Introduction to ROS: part B

Robot Programming and Control
Accademic Year 2021-2022

Diego Dall'Alba diego.dallalba@univr.it
Department of Computer Science – University of Verona
Altair Robotics Lab

Let’s take a quick look to the results of the survey

ROS Build System

CATKIN BUILDSYSTEM = CMake + python

catkin is the official build system of ROS that combines CMake macros and Python scripts to provide some functionality on top of CMake's normal workflow (improved automatic dependencies management and compilation of large project)

It is essential to know catkin build process for proficiently use ROS build system, having a good knowledge of CMake is also helping a lot in solving many problem when working in ROS

Please keep separate catkin workspace when you use catkin_make and where you use catkin command line tools (e.g. catkin init ; catkin build).

Many tutorial available online use catkin_make, even if I strongly suggest using catkin build

NEVER MIX THE TWO COMMANDS IN THE SAME WORKSPACE

```
sudo apt-get install python3-catkin-tools
```

Example of creating of a new catkin workspace using command line tools

```
source /opt/ros/kinetic/setup.bash
mkdir -p /tmp/quickstart_ws/src        # Make a new workspace
cd /tmp/quickstart_ws                  # Navigate to the workspace root
catkin init                            # Initialize it
cd /tmp/quickstart_ws/src              # Navigate to the source space
catkin create pkg pkg_a                # Populate the source space
catkin create pkg pkg_b
catkin create pkg pkg_c --catkin-deps pkg_a
catkin create pkg pkg_d --catkin-deps pkg_a pkg_b
catkin list                            # List the packages in the workspace
catkin build                           # Build all packages in the workspace
source /tmp/quickstart_ws/devel/setup.bash
```

ROS Hands on Demo

```
es  Terminal ▾                ven 14:55
                roscore http://victors:11311/
File  Edit  View  Search  Terminal  Help
ai-ray@victors:~$ roscore
... logging to /home/ai-ray/.ros/log/4699893e-522a-11e9-ad61-
unch-victors-2205.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://victors:41423/
ros_comm version 1.14.3


SUMMARY
========

PARAMETERS
 * /rosdistro: melodic
 * /rosversion: 1.14.3

NODES

auto-starting new master
process[master]: started with pid [2216]
ROS_MASTER_URI=http://victors:11311/

setting /run_id to 4699893e-522a-11e9-ad61-0800271b6865
process[rosout-1]: started with pid [2227]
started core service [/rosout]
```

```
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
```

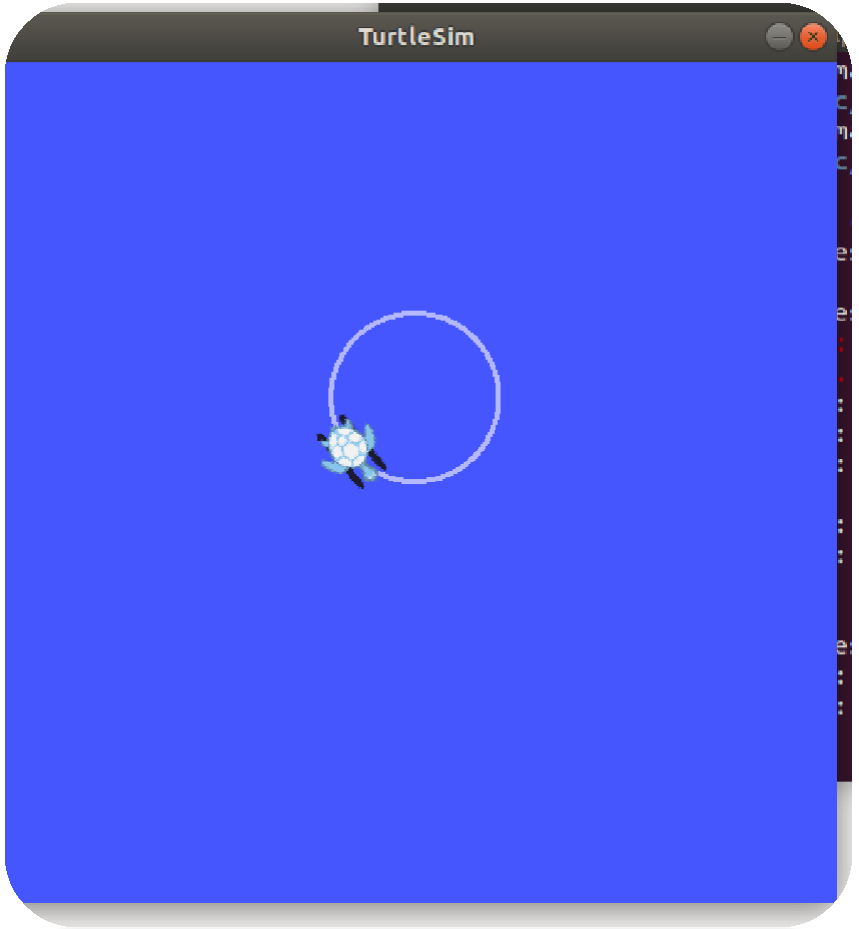
```
student@ubuntu:~/catkin_ws$ rosrun roscpp_tutorials listener
[ INFO] [1486053802.204104598]: I heard: [hello world 19548]
[ INFO] [1486053802.304538827]: I heard: [hello world 19549]
[ INFO] [1486053802.403853395]: I heard: [hello world 19550]
[ INFO] [1486053802.504438133]: I heard: [hello world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

Exercises C

turtlesim_node

turtle_teleop_key

- Install the following Ubuntu package:
  ros-kinetic-ros-tutorials
- Understand *«turtlesim»* package
  - navigate package contents
  - run different nodes
  - understand the communication architecture
- Create a separate package in your catkin workspace able to move the turtle on a circular trajectory
- Use roslauch to set parameters (radius and speed) → see later



In previous ROS Lessons

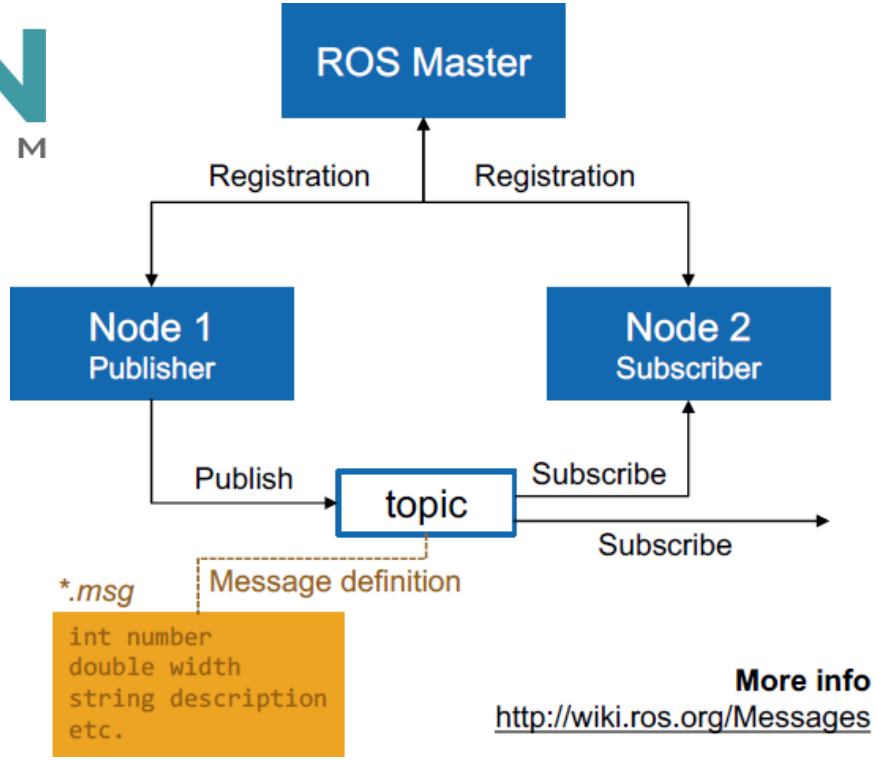
- ROS architecture & philosophy
- ROS master, nodes, and topics
- Catkin workspace and build system
- ROS package structure
- Console commands
- ROS C++ client library
- ROS subscribers and publishers

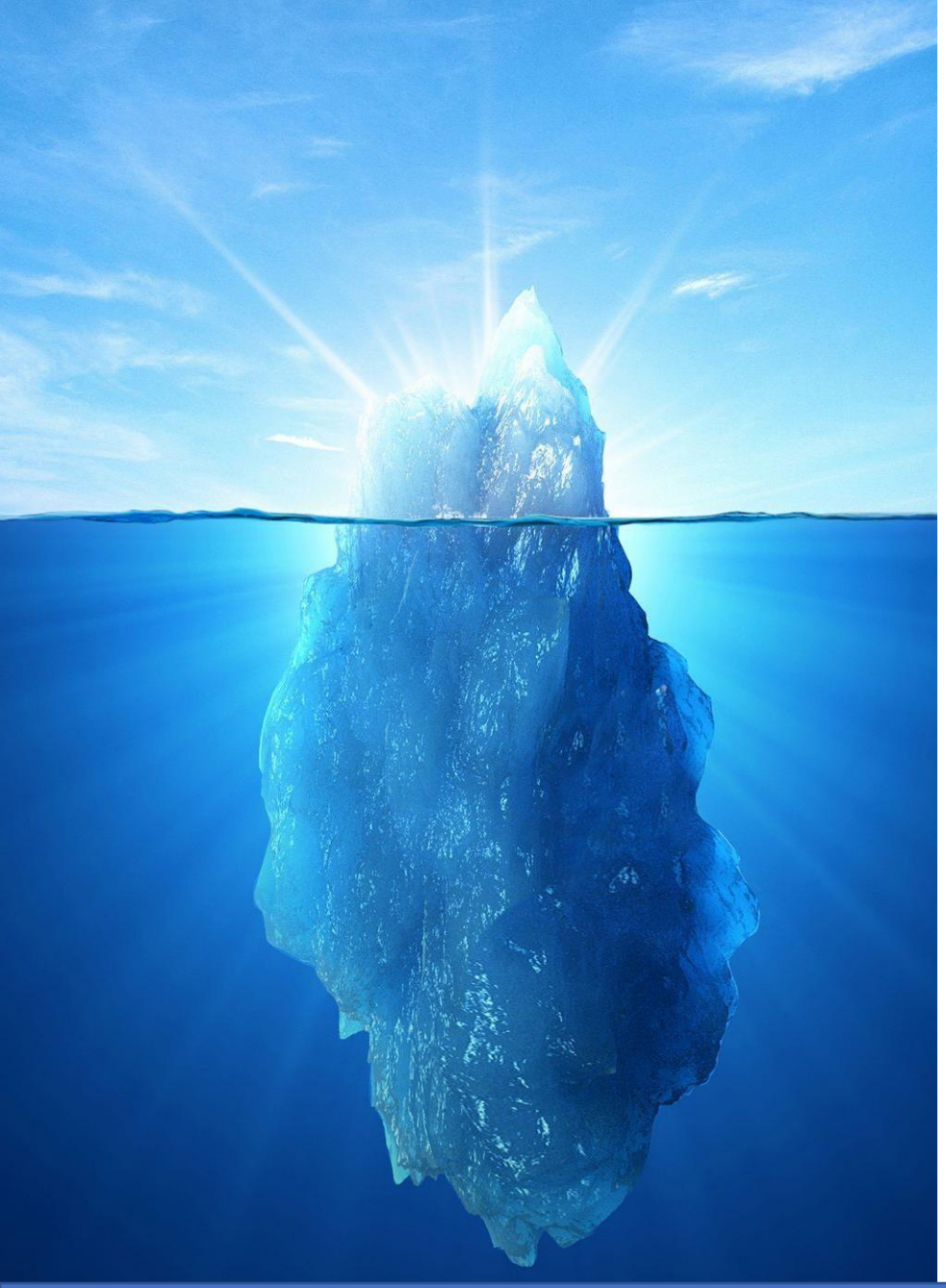
Applications

ROS

Operating System (Linux Ubuntu)

CATKIN BUILDSYSTEM

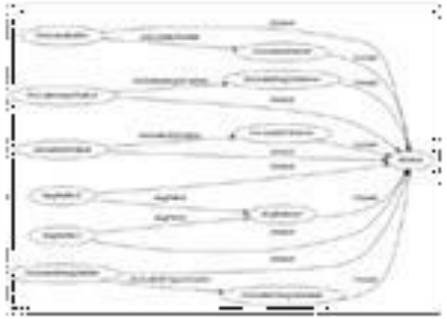




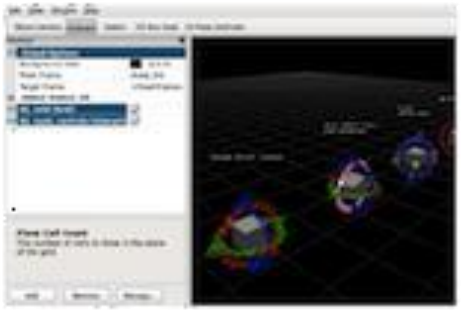
Overview

- ROS services
- ROS actions (actionlib)
- Launch-files
- ROS parameter server
- Logging in ROS: rosbag
- Data visualization and user interface in ros: rqt

ROS Characteristics



+



+

+

Plumbing

- Process management
- Inter-process communication
- Device drivers

Tools

- Simulation
- Visualization
- Graphical user interface
- Data logging

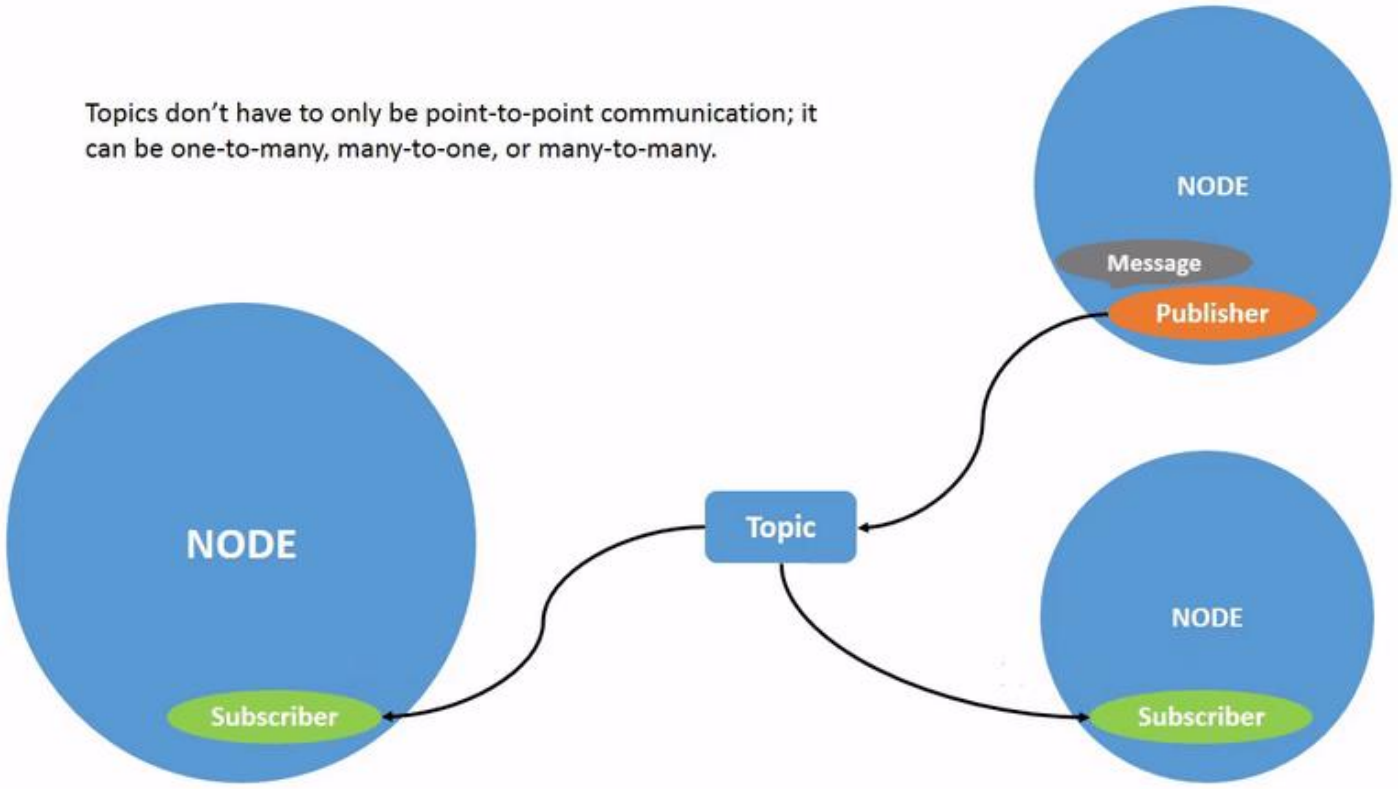
Capabilities

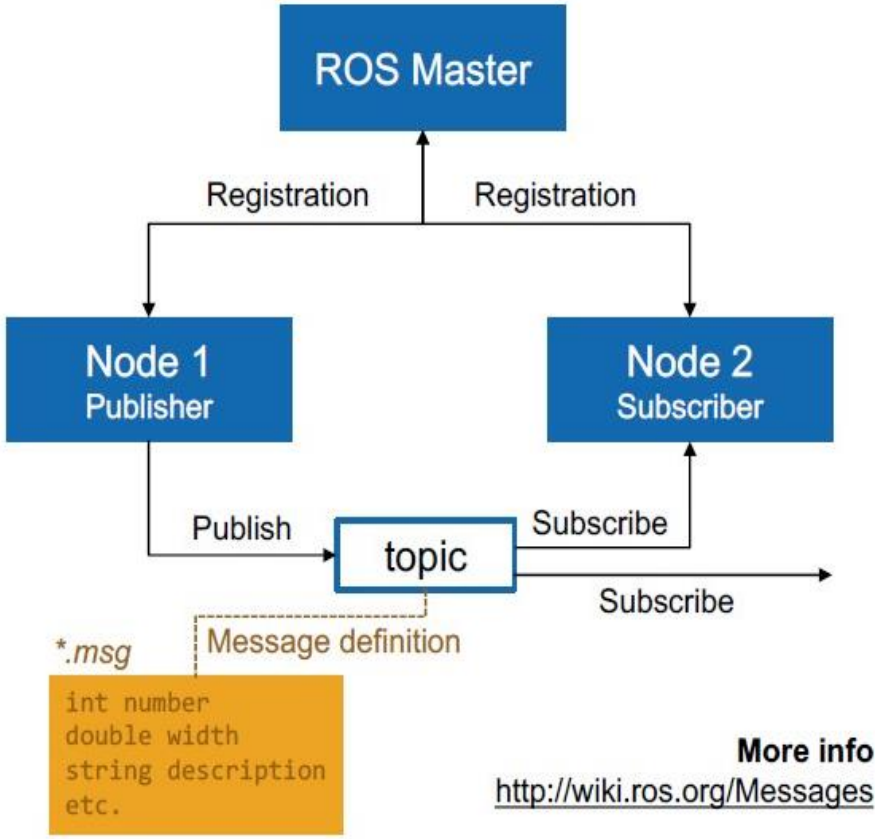
- Control
- Planning
- Perception
- Mapping
- Manipulation

Ecosystem

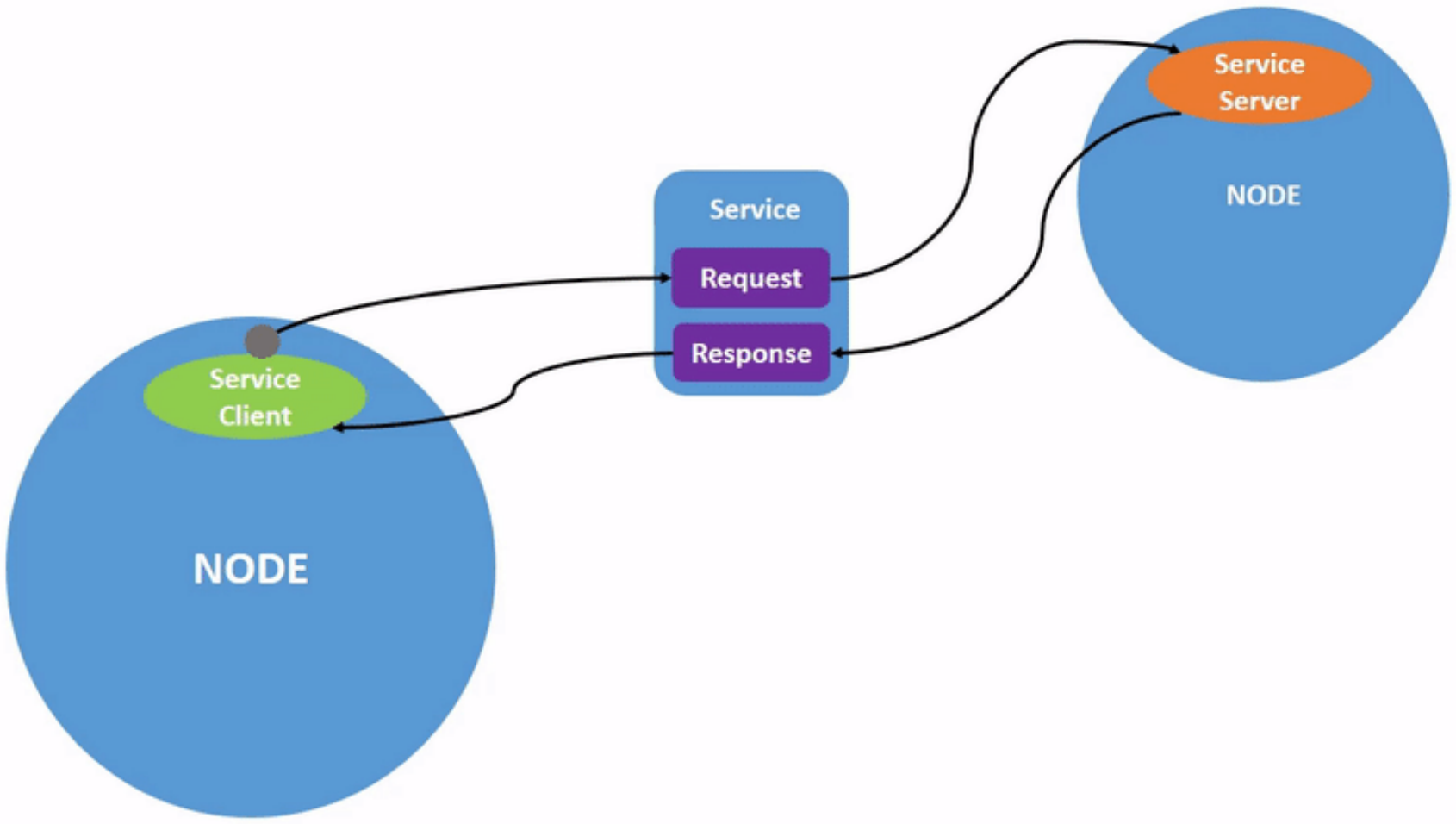
- Package organization
- Software distribution
- Documentation
- Tutorials

Quick recap on ROS Nodes and topic communication

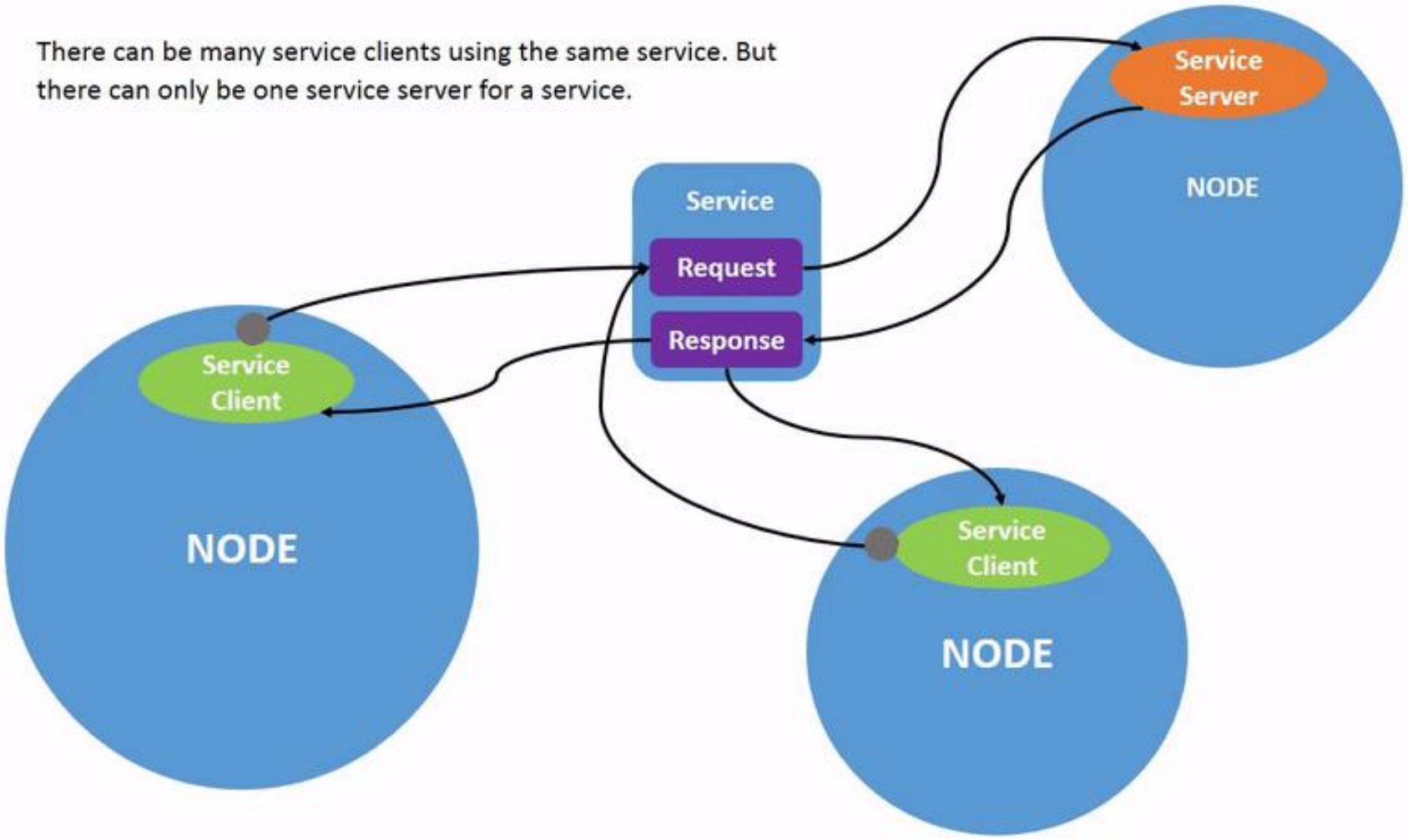




ROS Service communication (1)



ROS Service communication (1)



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

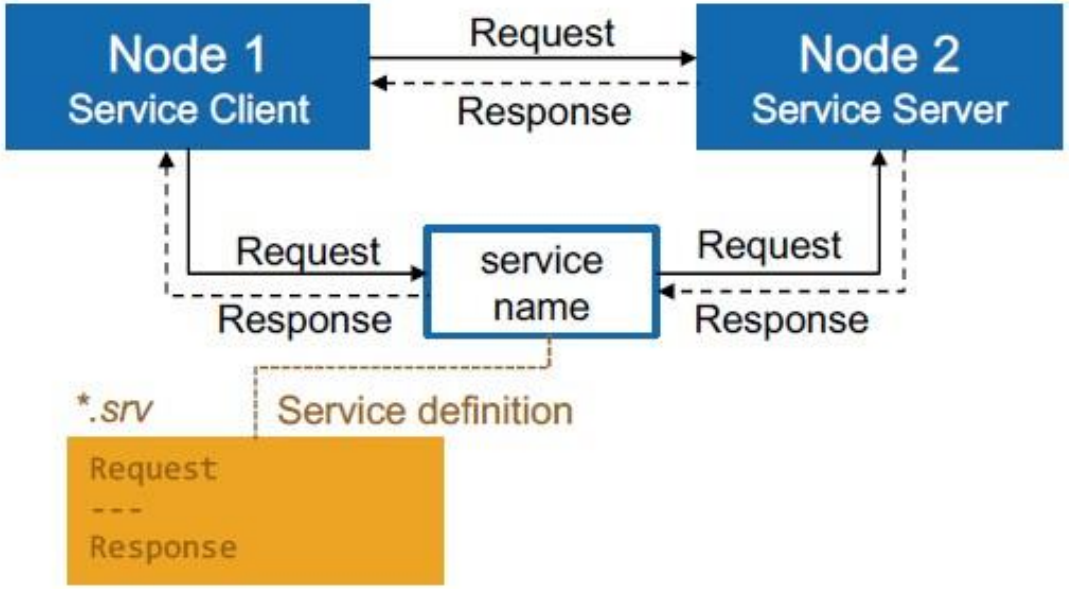
```
> rosservice list
```

Show the type of a service

```
> rosservice type /service_name
```

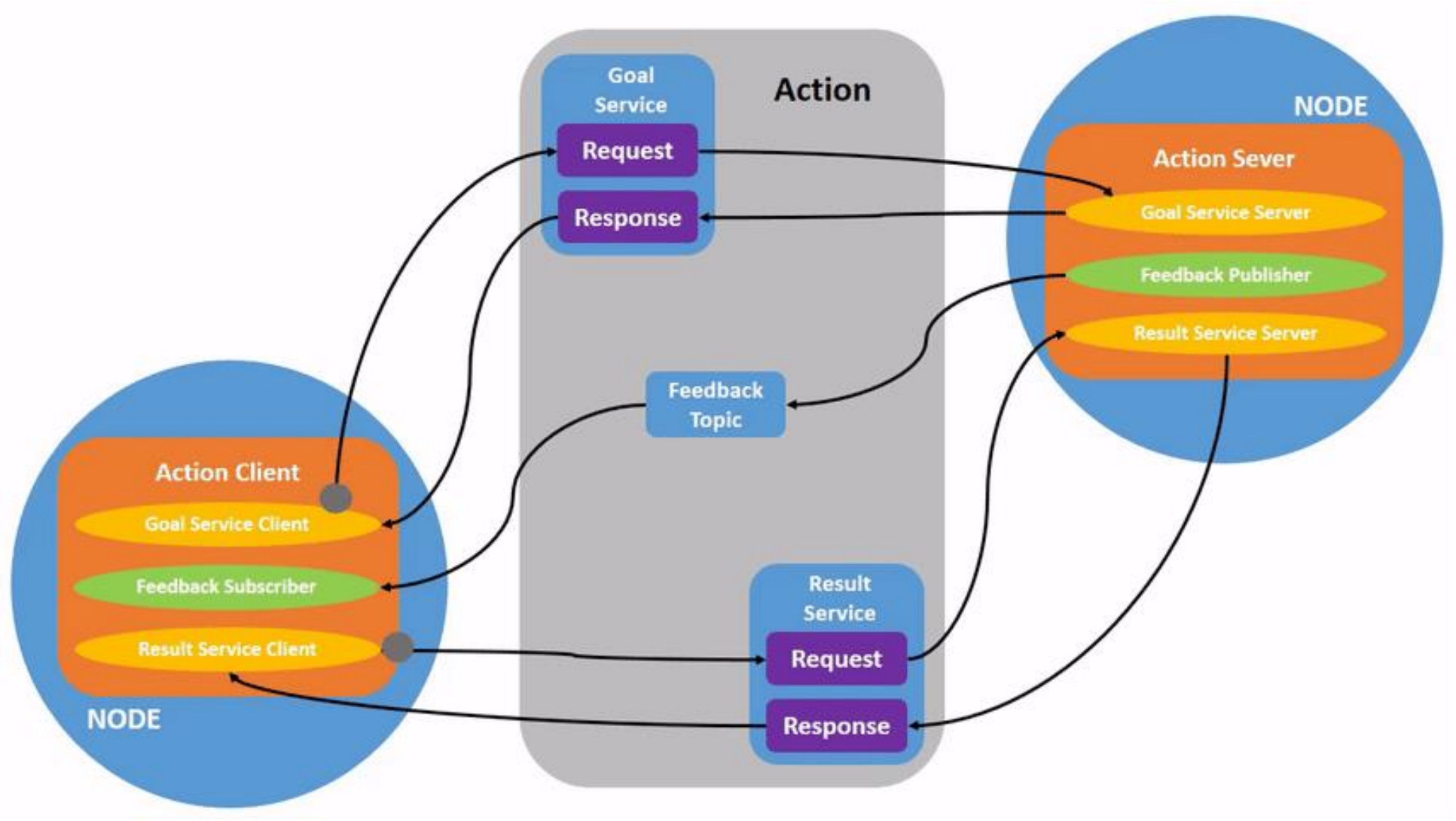
Call a service with the request contents

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



More info
http://wiki.ros.org/Services

ROS Actions Communication



ROS Actions

- 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

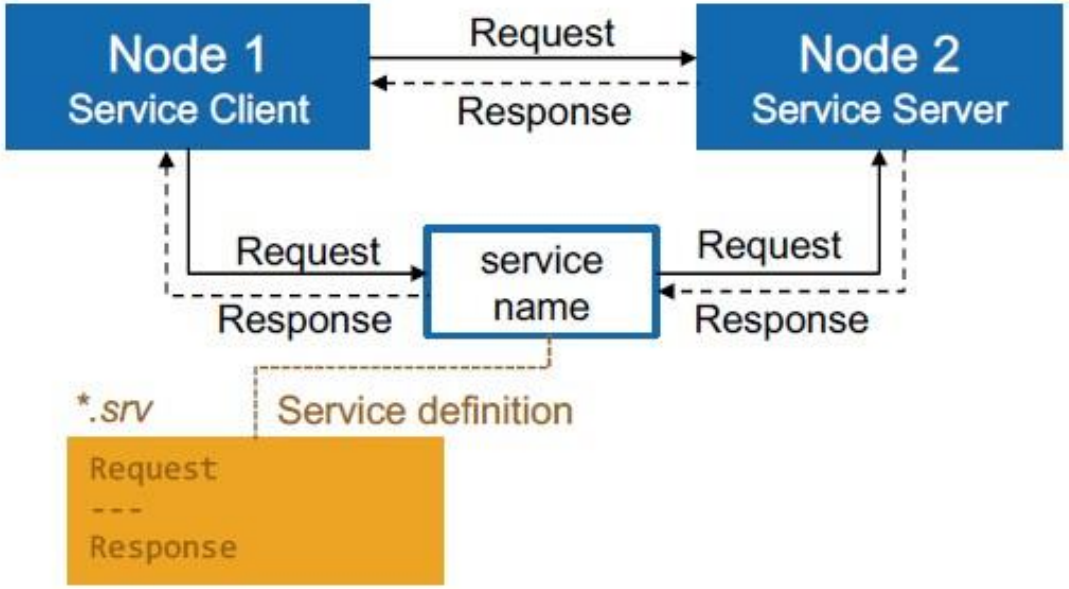
```
> rosservice list
```

Show the type of a service

```
> rosservice type /service_name
```

Call a service with the request contents

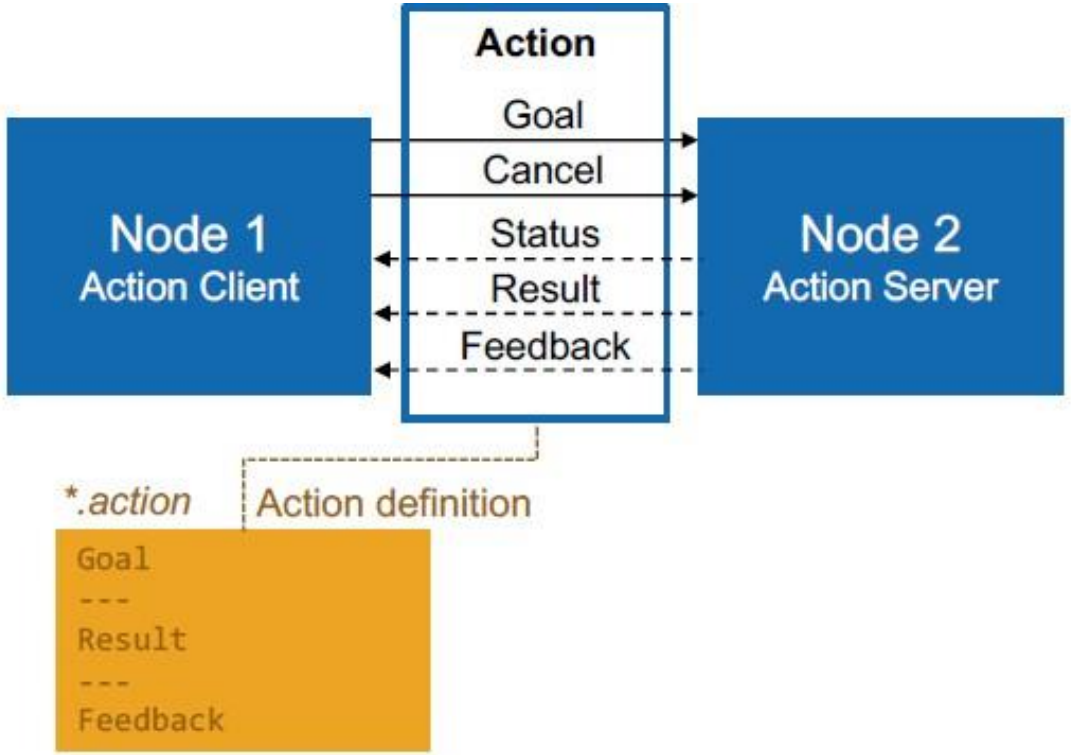
```
> rosservice call /service_name args
```



More info
http://wiki.ros.org/Services

ROS 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 Services and Actions Definition example

std_srvs/Trigger.srv

```
---
bool success
string message
```

Request

Response

nav_msgs/GetPlan.srv

```
geometry_msgs/PoseStamped start
geometry_msgs/PoseStamped goal
float32 tolerance
---
nav_msgs/Path plan
```

Averaging.action

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

Goal

Result

Feedback

FollowPath.action

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

ROS Launch: File format

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>
```

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

- **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

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: Parameter server and YAML format

- 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

```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

ROSCPP: Parameter server

- 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

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

http://wiki.ros.org/dynamic_reconfigure

ROSCPP: Node handle Types

- There are four main types of node handles
  1. Default (public) node handle:
     `nh_ = ros::NodeHandle();`
  2. Private node handle:
     `nh_private_ = ros::NodeHandle("~");`
  3. Namespaced node handle:
     `nh_eth_ = ros::NodeHandle("eth");`
  4. 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 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

Rqt visualizer & user interface (1)

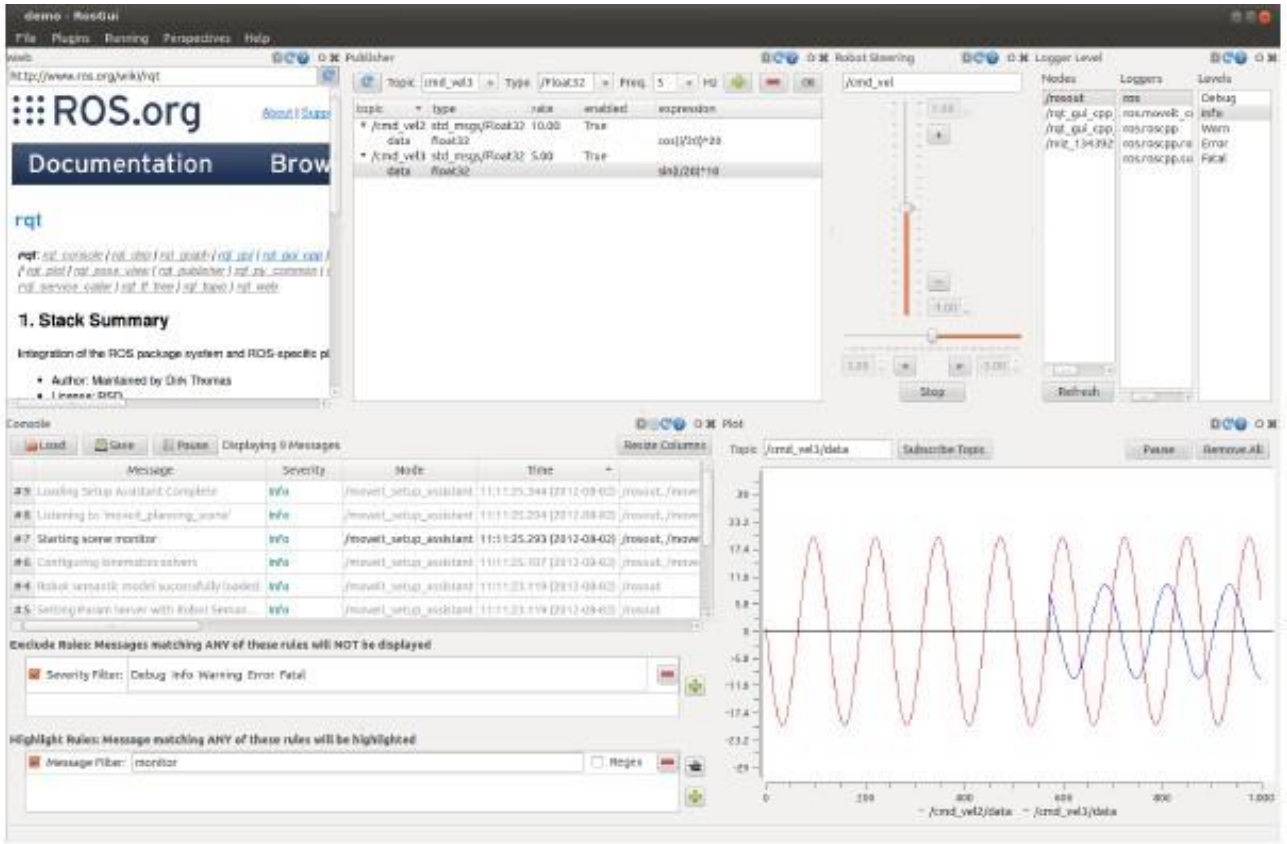
- User interface base on Qt
- Custom interfaces can be setup
- Lots of existing plugins exist
- Simple to write own plugins

Run RQT with

```
> rosrun rqt_gui rqt_gui
```

or

```
> rqt
```



More info
http://wiki.ros.org/rqt/Plugins

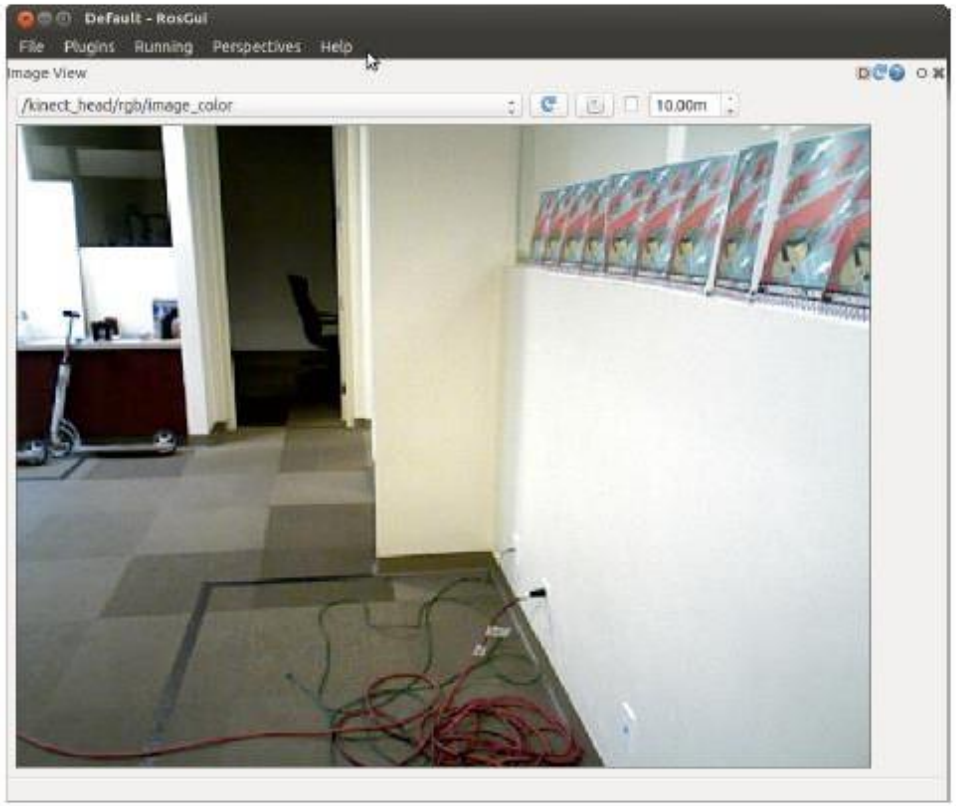
Rqt visualizer & user interface (2)

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

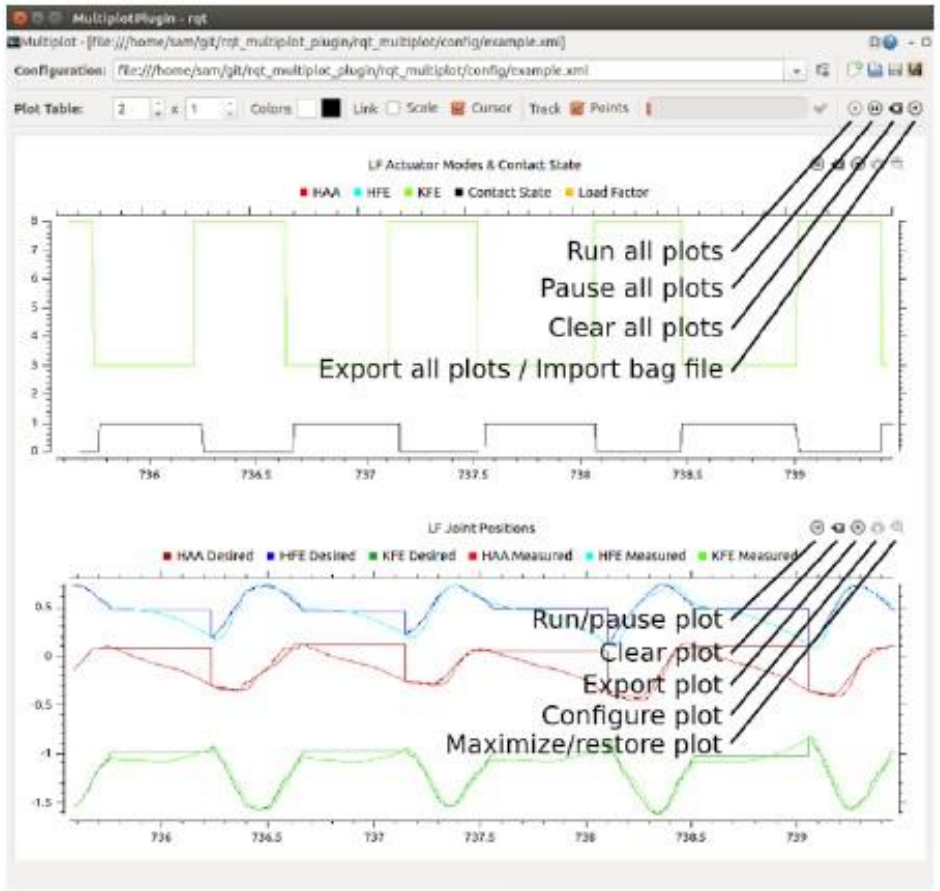
Rqt visualizer & user interface (3)

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

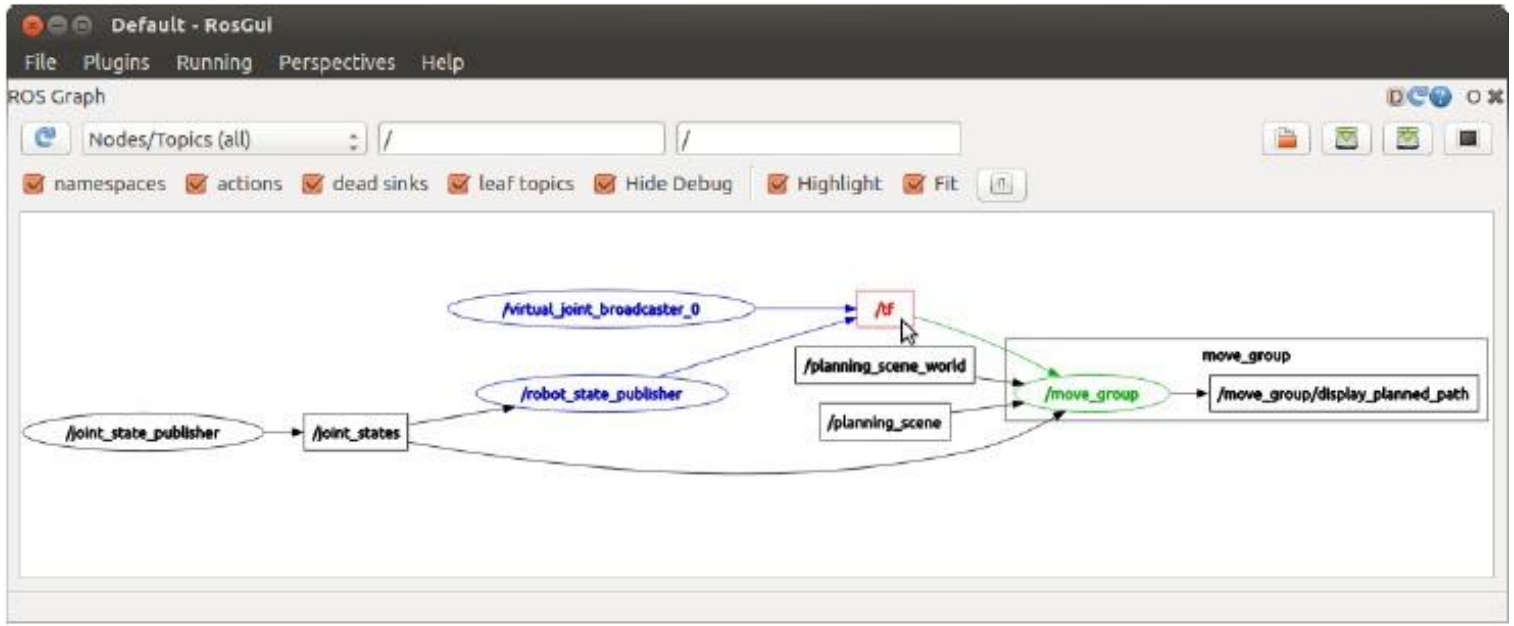
Rqt visualizer & user interface (4)

rqt_graph

- Visualizing the ROS computation graph

Run *rqt_graph* with

```
> rosrun rqt_graph rqt_graph
```



More info
http://wiki.ros.org/rqt_graph

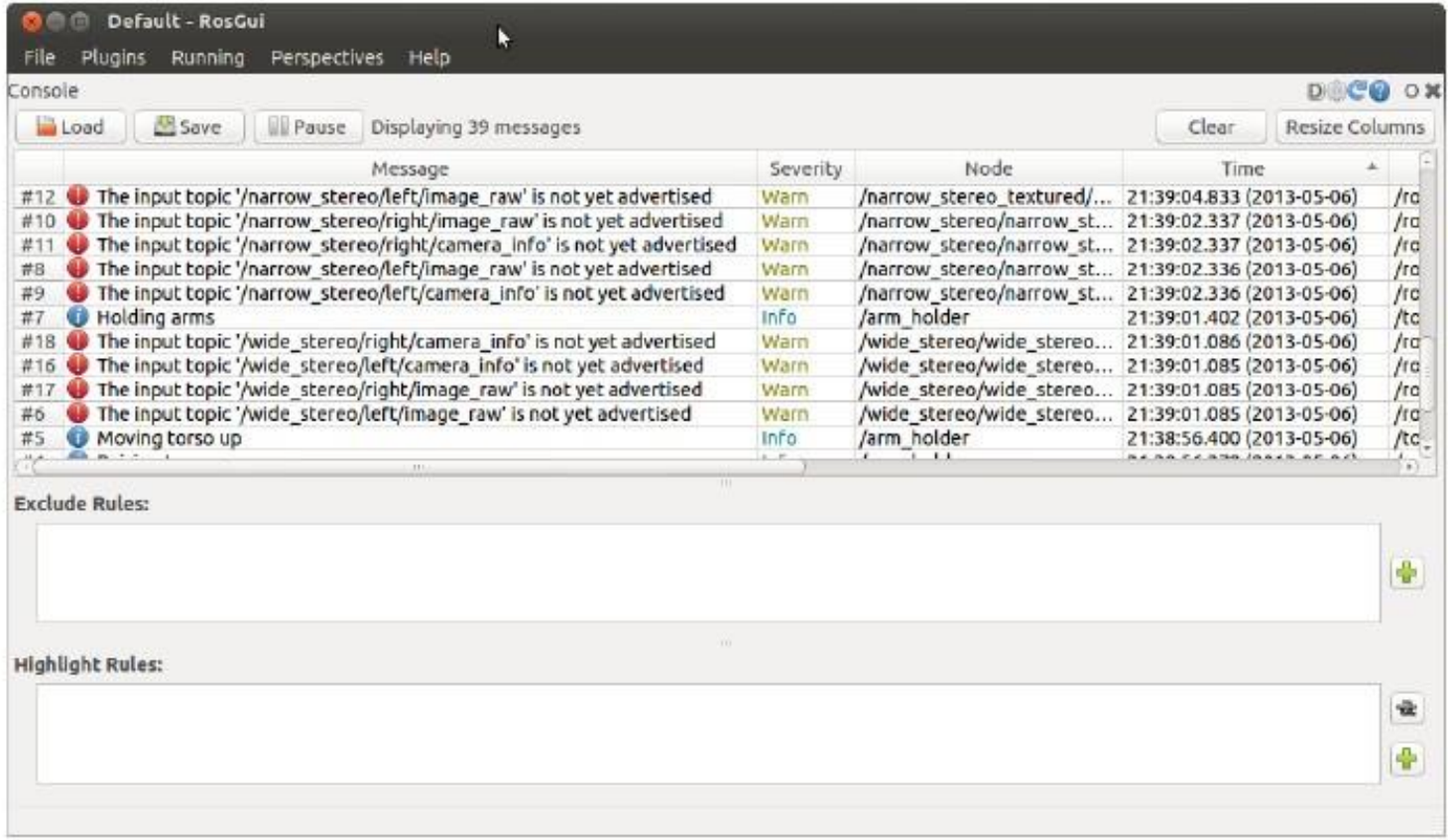
Rqt visualizer & user interface (5)

rqt_console

- Displaying and filtering ROS messages

Run *rqt_console* with

```
> rosrun rqt_console rqt_console
```



More info
http://wiki.ros.org/rqt_console

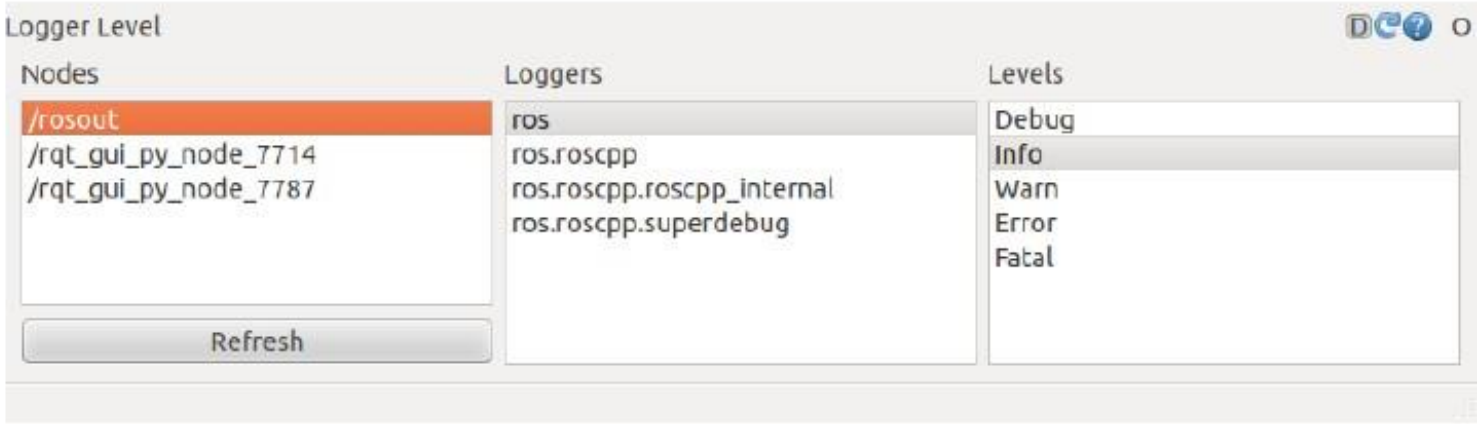
Rqt visualizer & user interface (6)

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

Questions?

The contents of these slides are partially based on:

Programming for Robotics - Introduction to ROS

February 2017

DOI: 10.13140/RG.2.2.14140.44161

Affiliation: Robotics Systems Lab, ETH Zurich

Péter Fankhauser · Dominic Jud · Martin Wermelinger · Marco Hutter