames

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

View Frames

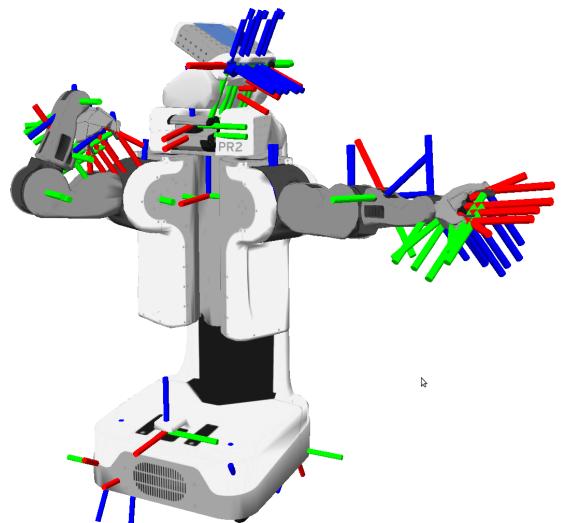
Creates a visual graph (PDF) of the transform tree

```
> rosrun tf view_frames
```



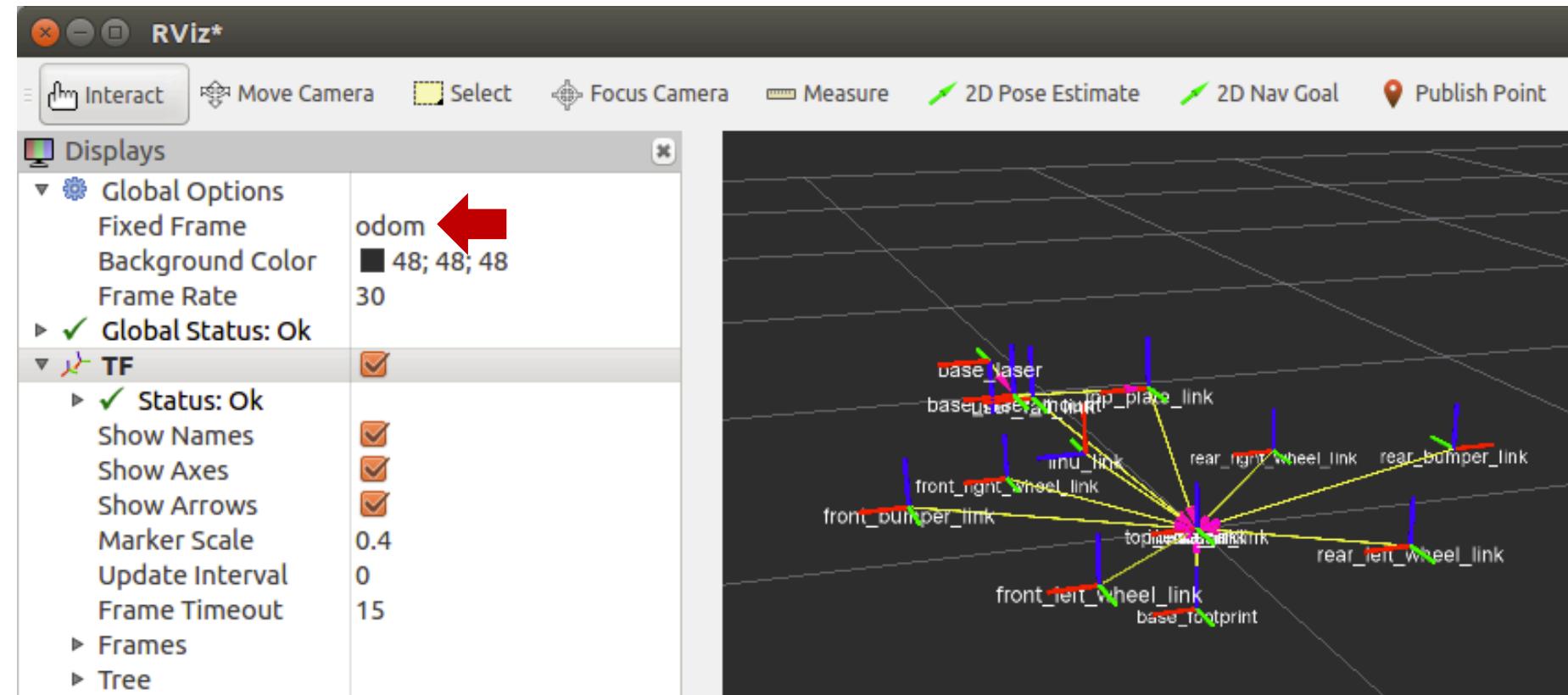
RViz

3D visualization of the transforms



TF Transformation System

RViz Plugin



TF Transformation System

Transform Listener C++ API

- Create a TF listener to fill up a buffer

```
tf2_ros::Buffer tfBuffer;
tf2_ros::TransformListener tfListener(tfBuffer);
```

- Make sure, that the listener does not run out of scope!
- To lookup transformations, use

```
geometry_msgs::TransformStamped transformStamped =
tfBuffer.lookupTransform(target_frame_id,
                         source_frame_id, ros::Time(0));
```

- For time, use `ros::Time(0)` to get the latest available transform

```
#include <ros/ros.h>
#include <tf2_ros/transform_listener.h>
#include <geometry_msgs/TransformStamped.h>

int main(int argc, char** argv) {
    ros::init(argc, argv, "tf2_listener");
    ros::NodeHandle nodeHandle;
    tf2_ros::Buffer tfBuffer;
    tf2_ros::TransformListener tfListener(tfBuffer);

    ros::Rate rate(10.0);
    while (nodeHandle.ok()) {
        geometry_msgs::TransformStamped transformStamped;
        try {
            transformStamped = tfBuffer.lookupTransform("base",
                                                       "odom", ros::Time(0));
        } catch (tf2::TransformException &exception) {
            ROS_WARN("%s", exception.what());
            ros::Duration(1.0).sleep();
            continue;
        }
        rate.sleep();
    }
    return 0;
};
```

More info

<http://wiki.ros.org/tf2/Tutorials/Writing%20a%20tf2%20listener%20%28C%2B%2B%29>

rqt User Interface

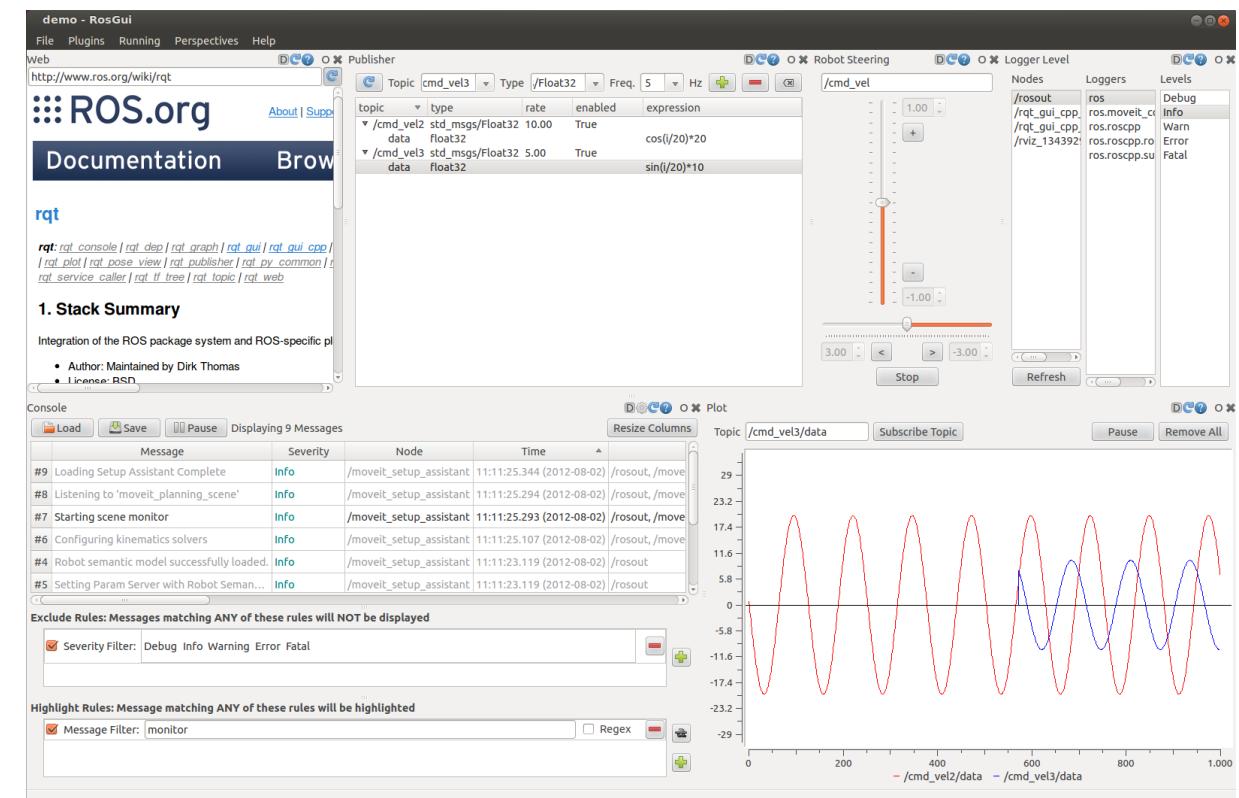
- User interface base on Qt
- Custom interfaces can be setup
- Lots of existing plugins exist
- Simple to write own plugins

Run RQT with

```
> rosrun rqt_gui rqt_gui
```

or

```
> rqt
```



More info

<http://wiki.ros.org/rqt/Plugins>

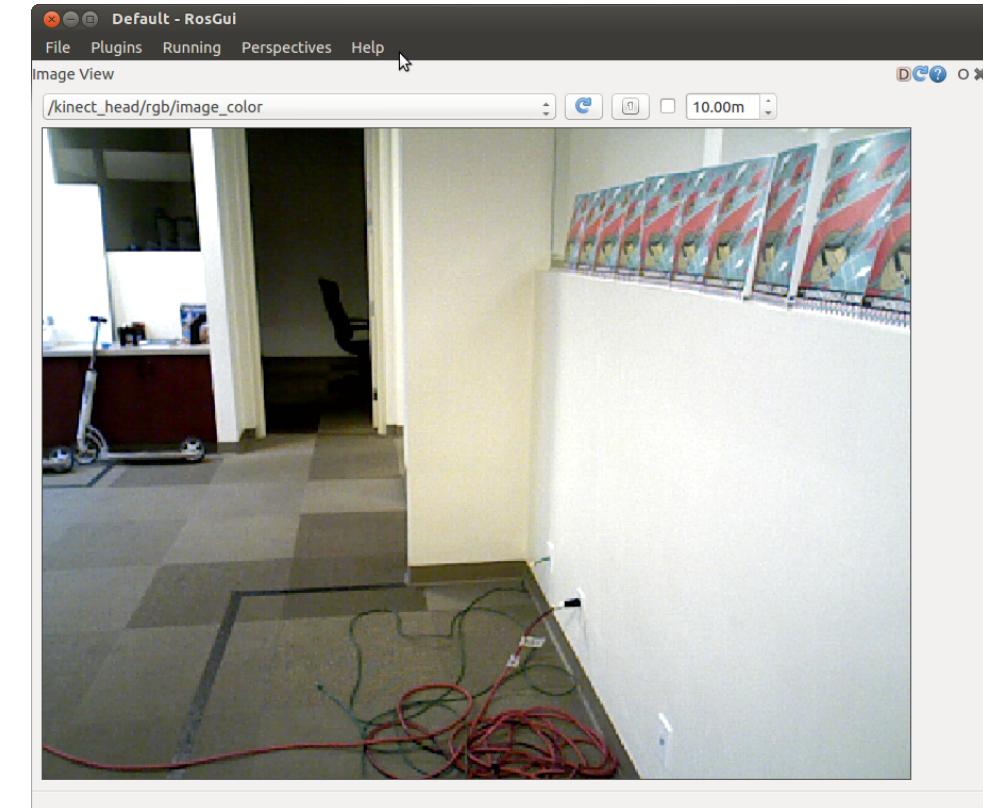
rqt User Interface

rqt_image_view

- Visualizing images

Run *rqt_graph* with

```
> rosrun rqt_image_view rqt_image_view
```



More info

http://wiki.ros.org/rqt_image_view

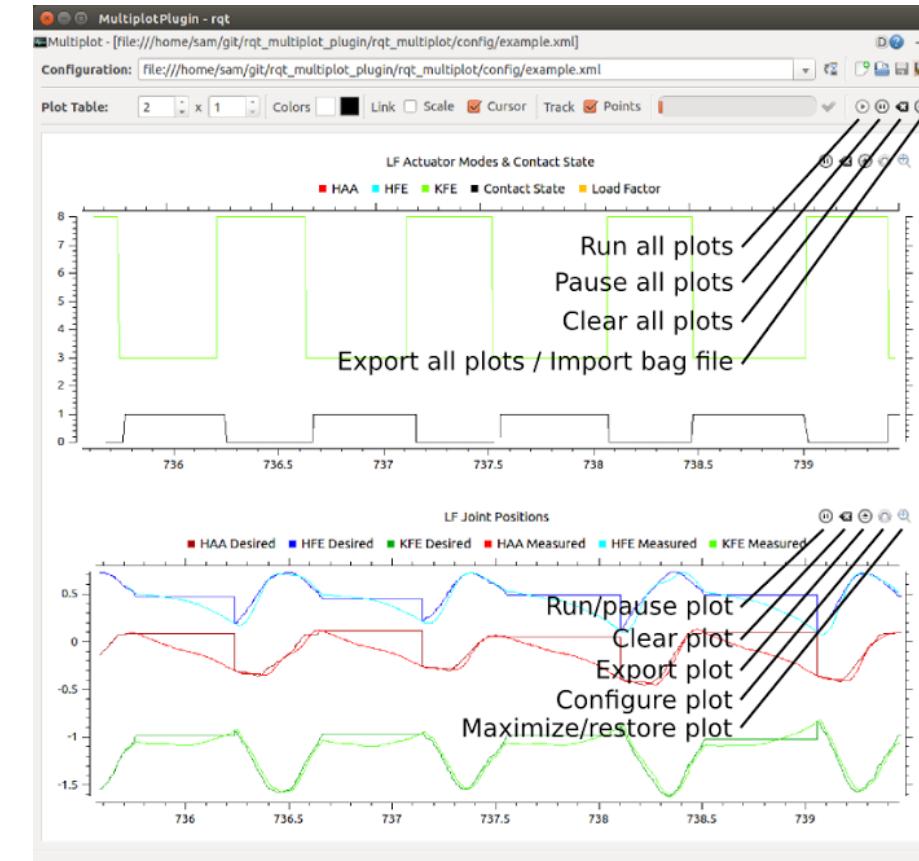
rqt User Interface

rqt_multiplot

- Visualizing numeric values in 2D plots

Run *rqt_multiplot* with

```
> rosrun rqt_multiplot rqt_multiplot
```



More info

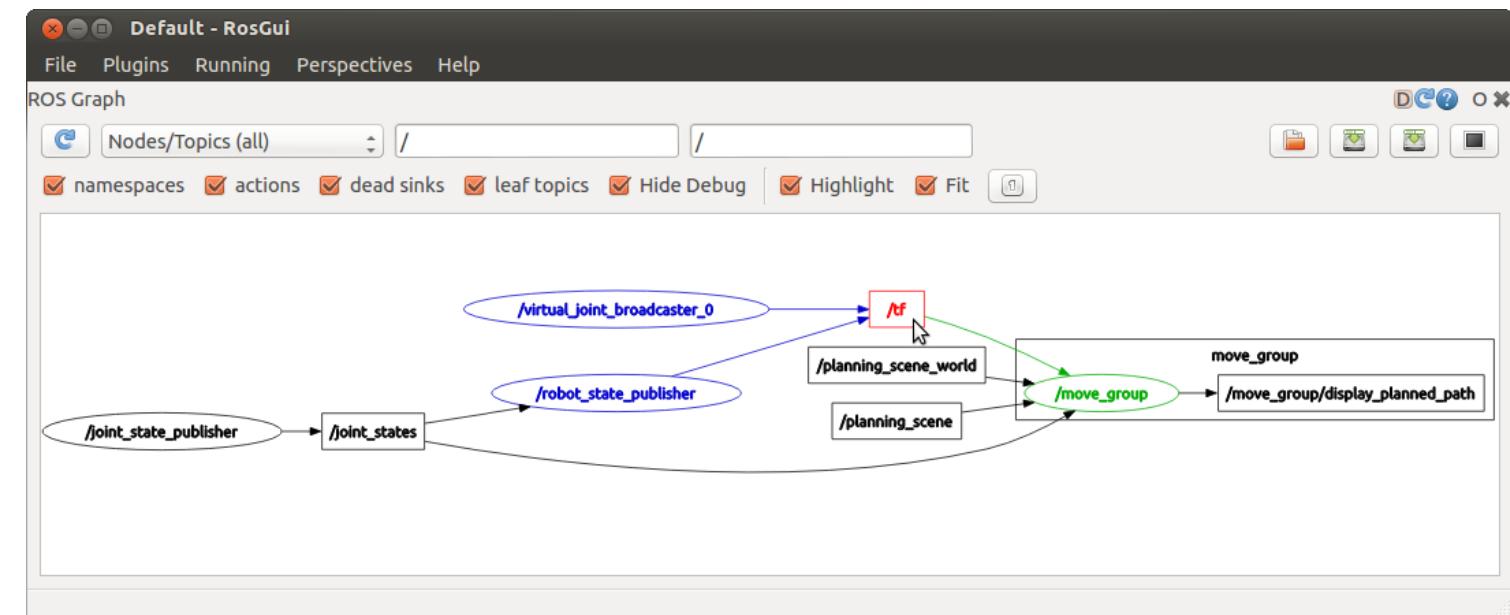
http://wiki.ros.org/rqt_multiplot

rqt User Interface rqt_graph

- Visualizing the ROS computation graph

Run *rqt_graph* with

```
> rosrun rqt_graph rqt_graph
```



More info

http://wiki.ros.org/rqt_graph

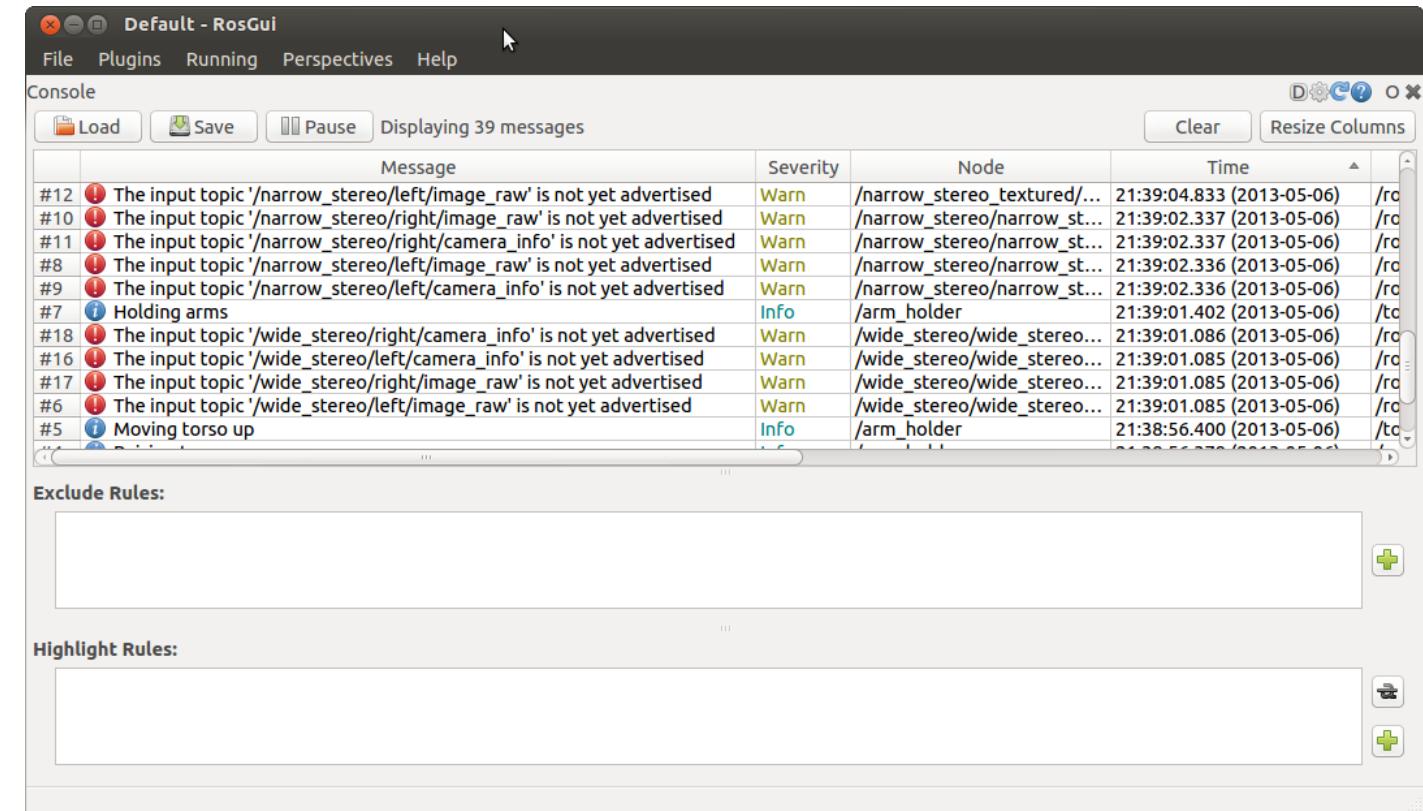
rqt User Interface

rqt_console

- Displaying and filtering ROS messages

Run *rqt_console* with

```
> rosrun rqt_console rqt_console
```



More info

http://wiki.ros.org/rqt_console

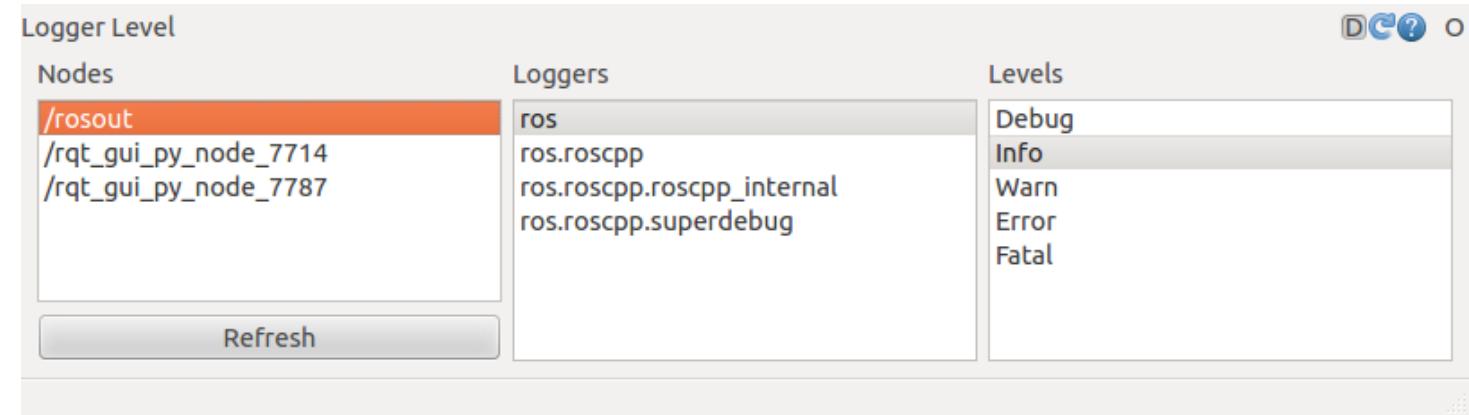
rqt User Interface

rqt_logger_level

- Configuring the logger level of ROS nodes

Run *rqt_logger_level* with

```
> rosrun rqt_logger_level  
rqt_logger_level
```



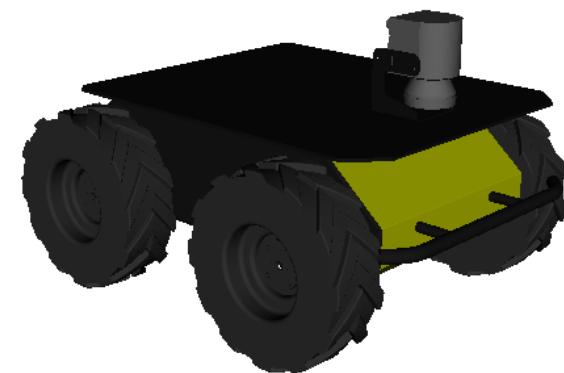
More info

http://wiki.ros.org/rqt_logger_level

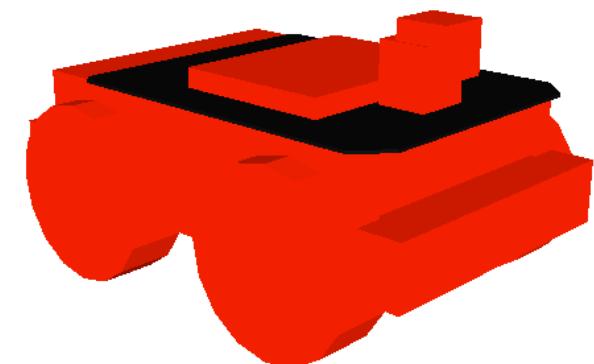
Robot Models

Unified Robot Description Format (URDF)

- Defines an XML format for representing a robot model
 - Kinematic and dynamic description
 - Visual representation
 - Collision model
- URDF generation can be scripted with XACRO



Mesh for visuals



Primitives for collision

More info

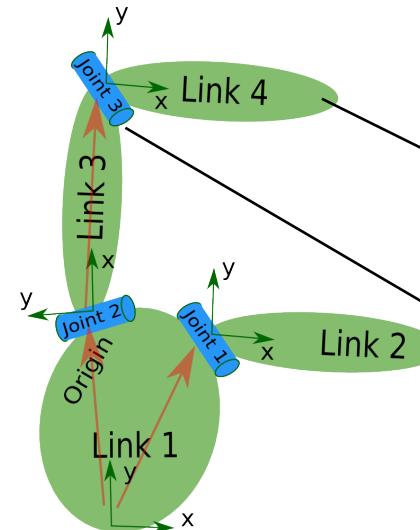
<http://wiki.ros.org/urdf>

<http://wiki.ros.org/xacro>

Robot Models

Unified Robot Description Format (URDF)

- Description consists of a set of *link* elements and a set of *joint* elements
- Joints connect the links together



robot.urdf

```
<robot name="robot">
  <link> ... </link>
  <link> ... </link>
  <link> ... </link>

  <joint> .... </joint>
  <joint> .... </joint>
  <joint> .... </joint>
</robot>
```

```
<link name="Link_name">
  <visual>
    <geometry>
      <mesh filename="mesh.dae"/>
    </geometry>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="10"/>
    <inertia ixx="0.4" ixy="0.0" .../>
  </inertial>
</link>
```

```
<joint name="joint_name" type="revolute">
  <axis xyz="0 0 1"/>
  <limit effort="1000.0" upper="0.548" ... />
  <origin rpy="0 0 0" xyz="0.2 0.01 0"/>
  <parent link="parent_Link_name/>
  <child link="child_Link_name/>
</joint>
```

More info

<http://wiki.ros.org/urdf/XML/model>

Robot Models

Usage in ROS

- The robot description (URDF) is stored on the parameter server (typically) under `/robot_description`
- You can visualize the robot model in Rviz with the  *RobotModel* plugin

husky_empty_world.launch

```
...
<include file="$(find husky_gazebo)/launch/spawn_husky.launch">
  <arg name="laser_enabled" value="$(arg laser_enabled)"/>
  <arg name="ur5_enabled" value="$(arg ur5_enabled)"/>
  <arg name="kinect_enabled" value="$(arg kinect_enabled)"/>
</include>
...
```

spawn_husky.launch

```
...
<param name="robot_description" command="$(find xacro)/xacro.py
'$(arg husky_gazebo_description)'
  laser_enabled:=$(arg laser_enabled)
  ur5_enabled:=$(arg ur5_enabled)
  kinect_enabled:=$(arg kinect_enabled)" />
...
...
```

Simulation Descriptions

Simulation Description Format (SDF)

- Defines an XML format to describe
 - Environments (lighting, gravity etc.)
 - Objects (static and dynamic)
 - Sensors
 - Robots
- SDF is the standard format for Gazebo
- Gazebo converts a URDF to SDF automatically



More info

<http://sdformat.org>

Overview Part 4

- ROS services
- ROS actions (actionlib)
- ROS time
- ROS bags
- Debugging strategies

ROS Services

- Request/response communication between nodes is realized with *services*
 - The *service server* advertises the service
 - The *service client* accesses this service
- Similar in structure to messages, services are defined in **.srv* files

List available services with

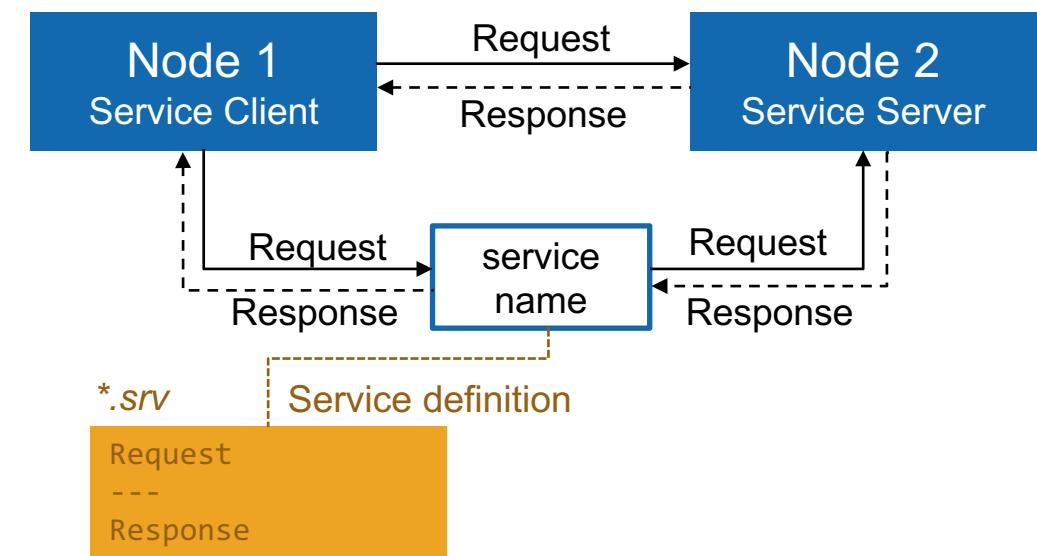
```
> rosservice list
```

Show the type of a service

```
> rosservice type /service_name
```

Call a service with the request contents

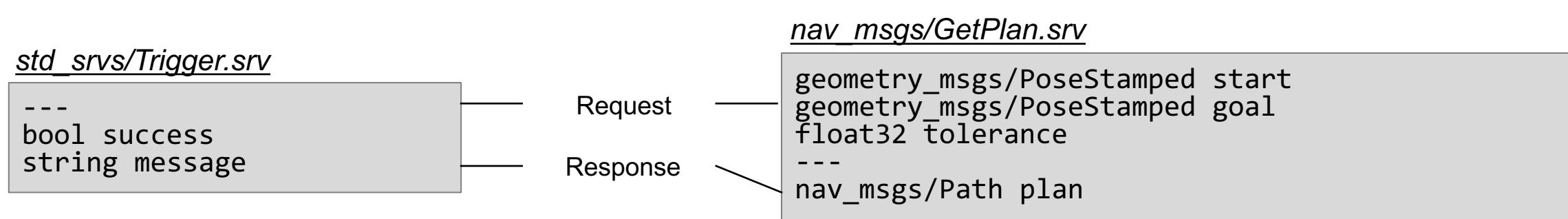
```
> rosservice call /service_name args
```



More info

<http://wiki.ros.org/Services>

ROS Services Examples



ROS Service Example

Starting a *roscore* and a *add_two_ints_server* node

In console nr. 1:

Start a roscore with

```
> roscore
```

```
PARAMETERS
  * /rosdistro: indigo
  * /rosversion: 1.11.20

NODES

auto-starting new master
process[master]: started with pid [6708]
ROS_MASTER_URI=http://ubuntu:11311/

setting /run_id to 6c1852aa-e961-11e6-8543-000c297bd368
process[rosout-1]: started with pid [6721]
started core service [/rosout]
```

In console nr. 2:

Run a service demo node with

```
> rosrun roscpp_tutorials add_two_ints_server
```

```
student@ubuntu:~$ rosrun roscpp_tutorials add_two_ints_server
```

ROS Service Example

Console Nr. 3 – Analyze and call service

See the available services with

```
> rosservice list
```

See the type of the service with

```
> rosservice type /add_two_ints
```

Show the service definition with

```
> rossrv show roscpp_tutorials/TwoInts
```

Call the service (use Tab for auto-complete)

```
> rosservice call /add_two_ints "a: 10  
b: 5"
```

```
student@ubuntu:~$ rosservice list  
/add_two_ints ←  
/add_two_ints_server/get_loggers  
/add_two_ints_server/set_logger_level  
/rosout/get_loggers  
/rosout/set_logger_level
```

```
student@ubuntu:~$ rosservice type /add_two_ints  
roscpp_tutorials/TwoInts
```

```
student@ubuntu:~$ rossrv show roscpp_tutorials/TwoInts  
int64 a  
int64 b  
---  
int64 sum
```

```
student@ubuntu:~$ rosservice call /add_two_ints "a: 10  
b: 5"  
sum: 15
```

ROS C++ Client Library (*roscpp*)

Service Server

- Create a service server with

```
ros::ServiceServer service =
nodeHandle.advertiseService(service_name,
                             callback_function);
```

- When a service request is received, callback function is called with the request as argument
- Fill in the response to the response argument
- Return to function with true to indicate that it has been executed properly

More info

<http://wiki.ros.org/roscpp/Overview/Services>

add_two_ints_server.cpp

```
#include <ros/ros.h>
#include <roscpp_tutorials/TwoInts.h>

bool add(roscpp_tutorials::TwoInts::Request &request,
          roscpp_tutorials::TwoInts::Response &response)
{
    response.sum = request.a + request.b;
    ROS_INFO("request: x=%ld, y=%ld", (long int)request.a,
             (long int)request.b);
    ROS_INFO(" sending back response: [%ld]",
             (long int)response.sum);
    return true;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "add_two_ints_server");
    ros::NodeHandle nh;
    ros::ServiceServer service =
        nh.advertiseService("add_two_ints", add);
    ros::spin();
    return 0;
}
```

ROS C++ Client Library (*roscpp*)

Service Client

- Create a service client with

```
ros::ServiceClient client =
    nodeHandle.serviceClient<service_type>
        (service_name);
```

- Create service request contents
service.request

```
service.request
client.call(service);
```

- Call service with

More info

<http://wiki.ros.org/roscpp/Overview/Services>

add_two_ints_client.cpp

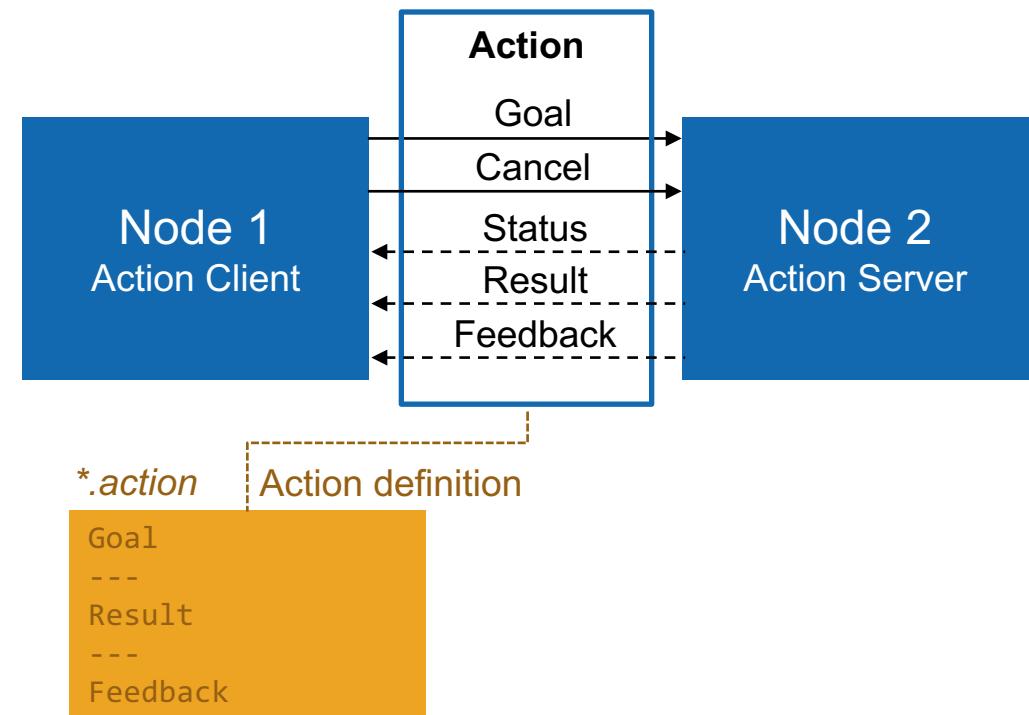
```
#include <ros/ros.h>
#include <roscpp_tutorials/TwoInts.h>
#include <cstdlib>

int main(int argc, char **argv) {
    ros::init(argc, argv, "add_two_ints_client");
    if (argc != 3) {
        ROS_INFO("usage: add_two_ints_client X Y");
        return 1;
    }

    ros::NodeHandle nh;
    ros::ServiceClient client =
        nh.serviceClient<roscpp_tutorials::TwoInts>("add_two_ints");
    roscpp_tutorials::TwoInts service;
    service.request.a = atoi(argv[1]);
    service.request.b = atoi(argv[2]);
    if (client.call(service)) {
        ROS_INFO("Sum: %ld", (long int)service.response.sum);
    } else {
        ROS_ERROR("Failed to call service add_two_ints");
        return 1;
    }
    return 0;
}
```

ROS Actions (actionlib)

- Similar to service calls, but provide possibility to
 - Cancel the task (preempt)
 - Receive feedback on the progress
- Best way to implement interfaces to time-extended, goal-oriented behaviors
- Similar in structure to services, action are defined in `*.action` files
- Internally, actions are implemented with a set of topics



More info

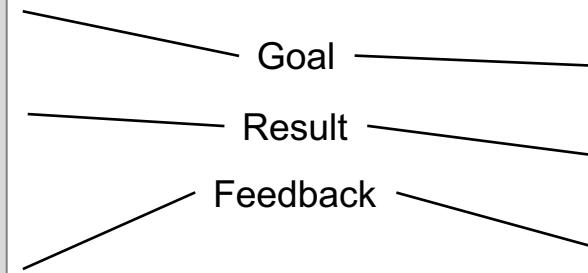
<http://wiki.ros.org/actionlib>

<http://wiki.ros.org/actionlib/DetailedDescription>

ROS Actions (actionlib)

Averaging.action

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



FollowPath.action

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

ROS Parameters, Dynamic Reconfigure, Topics, Services, and Actions Comparison

	Parameters	Dynamic Reconfigure	Topics	Services	Actions
Description	Global constant parameters	Local, changeable parameters	Continuous data streams	Blocking call for processing a request	Non-blocking, preemptable goal oriented tasks
Application	Constant settings	Tuning parameters	One-way continuous data flow	Short triggers or calculations	Task executions and robot actions
Examples	Topic names, camera settings, calibration data, robot setup	Controller parameters	Sensor data, robot state	Trigger change, request state, compute quantity	Navigation, grasping, motion execution

ROS Time

- Normally, ROS uses the PC's system clock as time source (*wall time*)
- For simulations or playback of logged data, it is convenient to work with a simulated time (pause, slow-down etc.)
- To work with a simulated clock:
 - Set the `/use_sim_time` parameter

```
> rosparam set use_sim_time true
```
 - Publish the time on the topic `/clock` from
 - Gazebo (enabled by default)
 - ROS bag (use option `--clock`)

- To take advantage of the simulated time, you should always use the ROS Time APIs:
 - **ros::Time**

```
ros::Time begin = ros::Time::now();  
double secs = begin.toSec();
```
 - **ros::Duration**

```
ros::Duration duration(0.5); // 0.5s
```
 - **ros::Rate**

```
ros::Rate rate(10); // 10Hz
```
- If wall time is required, use `ros::WallTime`, `ros::WallDuration`, and `ros::WallRate`

More info

<http://wiki.ros.org/Clock>

<http://wiki.ros.org/roscpp/Overview/Time>

ROS Bags

- A *bag* is a format for storing message data
- Binary format with file extension `*.bag`
- Suited for logging and recording datasets for later visualization and analysis

Record all topics in a bag

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

Record given topics

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

Stop recording with `Ctrl + C`

Bags are saved with start date and time as file name in the current folder (e.g. `2017-02-07-01-27-13.bag`)

Show information about a bag

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

Read a bag and publish its contents

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

Playback options can be defined e.g.

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

`--rate=factor`

Publish rate factor

`--clock`

Publish the clock time (set param `use_sim_time` to true)

`--loop`

Loop playback

etc.

More info

<http://wiki.ros.org/rosbag/Commandline>

Debugging Strategies

Debug with the tools you have learned

- Compile and run code often to catch bugs early
- Understand compilation and runtime error messages
- Use analysis tools to check data flow (rosnode info, rostopic echo, rosrun, rqt_graph etc.)
- Visualize and plot data (RViz, RQT Multiplot etc.)
- Divide program into smaller steps and check intermediate results (ROS_INFO, ROS_DEBUG etc.)
- Make your code robust with argument and return value checks and catch exceptions
- If things don't make sense, clean your workspace
 - > `catkin clean --all`

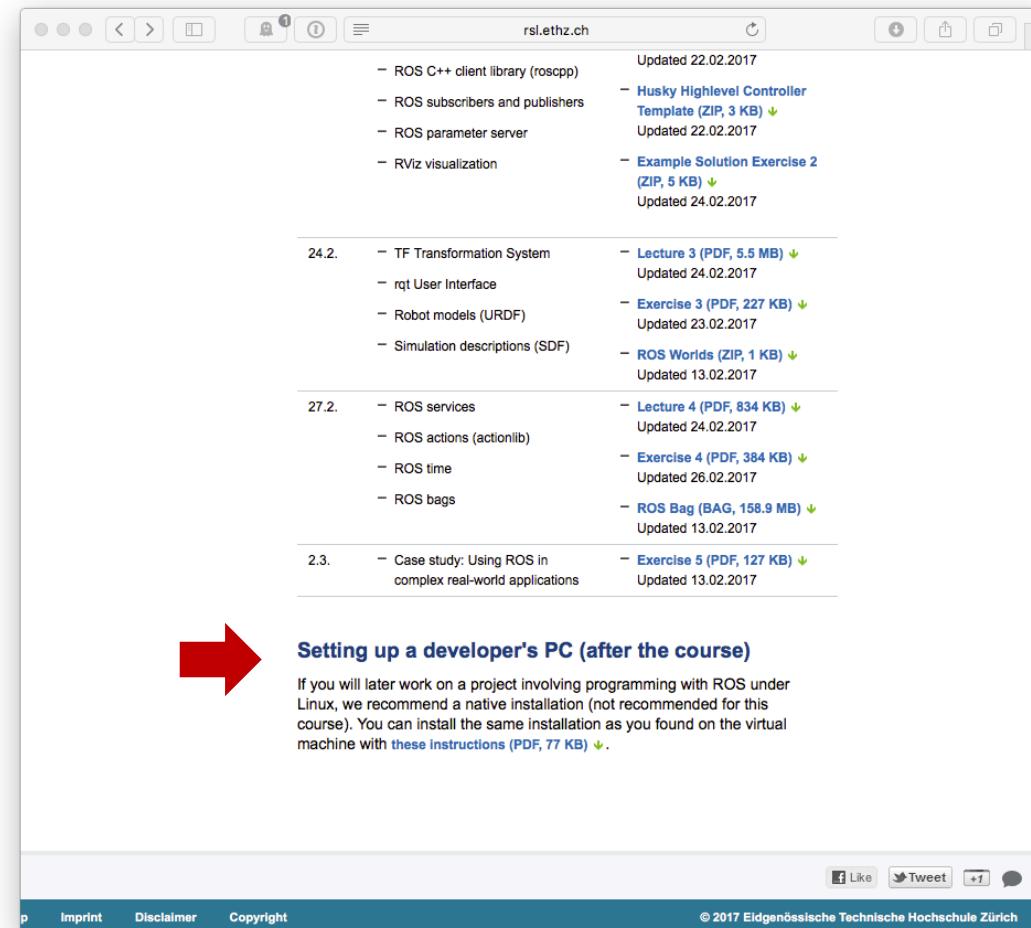
Learn new tools

- Build in *debug* mode and use GDB or Valgrind
 - > `catkin config --cmake-args -DCMAKE_BUILD_TYPE=Debug`
- Use Eclipse breakpoints
- Maintain code with unit tests and integration tests

More info

<http://wiki.ros.org/UnitTesting>
<http://wiki.ros.org/gtest>
<http://wiki.ros.org/rostest>
<http://wiki.ros.org/roslaunch/Tutorials/Roslaunch%20Nodes%20in%20Valgrind%20or%20GDB>

Setting Up up a Developer's PC



The screenshot shows a web browser window with the URL rsl.ethz.ch. The page displays a list of ROS-related files and resources, categorized by date. A red arrow points from the bottom left towards the 'Setting up a developer's PC (after the course)' section at the bottom of the page.

Date	File / Description	File Type	Size	Last Updated
22.02.2017	ROS C++ client library (roscpp)	–		
22.02.2017	Husky Highlevel Controller Template	(ZIP)	3 KB	Updated 22.02.2017
24.02.2017	Example Solution Exercise 2	(ZIP)	5 KB	Updated 24.02.2017
24.02.2017	TF Transformation System	–		
24.02.2017	rqt User Interface	–		
24.02.2017	Robot models (URDF)	–		
24.02.2017	Simulation descriptions (SDF)	–		
24.02.2017	Lecture 3 (PDF)	5.5 MB	Updated 24.02.2017	
24.02.2017	Exercise 3 (PDF)	227 KB	Updated 23.02.2017	
24.02.2017	ROS Worlds (ZIP)	1 KB	Updated 13.02.2017	
27.02.2017	ROS services	–		
27.02.2017	ROS actions (actionlib)	–		
27.02.2017	ROS time	–		
27.02.2017	ROS bags	–		
27.02.2017	Lecture 4 (PDF)	834 KB	Updated 24.02.2017	
27.02.2017	Exercise 4 (PDF)	384 KB	Updated 26.02.2017	
27.02.2017	ROS Bag (BAG)	158.9 MB	Updated 13.02.2017	
2.3.	Case study: Using ROS in complex real-world applications	–		
2.3.	Exercise 5 (PDF)	127 KB	Updated 13.02.2017	

Setting up a developer's PC (after the course)
If you will later work on a project involving programming with ROS under Linux, we recommend a native installation (not recommended for this course). You can install the same installation as you found on the virtual machine with [these instructions \(PDF, 77 KB\)](#).

Further References

- **ROS Wiki**
 - <http://wiki.ros.org/>
- **Installation**
 - <http://wiki.ros.org/ROS/Installation>
- **Tutorials**
 - <http://wiki.ros.org/ROS/Tutorials>
- **Available packages**
 - <http://www.ros.org/browse/>
- **ROS Cheat Sheet**
 - https://github.com/ros/cheatsheet/releases/download/0.0.1/ROScheatsheet_catkin.pdf
- **ROS Best Practices**
 - https://github.com/ethz-asl/ros_best_practices/wiki
- **ROS Package Template**
 - https://github.com/ethz-asl/ros_best_practices/tree/master/ros_package_template

Part 5 – Case Study

Using ROS in Complex Real-World Applications

P. Fankhauser, R. Diethelm, S. Bachmann, C. Gehring, M. Wermelinger, D. Bellicoso, V. Tsounis, A. Lauber, M. Bloesch, P. Leemann, G. Hottiger, D. Jud, R. Kaestner, L. Isler, M. Hoepflinger, R. Siegwart, and M. Hutter, “**ANYmal at the ARGOS Challenge – Tools and Experiences from the Autonomous Inspection of Oil & Gas Sites with a Legged Robot**,” in *ROSCon*, 2016.

Presentation



ROSCon 2016 SEOUL

ANYmal at the ARGOS Challenge
Tools and Experiences from the Autonomous Inspection of Oil & Gas Sites with a Legged Robot

Peter Fankhauser
Remo Diethelm, Samuel Bachmann, Christian Gehring, Martin Wermelinger, Dario Bellicoso, Vassilios Tsounis, Andreas Lauber, Michael Bloesch, Philipp Leemann, Gabriel Hottiger, Dominik Jud, Ralf Kaestner, Linus Isler, Mark Hoepflinger, Roland Siegwart, Marco Hutter

Video Recordings Supported By **ubuntu®**

<http://roscon.ros.org/2016>

<https://vimeo.com/187696096>

Slides



ANYmal at the ARGOS Challenge
Tools and Experiences from the Autonomous Inspection of Oil & Gas Sites with a Legged Robot

Péter Fankhauser
Remo Diethelm, Samuel Bachmann, Christian Gehring, Martin Wermelinger, Dario Bellicoso, Vassilios Tsounis, Andreas Lauber, Michael Bloesch, Philipp Leemann, Gabriel Hottiger, Dominik Jud, Ralf Kaestner, Linus Isler, Mark Hoepflinger, Roland Siegwart, Marco Hutter

RSL Robotic Systems Lab

Péter Fankhauser | 08.10.2016 | 1

<https://www.researchgate.net/publication/308953021>

Contact Information

ETH Zurich

Robotic Systems Lab
Prof. Dr. Marco Hutter
LEE J 225
Leonhardstrasse 21
8092 Zurich
Switzerland

<http://www.rsl.ethz.ch>

Lecturers

Péter Fankhauser (pfankhauser@ethz.ch)
Dominic Jud
Martin Wermelinger

Course website:

[http://www.rsl.ethz.ch/education-
students/lectures/ros.html](http://www.rsl.ethz.ch/education-students/lectures/ros.html)